

Dr. Thomas Erler
Dr. Michael Ricken

UML

Mehr als 750
Seiten + CD-ROM

€ 16,95 ^(D)



CD

DAS_{bhv} TASCHEN
BUCH



Dr. Thomas Erler
Dr. Michael Ricken

UML

Das bhv Taschenbuch



Die Informationen im vorliegenden Buch werden ohne Rücksicht auf einen eventuellen Patentschutz veröffentlicht.

Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt.

Bei der Zusammenstellung von Texten und Abbildungen sowie Material auf dem beiliegenden Datenträger wurde mit größter Sorgfalt vorgegangen. Trotzdem können Fehler nicht vollständig ausgeschlossen werden. Verlag, Herausgeber und Autoren können für fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung übernehmen.

Für Verbesserungsvorschläge und Hinweise auf Fehler sind Verleger und Herausgeber dankbar.

Alle Rechte vorbehalten, auch die der fotomechanischen Wiedergabe und der Speicherung in elektronischen Medien.

Die gewerbliche Nutzung der in diesem Buch und auf der beiliegenden CD gezeigten Modelle und Arbeiten ist nicht zulässig.

Dieses Buch wurde der Umwelt zuliebe auf chlorfrei gebleichtem Papier gedruckt.

Copyright © 2002 by
verlag moderne industrie
Buch AG & Co. KG, Landsberg
Königswinterer Str. 418
D – 53227 Bonn
www.vmi-Buch.de

1. Auflage

06 05 04 03 02
10 9 8 7 6 5 4 3 2 1

ISBN 3-8266-8066-9

Printed in Italy

Inhaltsverzeichnis

Vorwort	13

Teil I: Überblick – Vom objektorientierten Denken zur UML	15

1 Grundlagen der Objektorientierung	17
Zum Einstieg ein »Bild«	22
Objekte als kooperierende Handlungsträger	22
Analogie von Objekten und Kooperationspartnern	23
Denken und Ordnung	25
Denken und Modellieren	27
Gegenstände und ihre Merkmale	29
Gegenstände und Kategorien	30
Kategorien und Taxonomien	31
Die frühen Ansätze	31
Simulationen mit Simula – die Evolution	32
Kommunikation mit Smalltalk – die Revolution	36
Entwicklung im Abriss	43
Objekte	48
Was ist ein Objekt?	49
Modellierung mit Objekten	50
Name – eindeutige Identifikation von Objekten	53
Zustand – statische Objekteigenschaften	54
Verhalten – dynamische Objekteigenschaften	56
Datenkapselung	57
Klassen	59
Konkrete Klassen	60
Abstrakte Klassen	61
Vererbung	63
Polymorphie	66
Objektorientierte Systementwicklung	68

2 Grundlagen der UML	71
Motivation	73
Historie	77
Spracharchitektur	78
Diagrammarten	82

Teil II: Techniken – Modelle der UML **85**

3 Überblick	87
Außensicht (Use-Case-Diagramme)	89
Akteure	90
Anwendungsfälle	91
Innensicht	97
Jeder sieht's anders	98
Vom Groben zum Detail oder doch umgekehrt?	100
UML-Kerndiagramme im Überblick	102
4 Diagrammarten	109
Statik: Klassen- und Objektdiagramme	111
Klasse – Bauplan für Exemplare	112
Objekt – das konkrete Exemplar einer Klasse	115
Attribut – identitätslose Eigenschaft	118
Operationen – Verhalten eines Objektes	121
Sichtbarkeit – wer sieht was von wo?	123
Klassenattribute und Klassenoperationen – Arbeiten ohne Objekte	125
Schnittstelle – verkümmerte Klassen	127
Vererbung – verteilte Baupläne	130
Assoziation – Attribute mit Identität	145
Aggregation – Teile eines Ganzen	162
Komposition – die strenge Aggregation	168
Assoziation oder Vererbung?	177
Pakete – Ordnung schaffen	179
Abhängigkeiten – Änderungen beherrschen	181

Statik: Implementierungsdiagramme	186
Komponentendiagramm – realisierte Einheiten	187
Verteilungsdiagramme – Träger von Komponenten	189
Dynamik: Verhaltensdiagramme	191
Sequenzdiagramme	192
Kollaborationsdiagramme	202
Zustandsdiagramme	208
Aktivitätsdiagramme	220

Teil III: Know-How – UML im Software-Entwicklungsprozess

231

5 Der Entwicklungsprozess – wie Software entsteht	233
Sichtweisen von Software-Entwicklung	235
Die Sicht des kreativen Künstlers	235
Die Sicht des verfahrensorientierten Ingenieurs	236
Die Sicht des teamorientierten Designers	236
Die Beteiligten – wer entwickelt?	238
Die Aktivitäten – wie geht man vor?	240
Die Ergebnisse – was wird entwickelt?	244
6 Fallbeispiel: »Mein liebes Geld«	247
Ausgangssituation	249
Analyse – Anforderungen	250
Analyse – fachliche Beschreibung	250
Anwendungsfälle	250
Klassenstruktur	253
Typische Sequenz der Depotverwaltung	257
Entwurf – technische Konstruktion	259
Verfeinerung und Erweiterung	259
Datenhaltung	261
Benutzungsoberfläche	261

7 Implementierung von UML-Modellen	265
»Mein liebes Geld« – Sparkonto in C#	268
Assoziation – »LAG« von Objekten	271
Phasen einer Objektbeziehung	271
Beziehung aufbauen	272
Beziehung leben	276
Beziehung beenden	277
Beziehungsmengen – Kardinalität	278
Beziehungsarten	285
Vererbung – Weitergabe von Strukturen	287
Identifizieren von Vererbungsstrukturen	287
Redefinition – Operationen werden zu Spezialisten	299
Schnittstellen – mehr Schein als Sein	304
Vereinbarung von Schnittstellen	304
Vererbung und Implementierung von Schnittstellen	305
Schnittstellentyp als Attributtyp	312
Depotverwaltung – Schnittstellen im Überblick	315
Namensräume – Klassen organisieren	319
using – Einbinden von Namensräumen	320
namespace – Zuordnen zu Namensräumen	321
Sichtbarkeit von Elementen	326
Depotverwaltung – Sichtbarkeiten	329

Teil IV: Praxis – Spezialaspekte und Werkzeuge der UML

333

8 Checklisten für die objektorientierte Analyse	335
Problemstellung	337
Entwicklung objektorientierter Modelle	338
Grobe Systemanalyse	342
Checkliste Anwendungsfall	344
Checkliste Problemabgrenzung	348
Checkliste Teilsysteme	350
Checkliste Schnittstellen	353

Statische Analyse	355
Checkliste Klassenbildung	356
Checkliste Klassenbeschreibung	362
Checkliste Klassenstruktur	365
Checkliste Klassenbeziehungen	371
Dynamische Analyse	374
Checkliste Interaktionen	376
Checkliste Zustände	381
Checkliste Ereignisse	385
Checkliste Aktivitäten	387
Checkliste Operationen	389
9 Modellierung von Geschäftsprozessen	393
Grobe Systemanalyse	395
Beschreibung des Geschäftsprozesses	396
Abgrenzung des relevanten Problemausschnitts	405
Abgrenzung von Teilsystemen (Paketen)	408
Definition von Schnittstellen zwischen den Paketen	412
Statisches Modell	413
Identifikation von Elementen und Strukturen	413
Beschreibung von Klassen und Attributen	428
Ermittlung von Vererbungsstrukturen	430
Identifikation von Assoziationen	433
Dynamisches Modell	438
Identifikation von Interaktionen	438
Definition von Zuständen und Zustandsübergängen	445
Identifikation von Ereignissen	450
Identifikation von Aktivitäten	453
Beschreibung von Operationen	457
10 Fallbeispiel: »Passagierabfertigung«	463
Grobe Systemanalyse	465
Beschreibung des Anwendungsfalls	465
Abgrenzung des relevanten Problemausschnitts	468
Abgrenzung von Teilsystemen (Paketen)	471
Definition von Schnittstellen zwischen Paketen	474

Statisches Modell	477
Identifikation von Elementen und Strukturen	478
Beschreibung von Klassen und Attributen	486
Ermittlung von Vererbungsstrukturen	488
Identifikation von Assoziationen	492
Dynamisches Modell	497
Identifikation von Interaktionen	498
Definition von Zuständen und Zustandsübergängen	500
Identifikation von Ereignissen	506
Identifikation von Aktivitäten	507
Beschreibung von Methoden	510

11 Werkzeuge der objektorientierten Modellierung	515
Together	517
Einrichten eines Projekts	519
Erstellen von Klassendiagrammen	524
Vom Klassen- zum Objektdiagramm	530
Vom Klassen- zum Zustandsdiagramm	536
Vom Klassen- zum Sequenzdiagramm	541
Rational Rose	548
Überblick	548
Klassendiagramme erstellen	557
System Architect	568
MagicDraw	578
Einrichten eines Projekts	578
Erstellen von Klassendiagrammen	579
Vom Klassen- zum Objektdiagramm	592
Vom Klassen- zum Zustandsdiagramm	598
Vom Klassen- zum Sequenzdiagramm	605
Poseidon	609
Einrichten eines Projektes	611
Erstellen von Klassendiagrammen	612
Vom Klassen- zum Objektdiagramm	619
Vom Klassen- zum Zustandsdiagramm	622
Vom Klassen- zum Sequenzdiagramm	627
Toolübersicht	631

Teil V: Anhang	637
A Die CD-ROM zum Buch	639
B Installation der Tools	643
Together	645
Rational Rose	658
System Architect	671
MagicDraw	680
Poseidon	690
C Internetadressen	701
D OCL-Spezifikation	711
OCL als Ergänzung der UML	713
Kontext und Invarianten	714
Vor- und Nachbedingungen	715
Basistypen und -operationen	716
E Überblick UML-Diagramme	725
F Glossar	743
Index	761



Vorwort

In der Praxis der Software-Entwicklung hat sich die *Unified Modeling Language (UML)* in den letzten Jahren als formale Sprache zur objektorientierten Spezifikation, Visualisierung, Konstruktion und Dokumentation von Softwaresystemen und Geschäftsmodellen durchgesetzt. Dieses Taschenbuch soll Ihnen die Anwendung der UML erleichtern. Es gibt Ihnen einen Überblick über die Grundlagen und die wesentlichen Konzepte der UML. Darüber hinaus enthält es viele praktische Beispiele zur Anwendung der UML.

Durch beliebte und verbreitete Programmiersprachen wie Java, C++ oder neuerdings C# hat die Objektorientierung in der Softwareentwicklung in letzter Zeit große Bedeutung erlangt. Ihre starke Verbreitung bei der Entwicklung von Internet-Applikationen führt auf dem Arbeitsmarkt zu einer enormen Nachfrage nach Programmierern mit entsprechenden Kenntnissen.

Der effiziente Einsatz objektorientierter Programmiersprachen scheitert jedoch häufig an mangelnden Kenntnissen der zugrunde liegenden objektorientierten Konzepte. Die Objektorientierung ist nämlich nicht nur eine spezielle Art zu programmieren. Sie ist vielmehr ein eigener Denkstil, eine bestimmte Herangehensweise an Problemstellungen. Der große Vorteil der Objektorientierung ist ihre große Ähnlichkeit mit den menschlichen Denkstrukturen. Dadurch wird auch und vor allem dem Einsteiger, der sich bisher wenig oder gar nicht mit der Programmierung beschäftigt hat, das Erlernen der Objektorientierung erleichtert.

Um erfolgreich zu programmieren, ist zunächst eine intensive Analyse der jeweiligen Problemstellung erforderlich. Die UML ist eine formale Sprache, die genau diese Phase der objektorientierten Softwareentwicklung unterstützt. Mit den Konzepten der UML können reale Problemstellungen abgebildet und strukturiert werden, um sie daraufhin in einer objektorientierten Programmiersprache zu implementieren. Die UML ist inzwischen zu einem Standard gewor-

den. Sie wird in vielen Projekten eingesetzt und durch zahlreiche softwaregestützte Werkzeuge unterstützt.

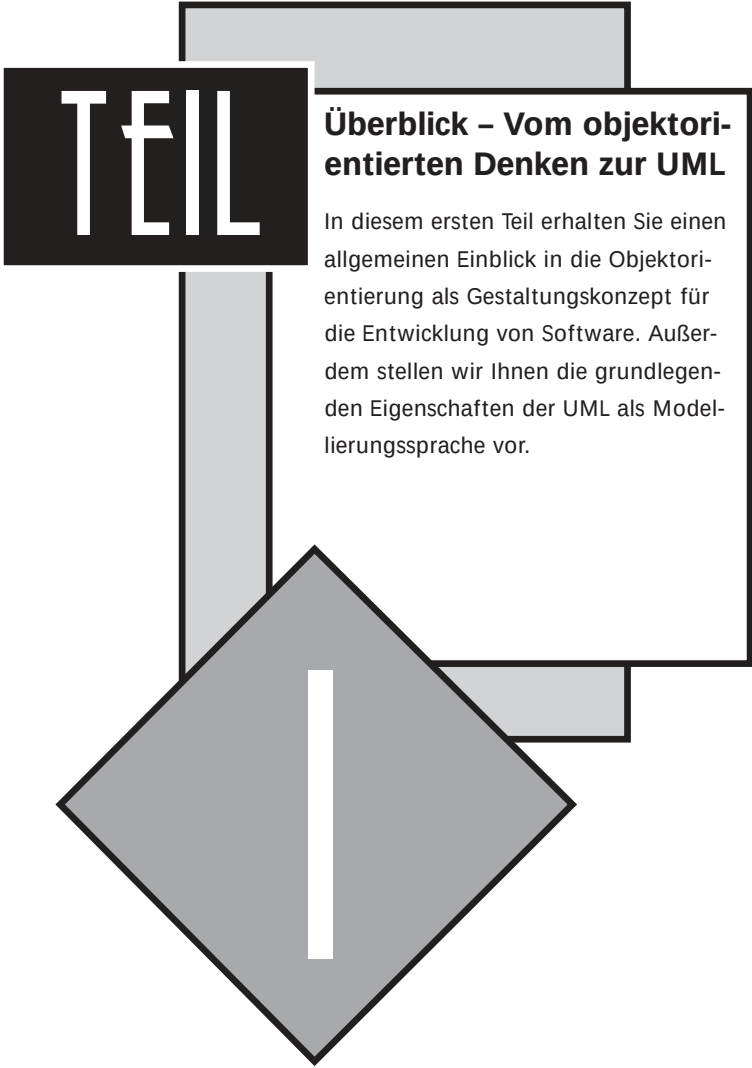
Dieses Taschenbuch macht Sie Schritt für Schritt mit den wesentlichen Konzepten der UML vertraut. Zahlreiche durchgängige Beispiele und mehrere Fallstudien veranschaulichen die vorgestellten Konzepte. Neben den Grundlagen der UML stellt dieses Buch auch wichtige Aspekte des Software Engineering vor und gibt Ihnen praktische Checklisten an die Hand, die Ihnen als Leitfaden bei der praktischen Umsetzung objektorientierter Software-Entwicklungsprojekte dienen sollen.

Durch die freundliche Unterstützung der Firmen Gentleware, Popkin, Togethersoft und Rational Software ist es uns möglich, Ihnen auf der beiliegenden CD-ROM einige softwaregestützte UML-Werkzeuge als Evaluierungsversionen zur Verfügung zu stellen. Installation und Handhabung der Tools werden im Buch erläutert. Die CD enthält darüber hinaus weitergehende Informationen zu den Tools sowie eine PDF-Version dieses Taschenbuchs, die Ihnen als mobiles Online-Nachschlagewerk dienen soll.

Sollten Sie weitergehende Fragen zu diesem Taschenbuch oder zur UML haben, besuchen Sie die Internet-Seiten zu diesem Buch unter <http://uml-buch.mindx.de> oder <http://www.stratim.de/uml-tb>. Dort finden Sie ständig aktualisierte Literaturhinweise, Linksammlungen zum Thema UML und Neuigkeiten zu den vorgestellten Tools. Über Anregungen und Verbesserungsvorschläge würden wir uns ebenfalls freuen.

Und nun wünschen wir Ihnen viel Spaß mit der UML und diesem Taschenbuch.

Thomas Erler und Michael Ricken



TEIL

Überblick – Vom objektorientierten Denken zur UML

In diesem ersten Teil erhalten Sie einen allgemeinen Einblick in die Objektorientierung als Gestaltungskonzept für die Entwicklung von Software. Außerdem stellen wir Ihnen die grundlegenden Eigenschaften der UML als Modellierungssprache vor.



KAPITEL

Grundlagen der Objektorientierung

Falls Sie bisher noch keine Erfahrung mit der Objektorientierung gesammelt haben, erhalten Sie in dem folgenden Kapitel einen kurzen Einblick in die objektorientierten Grundkonzepte.

1



Grundlagen der Objektorientierung

Die Objektorientierung ist ein hoch aktuelles Thema. Programmiersprachen wie Java oder C++ haben zu ihrer Verbreitung beigetragen. Programmierer mit Kenntnissen objektorientierter Sprachen sind auf dem Arbeitsmarkt sehr gefragt. Ein wesentlicher Grund dafür ist der Status von Java als *die* Programmiersprache für Internet-Applikationen. Der sich abzeichnende Erfolg von C#, der neuen Programmiersprache von Microsoft, wird diesen Trend sicher noch verstärken.



Die Bedeutung der Objektorientierung ist jedoch nicht auf ein reines Programmierkonzept beschränkt. Sie stellt vielmehr einen eigenen *Denkstil*, eine bestimmte Herangehensweise an Problemstellungen dar.

Die Objektorientierung bietet eine besondere Art, Problemstellungen zu strukturieren. Im weiteren Verlauf dieses Buchs werden Sie erkennen, dass diese stark an das menschliche Problemlösungsverhalten angelehnt ist. Menschen denken in Kategorien. Wir versuchen, unsere Erfahrungen mit bereits erprobten Lösungsmechanismen auf die Bewältigung neuer, bislang noch unbekannter Aufgaben zu übertragen. Neue Erfahrungen ordnen wir ähnlichen, bereits in unserem Gehirn gespeicherten Kategorien zu.

Insofern ist es gar nicht so abwegig, die Objektorientierung als einen besonderen Denkstil zu bezeichnen. Dieser lässt sich nicht allein bei der Entwicklung von Software einsetzen. Er eignet sich ebenso für die Analyse und die Gestaltung von Geschäftsprozessen oder Organisationsstrukturen in einer Unternehmung. Auf diesen Aspekt werden wir im vierten Teil des Buchs noch ausführlich eingehen.

Gerade in der Softwareentwicklung reicht jedoch der bloße Einsatz objektorientierter Sprachen nicht aus, um die Leistungspotenziale der Objektorientierung auszuschöpfen. Viele Softwareprojekte scheitern sogar an dem Versuch, herkömmliche Vorgehensmodelle des Software Engineering einfach auf objektorientierte Projekte zu übertragen.

Um erfolgreich objektorientiert zu programmieren, ist ein objektorientiertes Denken erforderlich.



Die objektorientierte Programmierung verlangt eine Anpassung des Entwicklungsprozesses und der eingesetzten Methoden an den Denkstil – und nicht umgekehrt.

Dies gilt auch und vor allem in den frühen Phasen des Entwicklungsprozesses – in der Analyse und im Entwurf.

Die *UML* (Unified Modeling Language) ist eine Modellierungssprache, die besonders diese frühen Phasen des Software-Entwicklungsprozesses unterstützt. Sie stellt eine Notation zur Verfügung, mit der Sie Ihren Problembereich strukturieren und abbilden können. Dabei umfasst die UML sämtliche Konzepte der Objektorientierung und erleichtert so auch die spätere Implementierung Ihres Modells in eine objektorientierte Programmiersprache.

Bevor wir endgültig in die Tiefen der UML einsteigen, gibt Ihnen dieses erste Kapitel einen kurzen allgemeinen Überblick über das Wesen der Objektorientierung. Es soll Ihnen die Grundlagen dieses Denkstils näher bringen. Denn fehlendes objektorientiertes Denken ist auch dem erfolgreichen Einsatz der UML nicht gerade förderlich.

Falls Sie bereits mit den Grundprinzipien der Objektorientierung vertraut sind, können Sie die folgenden Seiten überschlagen und gleich mit dem zweiten Kapitel beginnen. Wenn Sie jedoch wissen wollen, was es mit dem »objektorientierten Denken« auf sich hat, sollten Sie zunächst dieses Kapitel lesen.

Fragt man Software-Profis nach den ersten Ansätzen der Objektorientierung zur Gestaltung von Software, berichten sie häufig von der Aufbruchstimmung Mitte der achtziger Jahre, die mit der Verbreitung der Programmiersprachen Smalltalk und C++ begann. Kommt man auf die Besonderheiten und Vorteile der Objektorientierung zu sprechen, gelten Objekte, die aus Eigenschaften und Methoden bestehen und sogar zu Klassen zusammengefasst werden können, als die neuen Konstrukte. Im nächsten Atemzug schwärmen Entwickler von der Möglichkeit, Klassen in Vererbungshierarchien anzuordnen und so viel besser als mit anderen Programmiersprachen arbeiten zu können. Diese sehr enge Sicht der Objektorientierung spiegelt jedoch nur ein auf die technologische Ebene verengtes Ergebnis objektorientierter Forschung wider. Die für den gesamten Prozess der Systementwicklung, der von der Planung über die fachliche und technische Modellierung bis hin zur Programmierung reicht, revolutionären Qualitäts- und Produktivitätsfortschritte können sie jedoch nicht vollständig erklären.

Wir bahnen uns den Zugang zur Objektorientierung auf zwei verschiedenen Wegen. Zunächst betrachten wir im nächsten Abschnitt die Modellierung aus der Sicht eines Software-Architekten, der, bevor er fachliche Inhalte darstellen und in ein Programm umsetzen kann, für sich selbst, aber immer in Abstimmung mit allen anderen Beteiligten, eine gedankliche Ordnung der fachlichen Zusammenhänge schaffen muss.

Die vollständige Bedeutung der Objektorientierung für die Gestaltung von Software erschließt sich, wenn wir im darauf folgenden Abschnitt auf ihre Entstehungsgeschichte blicken. Dort lassen sich zwei eminent wichtige Impulse nachzeichnen, die gleichzeitig die Bedeutung der Objektorientierung für die Programmierung, aber auch für die Modellierung schlechthin widerspiegeln.

Zum Einstieg ein »Bild«

Versetzen Sie sich in die Lage eines Bankangestellten. Greifen wir die Einrichtung eines Girokontos bei der Bank als Beispiel heraus. Dieses Bild verdeutlicht zunächst das grundlegende Funktionsprinzip, auf dem der objektorientierte Ansatz beruht.

Um ein grundlegendes Verständnis für den objektorientierten Ansatz zu schaffen, betrachten wir mit Hilfe der Bank eine Situation menschlicher Kommunikation und menschlichen Verhaltens, die einen Analogieschluss auf Strukturen objektorientierter Softwaresysteme ermöglicht.

Diese anthropomorphe Perspektive eines Systems kooperierender menschlicher Handlungsträger ermöglicht dann einen intuitiven Zugang zum Verständnis objektorientierter Softwaresysteme, da Sie die Struktur dieses bekannten Falls auf die allgemeine Struktur objektorientierter Systeme übertragen können.

Objekte als kooperierende Handlungsträger

Kooperierende Handlungsträger sind in dem hier dargestellten Beispiel ein Bankkunde und ein Bankangestellter. Kunde und Angestellter kooperieren bei der Einrichtung eines Girokontos bei der Bank und dem Einzahlen von 100 Euro auf das eingerichtete Konto.

Im Kontext des Bankbetriebs sind bestimmte Eigenschaften des Kunden und des Angestellten für die Abwicklung der Kooperation notwendig. Der Bankangestellte, der zuständig für Einrichtung und Abwicklung von Girokonten ist, muss die entsprechenden Arbeitsabläufe zum Einrichten und Einzahlen von Girokonten beherrschen. Der Kunde hält zum Einrichten eines Kontos Angaben zur Person, zum Wohnort sowie zu seinen Einkommensverhältnissen bereit.

Zur Einrichtung des Kontos kommunizieren Kunde und Angestellter. Der Kunde spricht den Angestellten mit der Aufforderung an, ein Girokonto einzurichten. Dabei weiß der Kunde nicht, wie die Arbeitsabläufe zur Einrichtung des Kontos aussehen. Es genügt ihm zu wissen, welcher Angestellte diesen Auftrag ausführen kann und wie er ihn ansprechen kann. Der Angestellte wiederum erfragt im Zuge der Kontoeinrichtung die Personalien und sonstigen Angaben des Kunden, wobei es für ihn nicht relevant ist, ob der Kunde diese Angaben vom Personalausweis oder sonstigen Unterlagen abliest oder sie aus dem Gedächtnis angibt.

Deutlich zeigt sich, dass die Kooperationsträger sich gegenseitig beauftragen und ein Ergebnis des Auftrags als Antwort erwarten.

Die Art und Weise der Durchführung dieser Aufträge obliegt dabei jeweils dem angesprochenen Handlungsträger. Die Kooperation wird in Form von Dienstleistungen durchgeführt, die die Handlungsträger wechselseitig nachfragen. Jeder Handlungsträger weiß vom anderen, welche Dienstleistungen er anbietet und nimmt diese über die Erteilung von Aufträgen in Anspruch.

Somit ist von jedem Handlungsträger nach außen nur bekannt, welche Dienstleistungen er anbietet. Auf welche Art diese Dienste erbracht werden und welche Informationen dazu notwendig sind, muss nur dem Handlungsträger selbst bewusst sein.

Analogie von Objekten und Kooperationspartnern

Übertragen auf die Objektorientierung werden im Folgenden die Handlungsträger als Objekte bezeichnet, die möglichen Dienstleistungen der Handlungsträger (Objekte) als Verhalten und die in Anspruch genommenen Dienstleistungen als Auftrag an ein Objekt zur Aktivierung eines Verhaltens. Die Verborgenheit der inneren Abläufe und des Zustands wird als Geheimnisprinzip bezeichnet.

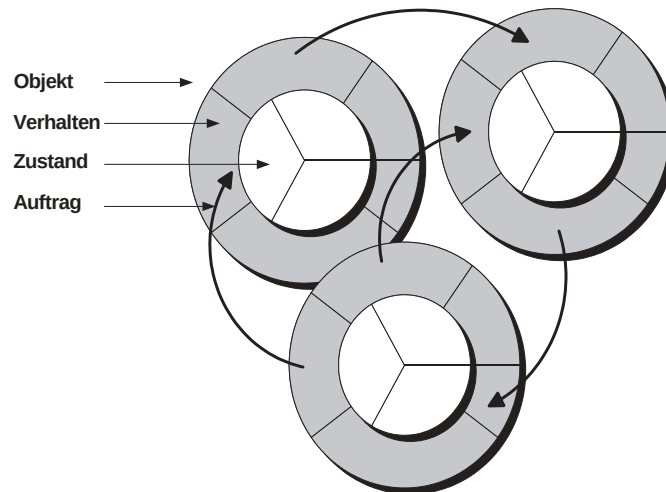


Abbildung 1.1: System kooperierender Objekte

Die Abbildung fasst die Ausführungen noch einmal zusammen und zeigt das Zusammenspiel von Objekten durch die Vergabe von Aufträgen schematisch. Ein wichtiges Prinzip ist dabei, dass die Attribute und die Arbeitsweise der Operationen eines Objekts anderen Objekten verborgen bleiben. Nach diesem Geheimnisprinzip ist fremden Objekten nur die Aufgabe, die sie erfüllen, sowie die Art, in der Methoden aktiviert werden können, bekannt.

Ein Objekt, das ein anderes Objekt beauftragt, aktiviert ein Verhalten, das in einem fremden Objekt gegeben ist. Das Verhalten dieses fremden Objekts wiederum nutzt seine eigenen Zustandswerte sowie eigene und fremde Verhaltensweisen zur Durchführung des Auftrags. So ist es durchaus üblich, Ketten von Verhaltensaktivierungen zur Auftragsdurchführung vorzunehmen.

Mit dem inneren Aufbau von Objekten und dem Prinzip der Kommunikation zwischen Objekten durch Aufträge ist nun ein Grund-

stein für das Verständnis objektorientierter Systeme gelegt. Wie das Prinzip zusammen mit weiteren Konzepten in Software-Systeme integriert wurde, zeigen die folgenden Abschnitte. Die Begriffe werden in den folgenden Abschnitten aufgegriffen und ausführlich erläutert.

Denken und Ordnung

Nachdem der vorhergehende Abschnitt einen Einblick in die Funktionsprinzipien der Objektorientierung gegeben hat, betrachten wir hier grundlegende Aspekte des menschlichen Denkens, das Grundlage jeder Modellierung ist.

Eine der größten Gefahren bei dem Einsatz objektorientierter Technologien ist deren falsche Handhabung. Nicht selten gebrauchen erfahrene C-Programmierer, die mit herkömmlichen Programmiersprachen erfolgreich ablaforientiert gearbeitet haben, eine objektorientierte Sprache wie C++, Java oder C# nicht wirklich objektorientiert. Betrachten wir dazu einige Beispiele aus anderen Bereichen.

Blättern Sie ein multimediales Lexikon Seite für Seite durch, um Informationen zu finden, oder nutzen Sie die komfortablen Hyper-text- und Suchfunktionen? Schalten Sie bei einem Auto, das ein synchronisiertes Getriebe besitzt, immer noch, wie es früher üblich war, indem Sie »Zwischengas« geben? Beheizen Sie Ihren elektrischen Backofen noch immer mit Holzscheiten? Sie können sich diese Fragen sehr leicht selbst beantworten.

Die Umstellung auf neue Technologien fällt häufig denjenigen Menschen am schwersten, die intensiv mit der Vorlängertechnologie gearbeitet haben. Fragen Sie einen LKW-Fahrer, der seit etlichen Jahren Lastwagen mit unsynchronisiertem Getriebe gefahren ist, so sagt er, dass ihm die Umstellung nicht so leicht fällt. Der Lehrer, der lange mit dem gebundenen Brockhaus-Lexikon gearbeitet hat, schmökert am PC vielleicht immer noch nach seinen ge-

wohnten Suchstrategien. Spätestens bei der Feuerung eines elektrischen Herdes mit Holzscheiten schütteln Sie den Kopf über den erfahrenen Koch. Der Backofen ist im übertragenen Sinn ein gutes Beispiel dafür, wie viel Effizienz ein Programmierer verschenkt, wenn er objektorientierte Sprachen wie C++, Java und C# und Darstellungsmittel wie die UML ablauforientiert gebraucht. Auch hier ist es so, dass gerade die erfahrenen Programmierer, die die Generation der ablauforientierten Sprachen wie C tagtäglich genutzt haben, mit der Umstellung häufig die größten Schwierigkeiten haben.

Wie können wir ein solches Verharren in gewohnten Gebrauchsmustern erklären? Es liegt nicht daran, dass es keine Gebrauchsanleitung für das Multimedialexikon, das Auto und den Backofen gibt. Ähnlich liegt das Problem bei den objektorientierten Sprachen, für die es ebenfalls einschlägige Gebrauchsanleitungen gibt. Das Problem liegt in den Denkgewohnheiten. Genau hier müssen wir ansetzen, um das Gebrauchsverhalten an die jeweilige Technologie anzupassen. Die Denkgewohnheiten müssen neu geordnet werden, um die neue Technologie effizient zu nutzen.

Neben der Forderung, dass die Nutzer von neuen Technologien sich den neuen Möglichkeiten der jeweiligen Technologie anpassen sollten, gilt umgekehrt für die Erfinder neuer Technologien, dass sie einen möglichst intuitiven Gebrauch ermöglichen sollten. So wird der Hersteller eines Elektroherds die Schalter und Knöpfe deutlich sichtbar und gut erreichbar anordnen. Woran aber müssen Erfinder von Programmiersprachen und -methoden denken?

Die Gestaltung eines Computerprogramms vollzieht sich in zwei grundlegenden, voneinander deutlich zu unterscheidenden Schritten. Zunächst muss die Aufgabe, die mit dem Computerprogramm unterstützt werden soll, aus fachlicher Sicht strukturiert werden. Die beteiligten Fachexperten, die häufig auch spätere Nutzer des Systems sind, und die Software-Experten müssen jeder für sich eine individuelle gedankliche Ordnung der Fachinhalte schaffen. Dazu kommunizieren sie untereinander so lange, bis sich eine mehrheitlich akzeptierte Sichtweise durchsetzt. Diese Sichtweise wird in

Form von Diagrammen und Texten festgehalten. In einem zweiten Schritt wird dann das erstellte Modell verfeinert und um technische Aspekte angereichert, bevor es mit einer Programmiersprache in ein lauffähiges Programm überführt wird.

Eine geeignete Unterstützung liegt also darin, dass die gedankliche Strukturierung, das Aushandeln von gemeinsam akzeptierten Sichten und Aspekte der technischen Konstruktion unter einen Hut gebracht werden müssen. Dabei zeigt die Erfahrung, dass es immer ein Drahtseilakt ist, Denken und Konstruieren aufeinander abzustimmen.



Die beiden elementaren Schritte und deren Abstimmung lassen sich als die drei großen *K* (Kognition, Konstruktion, Kommunikation) der Systemgestaltung zusammenfassen. Die *Kognition* (individuelle gedankliche Strukturierung) wird durch die *Konstruktion* (Festlegung der Inhalte nach technischen Kriterien) ergänzt und zwischen den Beteiligten mittels *Kommunikation* abgestimmt.

Da vor jedem Konstruktionsschritt die gedankliche Vorbereitung als Kognitionsschritt erfolgt, sollten Programmiersprachen und -methoden so ausgerichtet sein, dass der Übergang vom gedanklichen zum technischen Modell sich möglichst reibungslos vollzieht.

Denken und Modellieren

Stellen Sie sich vor, sie sind verantwortlich für die Entwicklung eines computergestützten Systems für eine Arztpraxis, das sämtliche Patienteninformationen, Diagnosen, Verschreibungen enthalten, aber auch den Informationsfluss vom Empfang ins Arztzimmer unterstützen soll.

Als Software-Entwickler können Sie sich auf den Standpunkt stellen, dass sie doch kein Arzt und keine Arzthelferin sind und sich damit nicht auskennen. Als verantwortlicher Software-Entwickler

müssen Sie jedoch dennoch die Zusammenhänge verstehen. Sie analysieren Dokumente, Abrechnungen, Rezepte, sprechen mit dem Arzt, den Arzthelferinnen und beobachten vielleicht einen Tag lang das Geschehen in der Arztpraxis. Nach und nach entsteht in Ihrem Kopf ein Bild von den Zusammenhängen und Abläufen.

Dabei spielt die Kommunikation mit Ärzten und Arzthelferinnen eine entscheidende Rolle. Arzt und Arzthelfer haben ebenso eine individuelle Sicht der Zusammenhänge und Abläufe. Dabei ist es ein immer wieder zu beobachtendes Phänomen, dass jeder Beteiligte eine andere Sicht der Dinge hat.

Das liegt zum einen an den unterschiedlichen Zielen, die die einzelnen Beteiligten verfolgen. Nehmen wir das Rezept. Für den Arzt ist es wichtig, Wirkstoff und Dosierung dem Patienten mitzuteilen; die Arzthelferin interessiert vornehmlich die Abrechnungsposition, die sie der Krankenkasse gegenüber mitteilen muss, der Patient möchte wissen, unter welchem Namen er das Medikament bekommt und wo die nächste Apotheke zu finden ist. Der Software-Entwickler überlegt vielleicht schon, ob er die Informationen in einer relationalen oder objektorientierten Datenbank hinterlegt. Er interessiert sich häufig weniger für die Inhalte als vielmehr für deren technische Verankerung in der zu erstellenden Software.

Aber selbst zwei Arzthelfer, die dieselben Aufgaben in einer Arztpraxis besitzen, schildern die Zusammenhänge und Abläufe sehr unterschiedlich. Liegt es daran, dass der eine nicht die Wahrheit sagt? Es ist wohl eher so, dass beide auf ihre Art Recht haben.

Jeder Mensch schafft sich mittels Wahrnehmung ein Bild seiner Umwelt und seiner selbst, indem er eine höchst individuelle gedankliche Ordnung in seiner Vorstellung schafft. Und diese Ordnung ist nicht natürlich sondern künstlich erdacht.



Die Sicht von fachlichen und technischen Zusammenhängen, die der Software-Entwickler im Laufe eines Software-Gestaltungsprozesses aufbaut, ist eine künstlich geschaffene Ordnung.

Modelle sind also immer sowohl von der subjektiven Wahrnehmung als auch von den individuellen Zielen der Beteiligten geprägt. Es gibt keine objektiven Modelle.

Gegenstände und ihre Merkmale

Entscheidend ist die Frage, wie eine gedankliche Ordnung entsteht und ob man durch geeignete Maßnahmen Einfluss auf die Struktur der entstehenden künstlich geschaffenen Ordnung ausüben kann.

Betreten Sie als Software-Entwickler zum ersten Mal die Arztpraxis, so erscheinen Ihnen die Zusammenhänge als Flut von scheinbar nicht zusammenhängenden Eindrücken, mit der Sie zunächst einmal nichts anfangen können.

In einem Lernprozess verstehen Sie nach und nach die Zusammenhänge. Sie trennen wichtige von unwichtigen Wahrnehmungen. Zunächst isolieren Sie Gegenstände, die Sie als Einheit wahrnehmen. Das kann das Rezept von Herrn Meier, der Patient Herr Schulz oder die Abrechnung mit der Krankenkasse für den Monat April sein. Sie identifizieren und benennen die Gegenstände. Da unser Gehirn assoziativ arbeitet, versuchen wir neu identifizierte Gegenstände bereits bekannten, ähnlichen Gegenständen zuzuordnen. So mag es sein, dass wir Ähnlichkeiten zwischen den Patientendaten und den Kundendaten, die wir in einer Bank in einem Projekt analysiert haben, feststellen. Darüber hinaus schaffen wir Assoziationen zwischen den neu identifizierten Gegenständen.

Wie aber entstehen solche Einheiten, die wir als Gegenstände wahrnehmen? Bei materiellen Gegenständen, wie einem Wecker einem Menschen oder einem Stuhl, ist es der materielle Zusammenhalt. Verschieben wir den Stuhl, so gehört alles, was sich mit bewegt, zum Stuhl. Er bildet damit eine operierende Einheit, die auf eine Manipulation (Verschieben) gemeinsam reagiert. Sie verhält sich anders als die Umwelt, sie hebt sich damit von ihr ab.

Diese bei materiellen Gegenständen offensichtliche Einheit finden wir ebenfalls bei immateriellen Gegenständen. Auch diese identifi-

zieren wir als gemeinsam operierende Einheit. Als Quartalsabrechnung gelten alle Informationen, die zum Ende eines Quartals an die Krankenkassen übermittelt werden.

Beschreiben wir die Gegenstände näher, neigen wir dazu, sie zu verstehen, als wären sie menschlich. Wir sehen einen Menschen als operierende Einheit, die auf Umweltreize reagiert. Gehe ich auf einen Menschen zu und grüße ihn, so wird er diesen Gruß erwidern. Dieses Prinzip übertragen wir unbewusst auf viele identifizierte Gegenstände: Die Aussagen »der Computer stürzt ab«, »der Fernseher läuft« oder »der Wind fegt um die Ecke« verstehen wir, weil wir ein menschliches Verhalten auf beliebige Gegenstände übertragen. Selbst jemand, der nichts von Computern versteht, weiß, dass mit dem Absturz der Rechner nicht mehr funktionsbereit ist.

Eine solche operierende Einheit unterliegt Veränderungen, ohne ihre Identität zu verlieren. Diese Veränderungen lassen sich in Merkmalen beschreiben. So hat der Stuhl neben seinen vier Beinen auch einen Standort, der sich verändern kann.

Gegenstände und Kategorien

Mit einer Vielzahl von Gegenständen, die isoliert nebeneinander stehen, kommen wir Menschen in unserem Denken nur sehr schlecht klar. Das hängt mit der von Miller zum ersten Mal untersuchten Kanalkapazität des menschlichen Denkens zusammen. Sie liegt im Mittel bei sieben. Das bedeutet, dass die meisten Menschen ungefähr sieben Gegenstände gleichzeitig erfassen und verarbeiten können. Auf diese Erkenntnis geht übrigens auch die berühmte Siebener-Regel für die Gestaltung von Präsentationen zurück, die besagt, dass eine gute Präsentation nicht mehr als sieben Hauptpunkte besitzen sollte, jeder Hauptpunkt maximal durch sieben Folien unterstützt werden sollte und jede Folie nicht mehr als sieben Aufzählungspunkte besitzen sollte.

Was machen wir als Systementwickler nun, wenn wir in der Arztpraxis mehrere hundert wichtige Gegenstände bei der Erstellung

berücksichtigen müssen? Zunächst versuchen wir, sie nach übereinstimmenden Merkmalen in Gruppen zusammenzufassen. Wir bilden gedankliche Schubladen für Patienten, Rezepte, Abrechnungen, Medikamente.

Kategorien und Taxonomien

Auch die gefundenen Kategorien können uns gehörige Probleme bereiten, wenn sie zu umfangreich werden. Patienten, Arzthelfer, Arzt, Pharmavertreter lassen sich auch wieder zusammenfassen zu einer Kategorie Personen, die mit der Arztpraxis etwas zu tun haben. Auch hier spielen Ähnlichkeiten wieder eine zentrale Rolle. Ähnliche Kategorien werden zueinander in Beziehung gesetzt und entlang einer Spezialisierungshierarchie angeordnet. So können die Personen in Praxispersonal und externe Personen unterteilt werden. Das Praxispersonal unterscheiden wir wiederum in Arzt und Helfer. So schaffen wir eine Ordnung, die Gegenstände zu Kategorien und Kategorien in Kategorienhierarchien zusammenfasst.

Im übertragenen Sinn ordnen wir Gegenstände in Schubladen, die in Schränken stehen. Entscheidend ist jedoch, dass diese Ordnung eine künstlich geschaffene ist.

Die frühen Ansätze

Aus Vergangenem lernen für die Zukunft. Was sich vernünftig anhört, finden wir dennoch nicht in der Entwicklung objektorientierter und anderer Ansätze wieder. Betrachten wir die Entwicklung der Objektorientierung, so finden wir bereits in den fünfziger und sebziger Jahren des letzten Jahrhunderts Ansätze, die gerade die Bedeutung der Objektorientierung für die kognitiven Aspekte der Modellierung betonen. Sie wurden zu großen Teilen in den achtziger und neunziger Jahren ignoriert, um jetzt wieder an Bedeutung zu gewinnen.

Die Wurzeln der Objektorientierung liegen jetzt fast 50 Jahre zurück. Dabei sind einige elementare Erkenntnisse der Pioniere auf diesem Gebiet durch eine Verengung auf konstruktionstechnische Überlegungen in der Diskussion verloren gegangen.

Die Geschichte der Informatik beschreitet einen Weg von der Maschinenzentrik zur Anthropozentrik. Das bedeutet, dass der Mensch als Systementwickler und Anwender Maßstab aller Aktivitäten ist und nicht die Technik alle Vorgaben allein bestimmt, wie es lange Zeit der Fall war.

Die Objektorientierung in ihrer heutigen Form ist ein alter Hut. Die Entstehungsgeschichte zeigt, dass sie über 50 Jahre alt ist. Dabei stehen zwei historische Ereignisse für die drei großen K der Systemgestaltung.

Simulationen mit Simula – die Evolution

Die Konstruktion eines Software-Systems stand im Mittelpunkt bei der Entwicklung der Programmiersprache *Simula*. Interessant für das Verständnis der Objektorientierung ist dort die Beschreibung der Anwendungsfälle, die mit objektorientierten Sprachen besonders gut umgesetzt werden können.

Die Entstehungsgeschichte

Die Vorgeschichte dieser Entwicklung begann in den späten vierziger und frühen fünfziger Jahren des letzten Jahrhunderts. *Nygaard* und *Dahl* beschäftigten sich mit Simulationsrechnungen für den ersten norwegischen Nuklearreaktor. Die dort zu bearbeitenden Probleme ließen sich jedoch nur unzureichend durch prozedurale Programmiersprachen abbilden. Typisch für derartige Simulationsprobleme sind nicht-deterministische Abläufe, die durch viele externe Ereignisse beeinflusst werden.

Später entwickelte Nygaard eine solchen Problemen angemessene Programmiersprache. Ansatzpunkt war dabei nicht wie bei proze-

duralen Sprachen die Abbildung weitgehend festgelegter Abläufe, sondern die Beschreibung von (Objekt-)Zuständen, die durch Operationen, aktiviert durch externe Ereignisse (Nachrichten), geändert werden können.

Sein Leitmotiv lautete: »Programmieren heißt zu verstehen.« Der Stand des Programms spiegelt den Stand des Verständnisses wider. So entstand mit der Programmiersprache *Simula 1967* die erste Programmiersprache, die wesentliche Konzepte der Objektorientierung Klassen, Objekte und Vererbung unterstützte.



Der Simula-Ansatz zeigt, dass aus der Notwendigkeit heraus, komplexe ereignisabhängige Zusammenhänge darzustellen, eine neue Technologie geschaffen wurde. Die Konstruktionseffizienz stand in diesem Ansatz eindeutig im Vordergrund.

Konzeptevolution in Universalsprachen

Um das Neue an objektorientierten Programmiersprachen zu verstehen, ordnen wir die wichtigsten Konzepte von Programmiersprachen.

Programmiersprachen lassen sich grundsätzlich nach verschiedenen Kriterien klassifizieren. So unterscheiden sich Sprachen nach der Sprachgeneration, dem vorherrschenden Anwendungsgebiet oder nach der Art der Beschreibung des Lösungsalgorithmus. Die Beschreibung des Lösungsalgorithmus wird hier durch den Begriff der Repräsentationsform ersetzt, da der Begriff Algorithmus eine funktionale Sicht impliziert, die in der objektorientierten Sicht nicht im Vordergrund steht (siehe Abbildung 1.2).

Die Abbildung verdeutlicht, dass die Objektorientierung in einer Entwicklungslinie als vorerst letzte evolutionäre Entwicklungsstufe einer logischen und zugleich auch historischen Abfolge von immer weiter verbesserten Konzepten in Programmiersprachen steht.

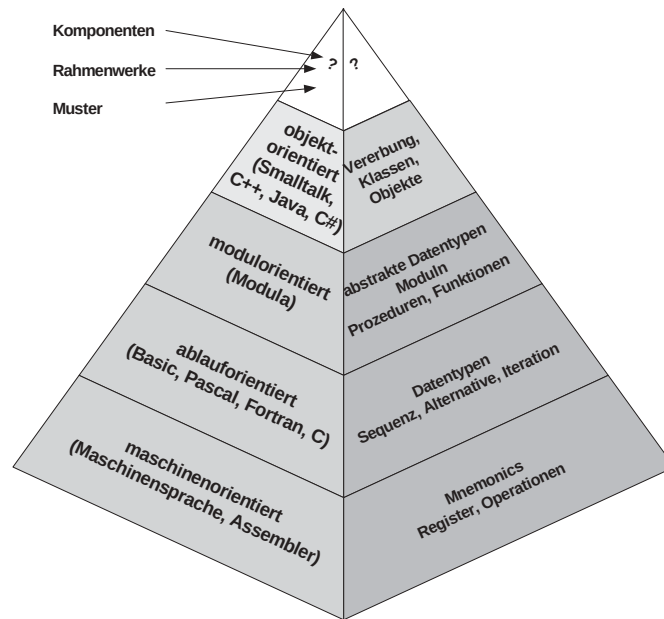


Abbildung 1.2: Sprachenpyramide

Die Abbildung beschreibt den Weg von der prozessorabhängigen hardwarenahen Darstellung bis hin zur objektorientierten Lösungsbeschreibung, die sich an Strukturen menschlicher Kognition und Kommunikation orientiert.

Der Schritt von ablauforientierten über modulare Sprachkonzepte zu objektorientierten Konzepten wird im Folgenden genauer beschrieben. Der ablauforientierte, primär prozedurale Ansatz geht wohl darauf zurück, dass die *Von-Neumann*-Architektur eine Trennung von Arbeitsspeicher und Prozessor beinhaltet. Der Prozessor in so konzipierten Rechnern operiert mittels eines Programms auf diesem Speicher.

Der Speicherbegriff wird in ablauforientierten Programmiersprachen durch den Begriff der Variablen ersetzt und das Programm in Form zeitlicher Abläufe von Lösungsschritten formuliert.

Die erste wichtige Abstraktionsstufe, die weg von prozessorabhängigen Programmen führte, waren Programmiersprachen, die Datentypen verwenden sowie die Prinzipien der strukturierten Programmierung beinhalten. Eine Trennung von Daten und Lösungsverfahren war gemäß der Orientierung an der Von-Neumann-Architektur immer noch vorhanden.

Eine Abkehr von diesem Prinzip bildet die nächste Stufe der modulatorientierten Programmiersprachen und die Erweiterung um abstrakte Datentypen. Abstrakte Datentypen legen den Schwerpunkt auf den Verhaltensaspekt. Sie definieren einen gültigen Wertebereich sowie die auf diesen Bereich zulässigen Operationen. Innerhalb eines Programms können Exemplare des abstrakten Datentyps erzeugt und benutzt werden. An dieser Stelle ist das Prinzip der Trennung von Daten und Funktionen aufgehoben. Ein abstrakter Datentyp erscheint nach außen als Einheit, die bestimmte Operationen ausführen kann.

Ein Modul ist, ähnlich einem abstrakten Datentyp, eine zusammengehörige Menge von Prozeduren und Daten. Jedes Modul ist unabhängig von einem bestimmten Programmkontext. Im Gegensatz zu abstrakten Datentypen dient ein Modul nicht als Muster für eine Menge gleichartiger Exemplare, sondern besitzt eine definierte Schnittstelle nach außen, während seine inneren Abläufe verborgen bleiben.

Zusammenfassend bleibt festzustellen, dass die Aufhebung der Trennung von Daten und Ablauf eines Programms einerseits aus datenorientierter Sicht in der Erweiterung des Datentyps zu abstrakten Datentypen geschieht. Andererseits wird die Aufhebung durch Erweiterung des Prozedurkonzeptes auch beim Modulkonzept erreicht. Beide Ansätze verbergen innere Abläufe vor dem Benutzer und bieten die Möglichkeit, Probleme abstrakt zu formulieren.

Die Objektorientierung vereinigt die Ideen beider Ansätze, indem sie das dem abstrakten Datentyp ähnliche Klassenkonzept als Zusammenfassung gleichartiger Objekte anbietet und Implementierungsdetails nach außen verbirgt. Darüber hinaus stellt das Klassenkonzept die Möglichkeit zur Modulbildung zur Verfügung, da eine Klasse wie ein Modul zur Lösung aller mit ihr verbundenen Teilprobleme verwendet wird.

Begrifflich tritt im objektorientierten Klassenkonzept die Beschreibung von Zustand und Verhalten von Objekten mit gleichen Eigenschaften an die Stelle der Definition von Wertebereich und Operationen für Daten gleichen Datentyps.

Diese Ausführungen zeigen, dass aus der Sicht der Konstruktionsprinzipien von Programmiersprachen nur kleine Schritte von abstrakten Datentypen und Modulen hin zu Objekten und Klassen führen - eher eine geringfügige Evolution denn eine bahnbrechende Revolution.

Dennoch wird Objektorientierung allgemein als andersartig und revolutionär geschildert. Das Revolutionäre und die Essenz der Objektorientierung erschließt sich eben nicht allein aus den Konstruktionsprinzipien, sondern in ihren grundlegenden Ansätzen und Annahmen, die hinter den Konstruktionsprinzipien stehen. Diese Aspekte greift der folgende Abschnitt auf.

Kommunikation mit Smalltalk – die Revolution

Im Gegensatz zum Simula-Ansatz ist der Smalltalk-Ansatz zum einen in seinen Sprachkonstrukten konsequenter objektorientiert und beruht zum anderen auf einem weiter gefassten Leitbild, das die Aufgaben eines Computers sowie das Verhältnis zwischen Computer und Benutzer explizit formuliert.

Der grundlegende Ansatz der Sprache Smalltalk geht auf Thesen von *Alan Kay* über reaktive Systeme zurück. In seiner »personal computing vision« beschreibt er ein so genanntes Dynabook als »hand-held, high-performance computer with a high-resolution

display, input and output devices supporting visual and audio communication paths, and network connections to shared information resources«. Am *Xerox Palo Alto Research Center (PARC)* beschäftigte sich seit 1970 die *Learning Research Group (LRG)*, die 1980 in *Software Concepts Group (SCG)* umbenannt wurde, zunächst mit dem Ziel, den Software-Part der Dynabook Vision von Kay umzusetzen. Dazu sollte das Software-System Smalltalk entwickelt werden.

Die Designprinzipien, die hinter der Programmiersprache Smalltalk stehen, werden im Folgenden ausführlicher erläutert, da sie verdeutlichen auf welche Weise sich der objektorientierte Ansatz sowohl in seiner Konzeption als Programmiersprache, aber insbesondere auch im Hinblick auf seine Fundierung von anderen Ansätzen zur Programmierung unterscheidet.

Menschliches Lernverhalten

Ausgangspunkt für die Forschungen war die Überlegung, menschliches Lernverhalten von Kindern auf den menschlichen Umgang mit dem Computer zu übertragen. Adele Goldberg hat diesen Ansatz 1981 beschrieben.

Grundlegende Forschungsziele der Learning Research Group waren zum einen die Entwicklung einer Programmiersprache, die als Bindeglied zwischen Modellen des menschlichen Geistes und den Schaltkreisen des Rechners dienen kann. Zum anderen sollte eine Schnittstelle zwischen Rechner und Benutzer entwickelt werden, die so weit wie möglich menschlichen Kommunikationsformen entspricht.

Ausgangspunkt für die Konzeption der Programmiersprache Smalltalk sind zwei Kerngedanken. Das Verhältnis von Mensch und Computer sowie deren Verhältnis zu einer kreativen Tätigkeit sollten untersucht werden. Im Zentrum der Betrachtung stand bei den Smalltalk-Entwicklern folgerichtig der Mensch, der bei seiner geistig-kreativen Arbeit von einem Computer in geeigneter Form unterstützt werden soll. Somit kommt dem Computer eindeutig eine Werkzeugfunktion zu.

»If a system is to serve a creative spirit, it must be entirely comprehensible to a single individual.«¹

Daraus folgt, dass dieses Werkzeug für den Benutzer im Hinblick auf die durchzuführende Tätigkeit so anzupassen ist, dass eine bestmögliche Unterstützung gewährleistet ist. Aus dieser menschenzentrierten Perspektive heraus leiten die Entwickler zwei Grundgedanken für die Mensch-Computer-Interaktion (MCI) ab.

Grundlage der MCI ist nach dem eingangs beschriebenen Forschungsziel eine beschreibende (Computer-) Sprache, die als Schnittstelle zwischen den Modellen des menschlichen Geistes und dem Modell der Rechner-Hardware dient. Aufgrund des Werkzeugcharakters des Computers dient die Semantik menschlicher Kommunikation als Maßstab und soll auf den Rechner übertragen werden. Ansatzpunkt der Überlegungen war ein vereinfachtes menschliches Kommunikationsmodell, das Kommunikation auf einer expliziten und einer impliziten Ebene darstellt, wie folgende Abbildung zeigt.

Die explizite Kommunikation umfasst den Austausch von Nachrichten durch Worte, Bewegungen und Gesten. Die implizite Kommunikation umfasst Annahmen über gemeinsame Kultur und Erfahrungen der Kommunikationspartner.

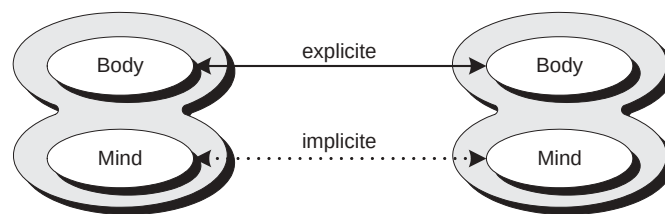


Abbildung 1.3: Smalltalk-Kommunikationsmodell²

¹ Diese Aussage geht auf Ingalls zurück.

² In Anlehnung an Ingalls.

Die Abbildung deutet die umfassenden Bezüge der Objektorientierung zur Kommunikationsforschung und Erkenntnistheorie an. Die Diskussion in der Informatik reduziert Objektorientierung häufig auf formale Aspekte von Programmiersprachen oder Analyse- und Designmethoden.

Ingalls betont in seiner Begründung der Objektorientierung drei Bezugspunkte, die konstituierend für die Entwicklung der Objektorientierung sind:

»The mechanisms of human thought and communication have been engineered for millions of years, and we should respect them as being of sound design. Moreover, since we must work with this design for the next million years, it will save time if we make our computer models compatible with the mind, rather than the other way around.«³

Er legt hier fest, wie Computer aufgebaut sein sollen. Frei nach dem Motto, so wie der Mensch von jeher funktioniert, kann das auch für Computer nicht schlecht sein. Hier zeigt sich deutlich der Nutzen und die Gefahr von Metaphern. Auf der einen Seite steht eine vorstellbare Struktur, die mit einem oder wenigen Worten beschrieben ist, und auf der anderen Seite die Gefahr, die mit dem Spruch »jeder Vergleich hinkt« zutage tritt. Eine Metapher enthält keine Begründung der Sinnhaftigkeit der von ihr erzeugten übertragenen Vorstellungen. Ein naturwissenschaftlich geprägtes Denken in Kausalbezügen führt, angewendet auf eine Metapher, nicht zu validen Ergebnissen. Später werden Leitbilder und Metaphern erneut aufgegriffen, um ihre Sinnhaftigkeit und ihren Einsatzkontext im Vergleich zu anderen Methoden zu betrachten.

Intuitiv schließt Ingalls, dass eine Übereinstimmung von menschlichem Denken und Kommunizieren mit dem Aufbau und der Funktionsweise von Computersystemen eine Voraussetzung zur Gestaltung »guter« Arbeitssysteme ist.

³ Ingalls über das Ziel der Smalltalk-Forschung.

Grundlegender Bezugspunkt ist der Mensch als Maßstab und Vorbild für die Funktionsweise und den inneren Aufbau von Computermodellen. Das menschliche Denken dient als Vorbild für die Gestaltung von Informationssystemen. Die menschliche Kommunikation ist Vorbild sowohl für die Mensch-Maschine-Interaktion als auch für die maschineninterne Kommunikation.

Die im Folgenden dargestellten Designprinzipien, die hinter der Sprache Smalltalk stehen, konkretisieren die im Bezugsrahmen aufgespannten Grundsätze. Aufgrund der seinerzeitigen Neuheit des Konzeptes und der Schaffung einer neuen Begriffswelt veröffentlichten die Mitarbeiter der Learning Research Group am Palo Alto Research Center folgende Prinzipien⁴:

- ✓ *Personal Mastery*: If a system is to serve a creative spirit, it must be entirely comprehensible to a single individual.
- ✓ *Good Design*: A system should be built with a minimum set of unchangeable parts; those parts should be as general as possible; and all parts of the system should be held in a uniform framework.
- ✓ *Purpose of Language*: To provide a framework of communication.
- ✓ *Scope*: The design of a language for using computers must deal with internal models, external media, and the interaction between these in both the human and the computer.
- ✓ *Objects*: A computer language should support the concept of »object« and provide a uniform means for referring to the objects in its universe.
- ✓ *Storage Management*: To be truly »object-oriented«, a computer system must provide automatic storage management.

⁴ Die im Folgenden dargestellten Designprinzipien von Smalltalk stammen von Ingalls, einem der Entwickler der Sprache Smalltalk.

- ✓ *Messages*: Computing should be viewed as an intrinsic capability of objects that can be uniformly invoked by sending messages.
- ✓ *Uniform Metaphor*: A language should be designed around a powerful metaphor that can be uniformly applied in all areas.
- ✓ *Modularity*: No component in a complex system should depend on the internal details of any other component.
- ✓ *Classification*: A language must provide a means for classifying similar objects, and for adding new classes of objects on equal footing with the kernel classes of the system.
- ✓ *Polymorphism*: A program should specify only the behaviour of objects, not their representation.
- ✓ *Factoring*: Each independent component in a system should appear in only one place.
- ✓ *Leverage*: When a system is well factored, great leverage is available to users and implementers alike.
- ✓ *Virtual Machine*: A virtual machine specification establishes a framework for the application of technology.
- ✓ *Reactive Principle*: Every component accessible to the user should be able to present itself in a meaningful way for observation and manipulation.

Die aufgeführten, auch im Original in einer Aufzählung dargestellten Prinzipien beziehen sich auf unterschiedliche Betrachtungsebenen. Es erscheint sinnvoll, diese Begriffe zu ordnen, um aus den Ordnungsbegriffen eine Verbindung zu den eingangs erläuterten Bezugspunkten herzustellen und damit die Vielschichtigkeit der Objektorientierung zu verdeutlichen.

Personal Mastery beschreibt eine intentionale Maxime – die Herrschaft des Menschen über den Computer – an der alle weiteren Prinzipien sich ausrichten. Umgesetzt werden soll diese Herrschaft durch ein einfaches universelles Design (*Good Design*), das einer

grundlegenden Metapher (*Uniform Metaphor*) entspricht, die für alle Bereiche eines Informationssystems gilt.

Diese Metapher leitet aus der Vorstellung kommunizierender Menschen, *Purpose of Language*, einheitlich referenzierbare Elemente (*Objects*) ab, die durch Nachrichten (*Messages*) mit anderen Systemelementen in Verbindung treten.

Das *Reactive Principle* bedingt, dass in Übereinstimmung mit der *Personal Mastery* jedes für den Benutzer verfügbare Objekt auch zur Überwachung und Manipulation für den Benutzer zugänglich ist.

Das *Scope-Prinzip* erweitert die Gültigkeit der einheitlichen Objektsicht neben dem internen Modell und der Interaktion von Mensch und Maschine insbesondere auch auf externe Kommunikationsmedien – im Konzept ein früher umfassender Ansatz, den Wissenschaft und Praxis aktuell unter dem Stichwort Multimedia diskutieren.

Neben den Zielen und Funktionsprinzipien der Objektorientierung werden hier die Strukturprinzipien der Klassenbildung (*Classification*) geordnet, die sowohl gleichartige Anwendungs- und Systemobjekte zusammenfasst als auch die als *Factoring* bezeichnete Vererbung, die entlang einer Klassenhierarchie Redundanzen vermeidet, indem sie Spezialisierungen immer als Veränderungsbeschreibung, Bezug nehmend auf ein allgemeineres Konzept, vornimmt.

Klassifizierende Hierarchien scheinen Ingalls ein typisches kognitives Ordnungsinstrument zu sein. Ein häufig angeführtes Beispiel ist die Taxonomie der Tierwelt, die eine Artenhierarchie darstellt. Diesen Aspekt haben Sie einleitend ja bereits kennen gelernt.

Die Strukturprinzipien Objektbildung, Klassenbildung und Vererbung sind sowohl in Bezug auf die Struktur menschlichen Denkens als auch in Bezug auf softwareinhärente Qualitätskriterien wichtige Bausteine für hochwertige Software-Systeme.

Als rein softwaretechnisch motivierte Prinzipien sind die Modularität (*Modularity*) – eine inhärente Eigenschaft der Objektstruktur –

und die virtuelle Maschine (*Virtual Machine*), die als hardwareunabhängige Zwischenform zwischen Interpreter und Compiler die Plattformunabhängigkeit bei Geringhaltung der Performanceverluste bewirkt, zu sehen.

Entwicklung im Abriss

Die Entwicklung von Programmiersprachen war der entscheidende Motor für die Objektorientierung in der Informatik. Dieser Abschnitt zeigt die wichtigsten Programmiersprachen und die für die Entwicklung verwendeten objektorientierten Entwicklungsmethoden.

Programmiersprachen im Zeitablauf

Dieser Abschnitt enthält eine kurze Übersicht über objektorientierte Programmiersprachen und stellt deren Konzepte im Überblick dar.

Generell lassen sich Programmiersprachen, die objektorientierte Konzepte enthalten, nach verschiedenen Kriterien unterscheiden. Sie können zunächst wiederum wie Programmiersprachen nach der Sprachgeneration, dem Anwendungsgebiet oder der Beschreibung des Lösungsalgorithmus unterschieden werden.

Auch hier ist es sinnvoll, das Kriterium der Beschreibung des Lösungsalgorithmus auszuwählen. Im Fall objektorientierter Sprachen ist der Begriff Lösungsalgorithmus irreführend und sollte durch die Bezeichnung Repräsentationsform ersetzt werden, da die Algorithmik Ablaufstrukturen betont, die Objektorientierung hingegen für die Aufhebung der Fokussierung von Daten oder Ablaufstrukturen steht.

Grundsätzlich gibt es Sprachen, die eine oder mehrere Repräsentationsformen unterstützen. Beinhaltet eine Sprache mehr als eine Repräsentationsform, so wird sie als hybride Sprache bezeichnet.

Hybride Sprachen unterstützen häufig das prozedurale und das objektorientierte Paradigma.

In diesen Hybridsprachen wie C++ oder Object-Pascal wird ein Klassentyp meist als Erweiterung eines Strukturtyps abgeleitet. Methoden werden aus Prozeduren und Funktionen syntaktisch gebildet. Methodenaufrufe werden aus Prozedur- bzw. Funktionsaufrufen abgeleitet. Durch die Integration des Vererbungskonzeptes entsteht so eine hybride Sprache.

Wird innerhalb des objektorientierten Sprachparadigmas der Grad der Unterstützung objektorientierter Prinzipien unterschieden, so hat sich folgendes Klassifikationsschema durchgesetzt.

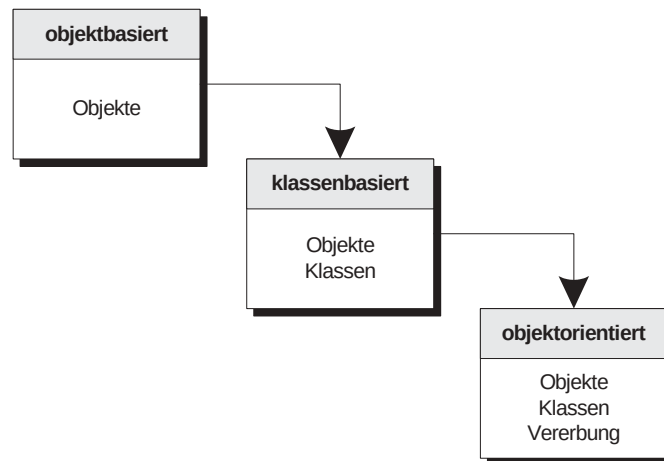


Abbildung 1.4: Klassifikationsschema objektbasierter Sprachen⁵

Den geringsten Grad an Unterstützung bieten Programmiersprachen, die lediglich Objekte als Konstrukt beinhalten. Können Ob-

⁵ Diese Unterscheidung geht auf Wegener zurück.

jekte mit gleicher Datenstruktur und gleichen Methoden durch Klassen beschrieben werden, wird eine solche Programmiersprache als klassenbasiert bezeichnet. Lassen sich die Klassen wiederum durch Vererbungsbeziehungen strukturieren, so gilt die Sprache als objektorientiert. Neben der Vererbung hat sich auch das Prinzip des Polymorphismus als anerkannt wichtige Eigenschaft etabliert.

Die folgende Abbildung zeigt die Entstehung objektorientierter Programmiersprachen. Objektorientierte Sprachen sind in der Abbildung grau hinterlegt. In der Regel sind die objektorientierten Sprachen aus nicht objektorientierten Sprachen, die weiß dargestellt sind, hervorgegangen oder haben bestimmte Eigenschaften von diesen übernommen. In der Darstellung sind zwei wesentliche Ursprünge objektorientierter Sprachen auszumachen. Zum einen ist der bereits beschriebene Entstehungsprozess der Sprachen im *Simula-Smalltalk*-Zweig zu nennen und zum anderen die aus der funktionalen Sprache *Lisp* entstandenen, der künstlichen Intelligenz zuzurechnenden Sprachen (siehe Abbildung 1.5).

Im Folgenden werden kurz wichtige Eigenheiten der vier Sprachen *Simula*, *Eiffel*, *Smalltalk*, *C++*, *Java* und *C#* erläutert.

Die Sprache *Simula 67* wurde von *Nygaard* und *Dahl* 1966 aus der Sprache *Algol* (1960) entwickelt. In *Simula* werden zustandsbeschreibende Daten und Zustandsübergänge definiert, um so komplexe, ereignisgesteuerte Simulationen abbilden zu können.

Smalltalk wurde im Jahr 1972 durch die *Learning Research Group* am *Xerox Palo Alto Research Center* zur Verbesserung der Kommunikation zwischen Mensch und Maschine entwickelt und ist in der Version *Smalltalk-80* als Universalsprache konzipiert.

Die Sprache *Eiffel* wurde von Bertrand Meyer im Jahr 1988 als Antwort auf die typungebundene Interpretersprache *Smalltalk* entworfen mit dem Ziel, Korrektheit, Portabilität, Effizienz und Robustheit zu verbessern durch Typbindung, Kompilation und Pre- und Postconditions, die die Zustandskorrektheit von Objekten gewährleisten.

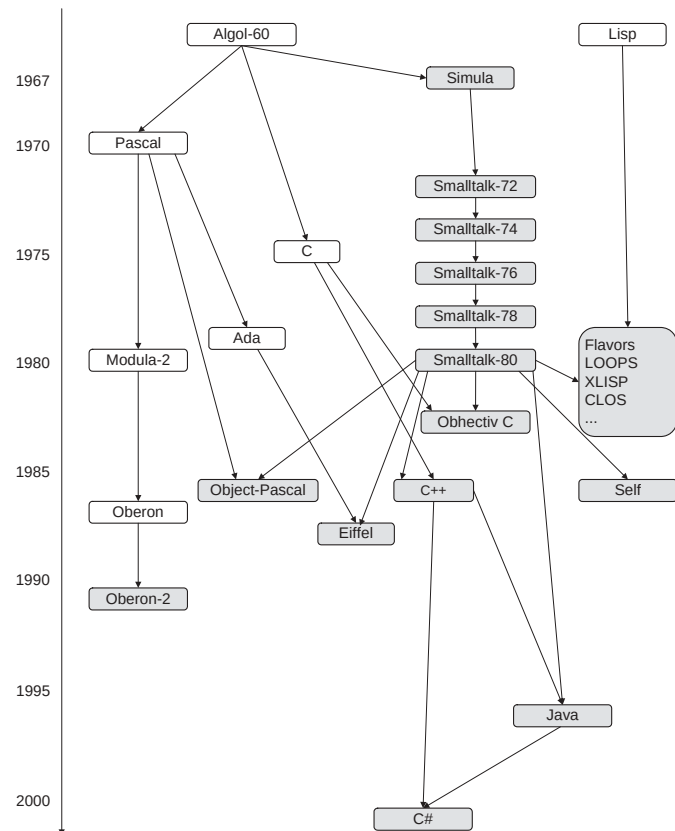


Abbildung 1.5: Programmiersprachen

Die Sprache C++ ist eine Erweiterung der weltweit verbreiteten Sprache C und bietet als Hybridsprache im objektorientierten Bereich hohe Effizienz und Objekte mit flexibel definierbaren Zugriffsrechten an. Dabei werden die beschriebenen Grundkonzepte,

wie das Geheimnisprinzip, zum Teil kontrolliert außer Kraft gesetzt.

Die Programmiersprache *Java* stellt im Wesentlichen eine Kombination der objektorientierten Syntax von *C++* mit einer auf *Smalltalk* basierenden Architektur, die Bytecode durch eine virtuelle Maschine ausführt, dar.

Neueste Entwicklung ist die Programmiersprache *C#*, die sich sehr eng an *Java* anlehnt, aber auch Elemente von *C++* besitzt.

Objektorientierte Methoden im Zeitablauf

Da objektorientierte Methoden nicht auf anderen Methoden wie zum Beispiel den strukturierten Methoden aufsetzen, soll dieser Abschnitt kurz die Entwicklung hin zur UML darstellen. Dagegen erfahren Sie im einleitenden Abschnitt zur UML mehr zu der Entwicklung der UML selbst.

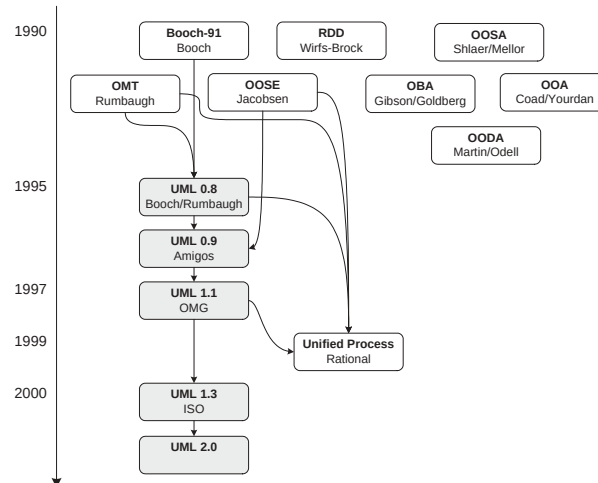


Abbildung 1.6: Vorläufer und Versionen der UML

Betrachten wir die Landschaft objektorientierter Methoden, finden wir verschiedene Ansätze. Je nach Verwendungszweck können wir sie in konstruktionsorientierte und kognitionsorientierte Methoden unterscheiden.

Aus der Anwendungssicht lassen sie sich in Universalmethoden und Methoden zur Geschäftsprozessanalyse unterscheiden.

Nach der Ausrichtung unterscheiden wir stärker verhaltensorientierte von zustandsorientierten Methoden.

Die UML ist ausgerichtet als Software-Konstruktionsmethode, die aber auch Elemente zur Geschäftsprozessanalyse enthält. Sie kann sowohl zustands- als auch verhaltensorientiert angewendet werden. Zur Versionsabfolge kommen wir im einleitenden Abschnitt zur UML zurück. Die Version 1.4 ist in der Grafik ausgelassen, da sie lediglich einige wenige Erweiterungen und Klarstellungen zur Version 1.3 enthält. Version 2.0 wird der nächste wichtige Meilenstein der UML.

Die *OBA* (Object Behavior Analysis) ist Vertreter einer streng verhaltensorientierten Sicht, mit der die Smalltalk-Erfinder viele Smalltalk-Projekte erfolgreich realisiert haben.

OOA von Coad/Yourdan legt den Schwerpunkt auf die Erstellung von Fachmodellen und ist eher zustandsorientiert.

Eine besondere Ausrichtung besitzt die Methode *RDD* (Responsibility Driven Design) von Wirffs-Brock, die eine ausdrücklich vermenschlichende Sichtweise einnimmt. Damit steht sie in guter Tradition zum Smalltalk-Ansatz. Objekte werden wie in unserem eingangs dargestellten Bank-Beispiel als Auftraggeber und Auftragnehmer, die vereinbarte Dienstleistungen anbieten und nachfragen.

Objekte

Wie die bisher dargestellten Grundprinzipien im objektorientierten Ansatz umgesetzt wurden, zeigen die folgenden Abschnitte.

Der Name »Objektorientierung« lässt uns schwer erkennen, dass sich in diesem Gestaltungskonzept alles um Objekte dreht. Sie werden diesen Begriff vermutlich selbst auch umgangssprachlich häufig verwenden.

Was ist ein Objekt?

Doch was ist eigentlich ein Objekt? Meist denkt man zunächst an Gegenstände im Allgemeinen. Jeder Gegenstand ist ein Objekt. Nach dieser Definition befinden sich jede Menge Objekte in Ihrer unmittelbaren Umgebung:

- ✓ der Stuhl, auf dem Sie sitzen,
- ✓ das Haus, in dem Sie wohnen,
- ✓ das Auto in Ihrer Garage,
- ✓ dieses Taschenbuch, das Sie gerade lesen.

Einige der genannten Objekte setzen sich wiederum aus anderen Objekten zusammen:

- ✓ Der Stuhl besteht aus einer Lehne, einer Sitzfläche und in der Regel vier Stuhlbeinen.
- ✓ Das Haus hat ein Dach, Wände, Zimmer, Türen usw.
- ✓ Das Auto setzt sich aus einem Chassis, einer Karosserie und Rädern zusammen.
- ✓ Sogar das Buch lässt sich in einen Umschlag und einzelne Seiten aufteilen, die jede für sich ein eigenes Objekt darstellen.

Im Rahmen einer solchen gegenständlichen Sichtweise können auch Menschen, Tiere und Pflanzen unter den Objektbegriff gefasst werden.

Die Definition des Objektbegriffs geht jedoch in der Objektorientierung weit über die rein gegenständliche Betrachtung hinaus.

Hier werden auch abstrakte Konstrukte als Objekte bezeichnet. Dazu zählen beispielsweise:

- ✓ Beziehungen
 - ✓ Familie
 - ✓ Arbeitsverhältnis
 - ✓ Mitgliedschaft in einem Verein
- ✓ Verträge
- ✓ Prozesse
 - ✓ Arbeitsabläufe im Büro
 - ✓ Warenlieferungen
 - ✓ Entwicklung eines Computerprogramms
 - ✓ Reifenwechsel
- ✓ Ereignisse
 - ✓ Lottogewinn
 - ✓ Einstellung eines neuen Mitarbeiters
 - ✓ Frühlingsanfang

Vermutlich werden Sie spätestens jetzt einwenden: Na, wenn man es so betrachtet, dann besteht die Welt ja nur noch aus Objekten!

Modellierung mit Objekten

Genau das ist die Philosophie, die hinter der Objektorientierung steckt: Die Welt wird als Sammlung von Objekten und ihrer Beziehungen verstanden. Allerdings geht es jeweils nur um einen kleinen Teil der Welt – nämlich um die direkte Umgebung, die Sie für Ihren ganz konkreten Problembereich als relevant ansehen. Auf den Punkt gebracht, kann der Objektbegriff vorerst so definiert werden:



Ein Objekt ist eine identifizierbare Einheit, die aus einem bestimmten Kontext isolierbar ist.

Den Kontext bestimmen Sie selbst durch die Abgrenzung Ihrer jeweiligen Problemstellung. Wenn Sie beispielsweise ein Warenwirtschaftssystem entwickeln wollen, dann gehört alles, was für die Warenwirtschaft in Ihrer Unternehmung relevant ist, zum aktuellen Kontext. Jedes Element innerhalb dieses Betrachtungsfokus, ob es sich nun um einen Gegenstand, einen Menschen oder einen abstrakten Geschäftsprozess handelt, ist ein Objekt.

Die folgende Abbildung veranschaulicht das Prinzip der Modellbildung.

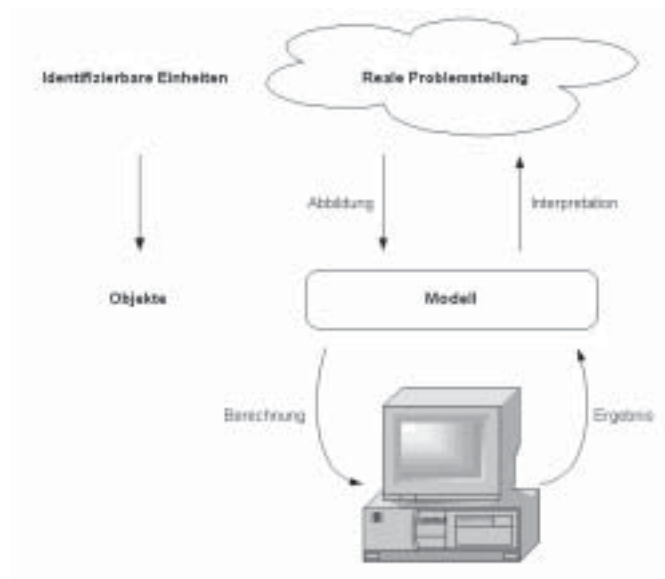


Abbildung 1.7: Modellbildung

Reale Problemstellungen, die mit Hilfe eines Programms gelöst werden sollen, sind in der Regel äußerst komplex und besitzen Wechselwirkungen mit vielen anderen Aufgabenstellungen. Es ist daher unmöglich, *sämtliche* Aspekte der Realität in der Softwareentwicklung zu berücksichtigen.

Durch Abstraktion wird deshalb die Realität in vereinfachter Form in einem Modell abgebildet. Das Modell repräsentiert nicht alle, aber dennoch die wesentlichen Aspekte, die zur Lösung der jeweiligen Aufgabenstellung eine Rolle spielen. Das Modell ist hinreichend einfach, um durch ein Computerprogramm berechnet werden zu können, und gleichzeitig genügend exakt, um eine auch in der Realität brauchbare Lösung zu gewährleisten. Die Ergebnisse der Berechnung gelten jedoch zunächst nur für die speziellen (vereinfachten) Modellannahmen und müssen anschließend noch für die reale Problemsituation interpretiert werden.

In der objektorientierten Modellierung werden sämtliche identifizierbaren Einheiten, die für die Problemlösung relevant sind, in Form von Objekten repräsentiert. Hierbei ist die persönliche Einschätzung des Modellierers (bzw. des beauftragten Teams) von entscheidender Bedeutung. Nur diejenigen Aspekte der Realität, die er für relevant hält, fließen in das Modell ein. Die Qualität des Modells – und damit auch der Berechnungsergebnisse – hängt maßgeblich von der individuellen Sichtweise des Modellierers auf die Problemstellung ab.



Beauftragen Sie drei verschiedene Teams mit ein und derselben Modellierungsaufgabe – Sie werden mindestens drei unterschiedliche Modelle erhalten.

Doch wie werden die in der Realität identifizierten Einheiten in einem Objekt repräsentiert? Die drei folgenden Abschnitte erläutern mit dem Namen, dem Zustand und dem Verhalten die drei Bestandteile eines Objekts, mit denen alle seine Eigenschaften erschöpfend beschrieben werden könne.

Name – eindeutige Identifikation von Objekten

Eine wichtige Voraussetzung für die Abbildung von Elementen der realen Problemstellung in Objekten ist ihre Identifizierbarkeit. Jedes Element muss sich klar von den übrigen Elementen abgrenzen lassen. Verwechslungen, Überschneidungen und Doppeldeutigkeiten sind zu verhindern. Daher wurde in der obigen Definition der Begriff »identifizierbare Einheit« gewählt.

Die Objektorientierung löst die Aufgabe, identifizierbare Einheiten der Realität zweifelsfrei zu bestimmen, durch die Vergabe von eindeutigen *Namen*.



Jedes Objekt ist durch seinen Namen eindeutig identifizierbar und von anderen Objekten abgrenzbar.

Haben Sie beispielsweise in Ihrer realen Problemstellung ein Element »Kunde« identifiziert, dann existiert das Objekt mit dem Namen Kunde in Ihrem Objektmodell genau ein Mal. Wollen Sie mehrere Kunden in dem Modell abbilden, dann muss jedes einzelne Kundenobjekt einen eigenen, unverwechselbaren Namen erhalten. Sie könnten beispielsweise die Kunden durchnummerieren. Die Kundenobjekte hießen dann Kunde1, Kunde2, ... KundeX. Eine andere Alternative besteht darin, die tatsächlichen Namen der Kunden auch für die Objektbezeichnungen zu verwenden: KundeMeier, KundeSchmidt usw.

Über den Namen ist jedes Objekt eindeutig identifizierbar. Damit beschreibt das Objekt aber noch keinerlei Eigenschaften des realen Elements, das es im Modell repräsentiert. Die Objektorientierung unterscheidet zwei Arten von Eigenschaften: statische und dynamische.

Zustand – statische Objekteigenschaften

Statische Eigenschaften eines Objekts werden als *Zustand* bezeichnet. Der Zustand drückt die aktuelle Situation eines Objekts aus, die mit Hilfe von *Attributen* beschrieben wird.



Der Zustand eines Objekts wird durch die aktuellen Werte seiner Attribute definiert. Er beschreibt statische Eigenschaften des Objekts.

Angenommen, Sie möchten ein Auto als Objekt identifizieren. Der Name des Objekts wäre dann beispielsweise »meinAuto«. Nun kann Ihr Auto verschiedene Zustände besitzen. Es kann an der Ampel stehen, mit 120 km/h über die Autobahn fahren, manchmal sitzen Sie allein in Ihrem Wagen und ein anderes Mal fährt Ihre gesamte Familie mit. All diese Situationen stellen unterschiedliche Zustände des Objekts *meinAuto* dar.

In diesem Beispiel lassen sich die genannten Zustände mit Hilfe von zwei Attributen definieren. Ob das Auto steht oder fährt, lässt sich mit einem Attribut *geschwindigkeit* beschreiben. Die aktuelle Geschwindigkeit des Autos wird dann durch den entsprechenden Attributwert repräsentiert. Ein zweites Attribut *anzahl_Insassen* zeigt an, wie viele Personen in dem Auto sitzen.

Die folgende Abbildung enthält eine grafische Darstellung des Objekts *meinAuto* mit den genannten Attributen.

meinAuto
geschwindigkeit = 120
anzahl_Insassen = 1

Abbildung 1.8: Das Objekt *meinAuto*

In der Abbildung beträgt die aktuelle Geschwindigkeit des Autos 120 km/h und es befindet sich ein Insasse an Bord. Wenn Sie nun langsamer fahren (z.B. 50 km/h), sitzen Sie immer noch in demselben Auto, es ändert lediglich seinen Zustand.

meinAuto
geschwindigkeit = 50
anzahl_Insassen = 1

Abbildung 1.9: Das Objekt `meinAuto` mit verändertem Zustand

Die Abbildung zeigt die objektorientierte Darstellung des veränderten Zustands. Das Attribut `geschwindigkeit` hat jetzt den Wert 50 statt 120.



Der Zustand eines Objekts bildet eine *Momentaufnahme*. Er repräsentiert durch die Attributwerte die Situation Ihrer realen Problemstellung zu einem bestimmten Zeitpunkt.

Jedes Objekt der realen Welt findet seine Entsprechung in einem solchen Objekt des Modells. Wenn Sie in Ihr Programm ein zweites Auto aufnehmen wollen, dann müssen Sie ihm einen anderen Namen geben, da jedes Objekt durch seinen Namen eindeutig identifizierbar sein muss. Definieren wir also ein zweites Auto, das ebenfalls nur einen Insassen hat und gerade 50 km/h schnell fährt:

meinAuto	deinAuto
geschwindigkeit = 50	geschwindigkeit = 50
anzahl_Insassen = 1	anzahl_Insassen = 1

Abbildung 1.10: Die Objekte `meinAuto` und `deinAuto`

Das neue Objekt heißt, wie der obigen Abbildung zu entnehmen ist, »deinAuto«. Es gibt also jetzt zwei Objekte in unserem Modell: meinAuto und deinAuto. Auch wenn beide Objekte denselben Zustand besitzen (50 km/h Geschwindigkeit und ein Insasse), so handelt es sich dennoch um zwei verschiedene Objekte, die jeweils ein eigenes Element der realen Welt repräsentieren. Welches Auto gemeint ist, kann jederzeit durch den Namen des entsprechenden Objekts eindeutig bestimmt werden.



Zwei Objekte, die ein und denselben Zustand besitzen, repräsentieren dennoch zwei verschiedene Elemente der realen Problemstellung.

Vielleicht ist Ihnen die etwas eigenartige Schreibweise der Namen und der Attribute aufgefallen. Sie entspricht der objektorientierten Konvention, die in vielen Modellierungs- und Programmiersprachen Verwendung findet. Auch für die UML und die Beispiele in diesem Buch gilt die folgende, allgemein übliche Vereinbarung:

- ✓ Objektnamen beginnen mit einem Kleinbuchstaben (z.B. auto).
- ✓ Bei zusammengesetzten Objektnamen beginnt das erste Wort mit einem Kleinbuchstaben, alle weiteren Worte stehen ohne Leerzeichen sofort dahinter und beginnen mit einem Großbuchstaben (z.B. meinAuto).
- ✓ Für Attribute gelten dieselben Regeln wie für Objektnamen (z.B. geschwindigkeit und hoechstGeschwindigkeit).
- ✓ Alternativ können zusammengesetzte Bezeichnungen zur besseren Lesbarkeit auch durch einen Unterstrich getrennt werden (z.B. anzahl_Insassen).

Verhalten – dynamische Objekteigenschaften

Um den Zustand eines Objekts zu ändern, ist die Ausführung einer Aktion erforderlich. In dem obigen Beispiel muss etwa ein zusätzli-

cher Passagier in das Auto einsteigen, damit sich die Anzahl der Insassen erhöht. Die Fähigkeit von Objekten, Aktionen durchzuführen, wird als *Verhalten* bezeichnet.

Erst durch das Verhalten, das die dynamischen Eigenschaften eines Objekts beschreibt, kommt Bewegung in Ihr Objektmodell. Die Aktionen, in denen sich das Verhalten eines Objekts äußert, werden in der Objektorientierung als *Operationen* oder auch als *Methoden* bezeichnet.



Das Verhalten eines Objekts wird durch seine Operationen (Methoden) beschrieben. Sie definieren dynamische Eigenschaften des Objekts.

Damit haben wir nun alle Bestandteile von Objekten beisammen:

- ✓ den Namen zur eindeutigen Identifikation,
- ✓ den Zustand für die Beschreibung statischer Eigenschaften
- ✓ und das Verhalten zur Repräsentation dynamischer Eigenschaften von Objekten.

Zusammengefasst können wir daraus eine abschließende Definition für den Objektbegriff ableiten.



Ein Objekt ist eine identifizierbare Einheit, die aus einem bestimmten Kontext isolierbar ist. Es wird durch seinen Namen, seinen Zustand und sein Verhalten eindeutig beschrieben.

Mit der Datenkapselung und der Kommunikation zwischen Objekten beleuchten die beiden folgenden Abschnitte zwei in diesem Zusammenhang wichtige Aspekte ein wenig näher.

Datenkapselung

Hinter dem Zusammenspiel von Attributen (Zustand) und Operationen (Verhalten) steht ein weiteres wichtiges Prinzip der Objekt-

orientierung. Alle (statischen und dynamischen) Informationen über ein Objekt werden in seiner Beschreibung zusammengefasst. Die Attribute sind grundsätzlich von außen unsichtbar. Das bedeutet insbesondere, dass Attributwerte eines Objekts nicht ohne weiteres von anderen Objekten geändert werden können. Eine Manipulation von Attributwerten ist nur über die Methoden eines Objekts möglich. Man bezeichnet diese Eigenschaft der Objektorientierung als *Datenkapselung*.



In objektorientierten Modellen bestehen Abläufe aus der Interaktionen von Objekten. Ein Prozess in einem Objektmodell setzt sich also zusammen aus einzelnen Kommunikationsvorgängen zwischen Objekten. Die Operationen dienen den Objekten dazu, Botschaften mit anderen Objekten auszutauschen.

In dem Auto-Beispiel kann die Anzahl der Insassen also nicht einfach durch die *direkte* Zuweisung eines neuen Werts an das Attribut `anzahl_Insassen` geändert werden. Nur die Operation `Einsteigen()` darf die Attribute seines Objekts manipulieren. Um die Anzahl der Insassen von einem auf zwei zu erhöhen, bleibt somit keine andere Möglichkeit, als die Operation `Einsteigen()` zu aktivieren, die dann die gewünschte Aktion durchführt und den Attributwert auf *indirektem* Wege ändert. Die folgende Abbildung veranschaulicht diesen Zusammenhang (siehe Abbildung 1.11).

Diese Vorgehensweise mag auf den ersten Blick umständlich erscheinen. Sie hat aber durchaus ihren Sinn. Eine wesentliche Voraussetzung für die Wiederverwendbarkeit von Softwarebausteinen ist nämlich deren jederzeitige Integrität. Wenn mehrere Objekte (oder Programme) gleichzeitig von außen ohne weiteres in den Zustand eines anderen Objekts eingreifen und ihn manipulieren können, dann ist nicht gewährleistet, dass jeweils mit den korrekten Attributwerten gearbeitet wird. Gibt es aber eine Operation, die allein für solche Manipulationen zuständig ist, lässt sich dieses Problem beheben.

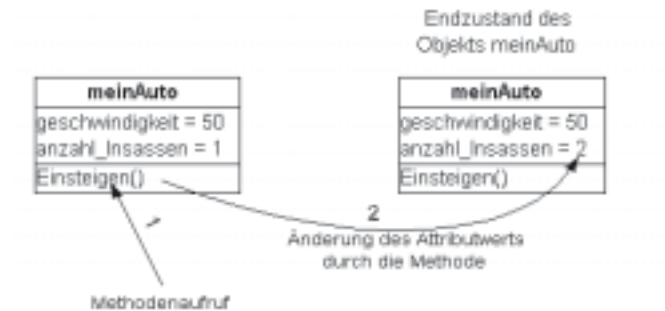


Abbildung 1.11: Manipulation von Attributwerten über Methoden

Ein weiterer wichtiger Aspekt für die *Wiederverwendung* von Objekten in verschiedenen Programmen ist die Vermeidung unklarer gegenseitiger Abhängigkeiten. Durch die Definition eindeutiger Schnittstellen ist jederzeit klar, wie und von welchen anderen Elementen auf ein Objekt zugegriffen werden kann. Einzelne Module können daher in verschiedenen Anwendungen wiederverwendet werden, ohne dass die Funktionalität dadurch beeinträchtigt wird. Darüber hinaus können einzelne Module eines Programms verändert werden, ohne dass die übrigen Objekte davon in irgendeiner Weise beeinträchtigt werden.

Klassen

Durch seinen Namen, seinen Zustand und sein Verhalten ist ein Objekt eindeutig definiert. Sie können nun jeden beliebigen Gegenstand Ihrer Umgebung als Objekt beschreiben. Am Beispiel der Autos wurde dies bereits dargestellt.

Was macht aber ein Autohersteller, der sämtliche Autos, die er produziert, in einer objektorientierten Software abbilden möchte? Es

wäre ziemlich mühsam, wenn er jedes einzelne Auto neu beschreiben müsste. Auch dafür hat die Objektorientierung mit der *Klassenbildung* eine geeignete Lösung.



Eine Klasse definiert ein Muster zur Beschreibung von Objekten mit gleichen oder ähnlichen Eigenschaften.

Konkrete Klassen

Klassen definieren Muster, denen alle zugehörigen Objekte entsprechen. Mit Hilfe des Musters, das in der Klassenbeschreibung vorgegeben ist, kann eine beliebige Anzahl von Objekten erzeugt werden, die alle dieselben Eigenschaften (Attribute und Operationen) haben. Die Objekte unterscheiden sich lediglich durch ihren Namen (eindeutige Identifizierbarkeit) und ihre konkreten Zustände (Attributwerte).

Die Beschreibung einer Klasse in einem Objektmodell entspricht weitgehend der eines Objekts. Auch eine Klasse ist über ihren Namen eindeutig identifizierbar. Im Unterschied zu Objekten beginnt der Klassenname nach objektorientierter Konvention stets mit einem Großbuchstaben. Sie beschreibt darüber hinaus den Zustandsraum ihrer Objekte über Attribute und deren Verhalten über Operationen. Die folgende Abbildung verdeutlicht diesen Zusammenhang anhand des Auto-Beispiels.

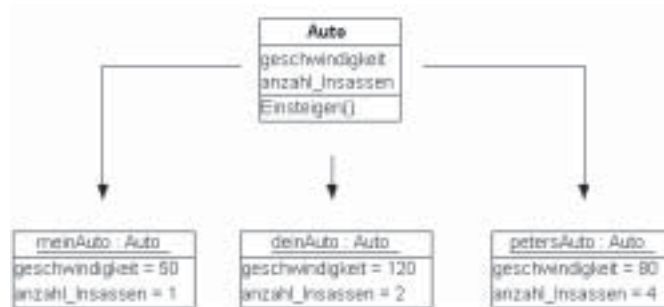
Die Klasse *Auto* definiert in der Abbildung ein Muster für alle ihre Objekte.



Die Objekte einer Klasse werden auch als ihre *Instanzen* bezeichnet.



Klassen, die in der Lage sind, Objekte nach dem in der Klassenbeschreibung definierten Muster zu erzeugen, heißen *konkrete Klassen*.

**Abbildung 1.12:** Die Klasse Auto

Dieses Muster besteht aus den beiden Attributen `geschwindigkeit` und `anzahl_insassen` sowie der Operation `Einsteigen()`. Alle Objekte, die zu der Klasse **Auto** gehören, besitzen diese Eigenschaften. Sie unterscheiden sich lediglich durch ihren eindeutigen Namen und ihren aktuellen Zustand. In dem Beispiel der Abbildung 1.12 hat die Klasse **Auto** mit den Objekten **meinAuto**, **deinAuto** und **petersAuto** drei Instanzen. Die drei Autos haben jeweils eine andere aktuelle Geschwindigkeit und damit unterschiedliche Zustände. Doch selbst wenn zwei oder mehr **Auto**-Objekte identische Zustände aufweisen, so bleiben es doch verschiedene Objekte – genau wie zwei reale Autos, die auf der Autobahn dieselbe Anzahl Insassen haben und mit derselben Geschwindigkeit unterwegs sind.

Abstrakte Klassen

Neben den konkreten Klassen, die Objekte nach dem in der Klassenbeschreibung definierten Muster erzeugen können, gibt es in der Objektorientierung auch so genannte *abstrakte Klassen*.

Diese werden für Verallgemeinerungen oder Oberbegriffe eingesetzt. Im Unterschied zu konkreten Klassen können abstrakte Klassen keine Objekte bilden. Sie sind also stets »leer«.



Abstrakte Klassen definieren lediglich Muster, aus denen die Eigenschaften speziellerer (konkreter) Klassen abgeleitet werden.

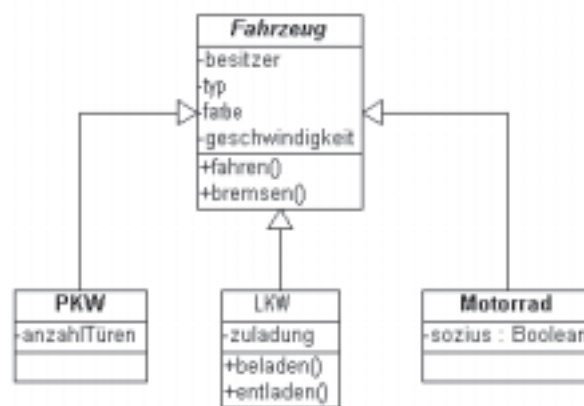


Abbildung 1.13: Abstrakte und konkrete Klassen

In der vorstehenden Abbildung ist *Fahrzeug* eine abstrakte Klasse – erkennbar an der kursiven Schrift des Klassennamens im Modell. Sie definiert ein Muster allgemeiner Eigenschaften von Fahrzeugen. Demnach besteht der Zustand eines Fahrzeugs aus dem Namen des Besitzers, dem Typ, der Farbe und der Geschwindigkeit. Das Verhalten äußert sich in den Methoden *fahren* und *bremsen*.

Aus dieser Beschreibung werden mit den Klassen *PKW*, *LKW* und *Motorrad* konkrete Arten von Fahrzeugen abgeleitet. In dem Modell der Abbildung existieren keine Objekte vom Typ »Fahrzeug«, da diese Oberklasse als abstrakte Klasse nicht in der Lage ist, Instanzen zu erzeugen. Es können allerdings sehr wohl Exemplare der Typen »PKW«, »LKW« und »Motorrad« gebildet werden. Die Klasse *PKW* erweitert die Definition von *Fahrzeug* durch das Attribut an-

zahlTüren. LKW besitzt zusätzlich das Attribut `zuladung` und die beiden Methoden `beladen` und `entladen`. Motorrad spezialisiert seine Oberklasse mit dem Attribut `sozius`.

Wie diese Spezialisierung von Klassen genau funktioniert, erläutert der folgende Abschnitt.

Vererbung

Neben der Klassenbildung ist die *Vererbung* ein weiteres wichtiges Konzept der Objektorientierung. Das Grundprinzip der Vererbung besteht darin, aus bereits bekannten und beschriebenen Elementen eines Objektmodells weitere Elemente abzuleiten.



Bei der Vererbung werden bereits existierende Klassen um bestimmte Eigenschaften (Attribute oder Methoden) erweitert und somit *spezialisiert*.

Daher bezeichnet man Vererbungsbeziehungen zwischen Klassen auch als *Generalisierung/Spezialisierung*.

Person
-anschrift
-geburtsdatum
-telefonNr
+druckeAnschrift()
+druckeTelefonNr()

Abbildung 1.14: Die Klasse Person

Im Beispiel der Abbildung wird zunächst eine Klasse mit dem Namen *Person* definiert. Jede Person besitzt eine Anschrift, ein Ge-

burtsdatum und eine Telefonnummer. Diese Merkmale definieren den Zustand einer Person und werden daher als Attribute der Klasse definiert. Um die Anschrift und das Geburtsdatum einer Person ausgeben zu können, werden die Methoden `druckeAnschrift` und `druckeTelefonNr` benötigt. Alle Objekte, die Personen repräsentieren, können nun mit Hilfe dieses Musters – also dieser Attribute und Methoden – beschrieben werden.

Allerdings reichen diese wenigen Attribute in der Regel nicht aus, um bestimmte Gruppen von Personen zu beschreiben. So gibt es z.B. an einer Universität u.a. Professoren, Studenten und Verwaltungsangestellte. Jede dieser Gruppen besitzt neben den allgemeinen Eigenschaften, die durch die Klasse *Person* abgedeckt werden, weitere zustandsbestimmende Merkmale und Verhaltensweisen. Daher wird für jede der drei Personengruppen eine eigene Klasse definiert, die sich aus der Klasse *Person* ableitet (vgl. die folgende Abbildung). Zwischen der Klasse *Person* einerseits und den Klassen *Professor*, *Student* und *Verwaltungsangestellte* andererseits besteht eine *Vererbungsbeziehung*. *Person* bezeichnet man als *Oberklasse* (oder *Superklasse*), die drei übrigen Klassen als *Unterklassen* (oder *Subklassen*).

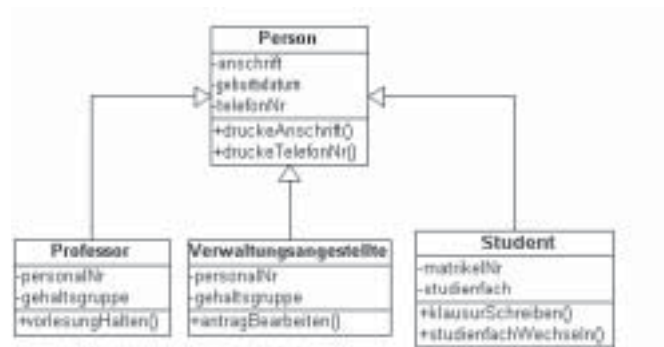


Abbildung 1.15: Die Superklasse *Person* und ihre Subklassen

Jede der drei Subklassen »erbt« nun die Eigenschaften der Superklasse. Das heißt, die Attribute `anschrift`, `geburtsdatum` und `telefonNr` sowie die beiden Methoden der Superklasse `Person` sind in den Subklassen verfügbar, ohne dass sie dort explizit genannt werden.

Darüber hinaus können für die Subklassen weitere Eigenschaften vereinbart werden. Diese werden dann, wie in der Abbildung 1.9 zu sehen, in die Klassenbeschreibung aufgenommen. So besitzen Professoren und Verwaltungsangestellte eine Personalnummer und eine Gehaltsgruppe, Studenten eine Matrikelnummer und ein Studienfach.

Die Methoden der Klassen beschreiben wiederum mögliche Verhaltensweisen der Instanzen dieser Klassen. Professoren halten Vorlesungen, Verwaltungsangestellte bearbeiten Anträge und Studenten schreiben Klausuren oder wechseln das Studienfach. Es handelt sich bei allen drei Klassen um spezielle Gruppen von Personen. Jedes zu diesen Klassen gehörende Objekt besitzt daher alle Eigenschaften der Superklasse `Person` und zusätzlich die Attribute und Methoden, die in der jeweiligen Subklasse definiert sind.

Mit der Vererbung kennen Sie nun eines der wichtigsten Konzepte der Objektorientierung. Die bisherigen Ausführungen beziehen sich auf den Fall, dass eine Unterklasse immer nur zu einer Oberklasse gehört. Man spricht hierbei von *Einfachvererbung*. Im praktischen Einsatz der objektorientierten Modellierung werden Sie jedoch häufig feststellen, dass einige Klassen Ihrer realen Problemstellung, bzw. deren Instanzen, sich mehr als nur einem Muster (einer Oberklasse) zuordnen lassen. Die Unterklasse erbt dann die Eigenschaften von zwei oder mehr Oberklassen. Dies bezeichnet man in der objektorientierten Terminologie als *Mehrfachvererbung* (auch: *multiple Vererbung*).



Sie haben grundsätzlich die Möglichkeit, multiple Vererbungsbeziehungen in Ihre objektorientierten Modelle einzubauen. Allerdings werden diese von einigen Programmiersprachen nicht

unterstützt oder sind nur über Umwege realisierbar (z.B. Java und Smalltalk). Hierfür gibt es einen guten Grund: Mehrfachvererbung ist nämlich nicht unproblematisch. Sie kann zu Inkonsistenzen in Ihrem Modell führen.

Die »Verweigerung« multipler Vererbungsbeziehungen durch einige Programmiersprachen dient also u.a. dem Schutz des Programmierers vor logischen Konflikten im Programm.

Polymorphie

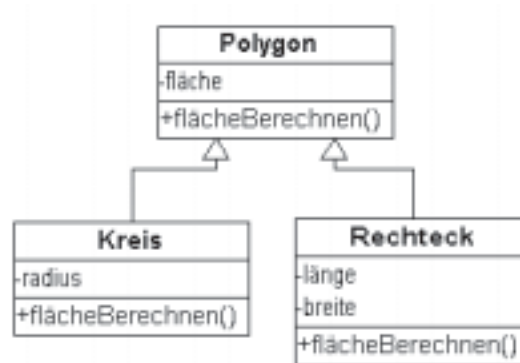
Polymorphie bedeutet »Vielgestaltigkeit«. In der Objektorientierung ist Polymorphie ein Konzept, das es Objekten erlaubt, auf ein und dieselbe Nachricht unterschiedlich zu reagieren.



Statt *Polymorphie* wird häufig auch die Bezeichnung *Polymorphismus* verwendet.

Untrennbar damit verbunden ist das sogenannte *späte Binden*. Dabei wird erst zur Laufzeit eines objektorientierten Programms festgelegt, welche Methode durch eine bestimmte Nachricht ausgelöst wird. Im Beispiel der folgenden Abbildung besitzt die Klasse *Polygon* zwei Unterklassen: *Kreis* und *Rechteck* (siehe Abbildung 1.16).

Alle drei Klassen der Abbildung besitzen eine Methode *flächeBerechnen*. Eigentlich wäre das gar nicht nötig. Aus dem Abschnitt zur Vererbung haben Sie bereits erfahren, dass Unterklassen alle Attribute und Methoden ihrer Oberklasse erben. Also besitzen doch hier die Klassen *Kreis* und *Rechteck* neben dem Attribut *fläche* schon eine Methode *flächeBerechnen* durch die Vererbung von *Polygon*.

**Abbildung 1.16:** Polymorphie

Nun wird aber die Fläche eines Kreises anders berechnet als die Fläche eines Rechtecks. Es ist daher sinnvoll, dass Unterklassen die Methoden ihrer Oberklassen *überschreiben* können. Genau dies geschieht in der obigen Abbildung. Die Klassen Kreis und Rechteck definieren ihre eigenen Methoden flächeBerechnen. Je nachdem, ob es sich bei dem aufgerufenen Objekt um eine Instanz der Klasse Polygon, Kreis oder Rechteck handelt, wird trotz desselben Methodennamens eine andere Rechenoperation ausgelöst.



Das Überschreiben von Methoden ist nicht mit dem Überladen zu verwechseln. Beim Überschreiben wird in einer Unterklasse die aus einer Oberklasse geerbte Methode neu definiert. Beim Überladen richtet sich die Ausführung einer Methode nach der jeweiligen Parameterliste.

Neben dem Überschreiben gibt es in der Objektorientierung auch ein so genanntes *Überladen* von Methoden. Dabei wird der Methodenname innerhalb einer Klasse mehrfach aufgeführt. Jede dieser Methoden beschreibt ein anderes Verhalten und besitzt eine andere Parameterliste. Beim Methodenaufruf erfolgt die Identifikation der

»richtigen« (also tatsächlich gemeinten) Methode durch die übergebene Parameterliste.

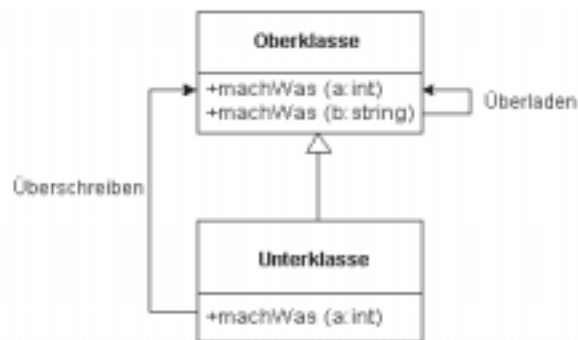


Abbildung 1.17: Überladen von Methoden

Im Beispiel der Abbildung wird die Methode `machWas` in der Unterklasse überschrieben. Sie wird mit demselben Parametertyp aufgerufen wie in der Oberklasse: mit einem ganzzahligen Parameter. Die zweite Methode mit dem Namen `machWas` in der Oberklasse hat hingegen statt einer Zahl eine Zeichenkette (Typ `String`) als Parameter. Welche der beiden Methoden in der Oberklasse bei einem Aufruf aktiviert wird, hängt vom Datentyp des mitgelieferten Parameters ab.

Objektorientierte Systementwicklung

Sie haben nun die wichtigsten Konzepte der Objektorientierung kennen gelernt. Wenn Sie bereits Erfahrungen mit klassischen Programmiersprachen gesammelt haben, dann werden Sie feststellen, dass die Objektorientierung ganz anders an Problemstellungen herangeht. Es ist leicht nachzuvollziehen, dass sich daraus auch ein

verändertes Vorgehen im Rahmen der Entwicklung objektorientierter Software ergibt.

Die objektorientierte Systementwicklung besteht aus den drei Phasen

- ✓ Objektorientierte Analyse (OOA),
- ✓ Objektorientiertes Design (OOD) und
- ✓ Objektorientierte Programmierung (OOP).

Die *objektorientierte Analyse* hat die Aufgabe, die Anforderungen an ein Softwaresystem zu ermitteln und die reale Problemstellung abzubilden. Völlig unabhängig von der späteren Implementierung der Software wird der abzubildende Problembereich in Objekten und Klassen sowie ihren Beziehungen beschrieben. Auch die Wahl der Programmiersprache spielt in dieser Phase noch keine Rolle. Die Herausforderung für den Systemanalytiker besteht darin, das objektorientierte Modell so weit von der Realität zu abstrahieren, dass es alle für den Auftraggeber relevanten Bestandteile der realen Problemstellung abbildet.

Das im Rahmen der objektorientierten Analyse erstellte Fachkonzept wird anschließend im *objektorientierten Design* an die Bedingungen der technischen Plattform angepasst und ergänzt um die Modellierung der Benutzungsoberfläche und die Gestaltung der Datenhaltung.

Eng verbunden mit dem objektorientierten Design ist die Implementierung des Modells in einer objektorientierten Programmiersprache (*objektorientierte Programmierung*). Aufgrund des streng modularen Aufbaus können entworfene Klassen direkt implementiert und getestet werden. So ist es möglich, dem Auftraggeber relativ schnell die Funktionalität des Systems vorzuführen und durch eine enge Benutzerbeteiligung am Entwicklungsprozess Fehler frühzeitig zu korrigieren.

Durch die laufende Anpassung des Systems entsteht ein evolutionärer Prozess, in dem die Phasen Analyse, Design und Implementie-

rung ständig wiederholt werden, bis eine zufrieden stellende Qualität der Software erzielt wird.

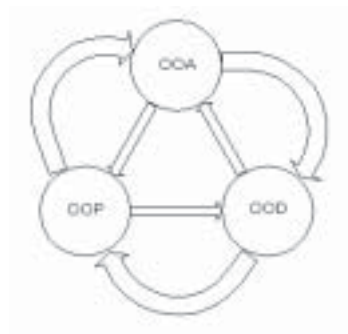


Abbildung 1.18: OOA, OOD und OOP

Wie in der Abbildung zu sehen, kann die Abfolge der Phasen Analyse, Design und Programmierung durchaus unterbrochen werden. Falls nötig, ist auch ein Rücksprung aus der Programmierung ins Design oder gar aus der Analyse in die direkte Implementierung möglich.

Damit haben Sie die wesentlichen Grundlagen der Objektorientierung kennen gelernt und können nun im folgenden Kapitel mit der UML beginnen. Die in diesem ersten Kapitel beschriebenen Konzepte der Objektorientierung werden Ihnen im weiteren Verlauf dieses Einsteigerseminars immer wieder begegnen.

KAPITEL

Grundlagen der UML

Dieses zweite Kapitel gibt einen Überblick über grundlegende Aspekte der UML. Sie erfahren Näheres zu ihren Zielen, ihrer historischen Entwicklung und der Spracharchitektur. Außerdem werden die wesentlichen Diagrammartarten einführend vorgestellt.

2



Grundlagen der UML

Die *Unified Modeling Language* (UML) ist eine formale Sprache zur Spezifikation, Visualisierung, Konstruktion und Dokumentation von Softwaresystemen und Geschäftsmodellen.

Sie unterstützt die Analyse realer Problemstellungen durch die Bereitstellung zahlreicher Diagramme, die in einem objektorientierten Modell verschiedene Aspekte aus jeweils unterschiedlichen Blickwinkeln beleuchten. Dem Modellierer steht mit der UML eine mächtige Sprache zur Verfügung, die es ihm ermöglicht, auch komplexe Zusammenhänge mit Hilfe objektorientierter Konzepte einfach und dennoch umfassend abzubilden.

Bevor wir in die objektorientierte Modellierung mit der UML einsteigen, soll Ihnen dieses Kapitel einen kurzen Überblick über den Hintergrund der Sprache geben. Neben der Zielrichtung und der historischen Entwicklung erfahren Sie hier etwas über die zugrunde liegende Spracharchitektur und erhalten einen ersten Einblick in die verschiedenen in der UML verfügbaren Diagramme zur Beschreibung einer Problemdomäne.

Motivation

Die Entwicklung leistungsfähiger Softwaresysteme ist ein äußerst komplexes Problem. Projekte dieser Art umfassen oft viele Mannjahre. Die zahlreichen Aspekte, die dabei zu berücksichtigen sind, erschweren die Überschaubarkeit des Gesamtprojekts. Anwender und Entwickler arbeiten zunehmend in allen Phasen der Softwareentwicklung, von der Analyse bis hin zur Implementierung, zusammen.

Daraus ergeben sich häufig Schwierigkeiten in der Kommunikation: Der Anwender versteht den Programmierer nicht und umge-

kehrt. Ähnlich sieht es beim Reengineering von Geschäftsprozessen aus. Auch solche Projekte leiden oftmals unter der fehlenden oder mangelhaften gemeinsamen Kommunikationsbasis zwischen den beteiligten Gruppen.

Modelle – seien es nun Softwaremodelle oder Geschäftsprozessmodelle – können hier hilfreiche Unterstützung bieten. Sie reduzieren die oft enorme Komplexität von Sachverhalten auf ein überschaubares Maß. Indem wir von dem Gesamtproblem abstrahieren und vereinfachte Modelle bilden, können wir komplexe Zusammenhänge besser verstehen und gedanklich verarbeiten.

Darüber hinaus sind Modelle auch geeignet, Projektbeteiligte mit unterschiedlicher Spezialisierung und Erfahrung zusammenzubringen und ihnen die Diskussion gemeinsamer Aufgaben zu erleichtern. Modelle stellen ein Kommunikationsmittel dar, das ihnen eine gemeinsame Sprache und somit gegenseitiges Verständnis ermöglicht.

Vor der Entstehung der UML existierte eine Vielzahl objektorientierter Modellierungssprachen, die alle unterschiedliche (sich zum Teil auch überschneidende) Modellelemente, Notationen und Regeln besaßen. Richtig durchsetzen konnte sich keine dieser Sprachen, was viele Anwender davor zurückschrecken ließ, sich mit objektorientierten Modellen zu beschäftigen.

Die Entwicklung der UML erfolgte (und erfolgt) vor diesem Hintergrund auf der Basis folgender Ziele:

- ✓ *Bereitstellung einer konkreten, visuellen Modellierungssprache für die Entwicklung und den Austausch umfassender, aussagekräftiger Modelle*

Für die objektorientierte Analyse ist es von großer Bedeutung, dass die verwendete Modellierungssprache *konkrete*, allgemein akzeptierte Konzepte zur Verfügung stellt, die wesentliche Bereiche einer Anwendung abdecken. Die bloße Definition abstrakter Gestaltungsvorschriften birgt gerade beim Austausch von Modellen die Gefahr von Inkonsistenzen und Überschneidungen. Gleiches gilt für die Schnittstellen zwischen den Mo-

dellen verschiedener Sichten auf ein und denselben Gestaltungsgegenstand.

✓ *Erweiterbarkeit und Verfügbarkeit von Mechanismen zur Ausweitung der Kernkonzepte*

Die objektorientierte Analyse erfreut sich in der Praxis einer ständigen Ausweitung ihrer Einsatzgebiete. Neben der Software-Entwicklung und der Geschäftsprozessanalyse werden objektorientierte Konzepte z.B. auch zunehmend für die Organisationsgestaltung entdeckt. Neue Anwendungen verlangen oftmals spezielle Modellelemente, die eine Anpassung der Modellierungssprache erfordern. Um die Einheitlichkeit der UML zu gewährleisten, sollten dabei jedoch nicht die Kernkonzepte selbst verändert werden. Vielmehr sind diese so formuliert, dass sich benötigte Erweiterungen direkt aus ihnen ableiten lassen. Die Design-Prinzipien der UML beruhen daher u.a. auf den folgenden Anforderungen:

- ✓ Der Benutzer soll in der Lage sein, die meisten einfachen Anwendungen mit Hilfe der Kernkonzepte der UML abzubilden, ohne individuelle Erweiterungen vornehmen zu müssen.
- ✓ Spezialfälle sollen möglichst durch die Ableitung neuer Elemente aus den bestehenden Konzepten lösbar sein.
- ✓ Für Anwendungsbereiche, die durch die Kernkonzepte nicht abgedeckt werden, sollen diese durch neue Konzepte und Notationen ergänzt werden können.
- ✓ *Unterstützung von Spezifikationen, unabhängig von den eingesetzten Programmiersprachen und Vorgehensmodellen im Entwicklungsprozess*
Die UML soll alle gängigen (objektorientierten) Programmiersprachen unterstützen. Darüber hinaus soll sie im Umfeld möglichst vieler verschiedener Entwicklungsmethoden und Vorgehensmodelle eingesetzt werden können.

✓ *Aufbau auf einer formalen Basis zur Förderung des Verständnisses der Modellierungssprache*

Ein Dilemma formaler Modellierungssprachen besteht in der Abwägung zwischen der notwendigen Präzision der formalen Definition einerseits und dem bequemen Einsatz der Sprache andererseits. Die UML verfolgt den Anspruch, leicht erlernbar und anwendbar zu sein, dabei jedoch gleichzeitig auf einem in formaler Hinsicht wohlgedachten Konzept zu basieren.

✓ *Förderung objektorientierter Entwicklungswerkzeuge*

Die Durchsetzung eines Quasi-Standards in der objektorientierten Modellierung fördert auch die Entwicklung und Verbreitung softwaregestützter Werkzeuge. Erst durch eine gemeinsame Modellierungssprache mit einheitlichen Basiskonzepten werden die Modelle über Toolgrenzen hinweg austauschbar.

✓ *Unterstützung weitergehender Entwicklungskonzepte wie Komponenten, Frameworks und Patterns*

Die UML fördert die Entwicklung weitergehender Konzepte, die auf ihren Basiselementen aufbauen. Durch die Ableitung aus den bestehenden Konzepten der UML ist die Interoperabilität von Komponenten, Frameworks und Patterns leichter zu gewährleisten.

Die UML führt die Konzepte verbreiteter und bewährter Modellierungssprachen, wie Booch, OMT und OOSE, zusammen. Sie überwindet somit die Grenzen der Austauschbarkeit objektorientierter Modelle und vereint die Anwender objektorientierter Modellierung in einer einzigen Sprachregelung.

Sie zielt auf die Etablierung einer Standard-Modellierungssprache und nicht etwa eines bestimmten Vorgehensmodells ab. Da die Modellierungsprozesse bei der Entwicklung von Software und der Gestaltung von Geschäftsprozessen in Abhängigkeit vom Unternehmenskontext und der jeweiligen Problemdomäne stark variieren, ist die UML auf einen weit gefächerten Einsatzbereich – unabhängig vom speziellen Vorgehensmodell – zugeschnitten.

Historie

Mitte der siebziger Jahre tauchten die ersten Modellierungssprachen für die objektorientierte Analyse und das objektorientierte Design auf. Bis etwa Mitte der neunziger Jahre entstanden mehr als 50 Ansätze, die sich teilweise nur in Nuancen, oft aber auch von ihrer grundsätzlichen Zielsetzung her unterschieden. Recht bekannte Modellierungsmethoden wurden entwickelt von Grady Booch, James Rumbaugh, Ivar Jacobson sowie Peter Coad und Edward Yourdan.

1995 machten sich Booch und Rumbaugh im Rahmen der Rational Software Corporation daran, ihre Notationen zur so genannten *Unified Method (UM)* zu vereinen. Wenig später stieß Ivar Jacobson dazu und brachte Elemente seines *Object Oriented Software Engineering (OOSE)* ein. Dies war die Geburtsstunde der UML-Version 0.9.

Der Zusammenschluss dieser drei Vorreiter objektorientierter Modellierung (seitdem in der Fachwelt bekannt als »die drei Amigos«) führte dazu, dass sich immer mehr namhafte Firmen und andere Organisationen den Bemühungen um eine einheitliche Sprache anschlossen. Darunter waren u.a. Hewlett-Packard, IBM, Microsoft und Oracle. So wurde die UML-Version 1.1 entwickelt, die 1997 bei der *Object Management Group (OMG)* zur Standardisierung eingebracht und akzeptiert wurde. Seitdem ist die OMG verantwortlich für die Pflege und Weiterentwicklung der UML, die inzwischen nach kleineren Änderungen und Ergänzungen in der Version 1.3 vorliegt. Aktuell werden weitere kosmetische Korrekturen anhand eines Entwurfs für die Version 1.4 diskutiert.



Die Object Management Group (OMG) ist eine Vereinigung von Entwicklern und Anwendern objektorientierter Technologien, die versuchen, allgemein anerkannte Standards und Spezifikationen für die Objektorientierung zu schaffen, welche die Entwicklung offener objektorientierter Systeme ermöglichen. Sie wurde 1989 gegründet und hat derzeit ca. 1000 Mitglieder.

Die Standardisierung durch die Object Management Group hat der Verbreitung der UML einen gewaltigen Schub verliehen. Inzwischen wird sie weltweit in vielen Softwareprojekten eingesetzt. Doch auch im Rahmen des Business Reengineering erfreut sie sich immer größerer Beliebtheit. Zunehmend wird erkannt, dass die Objektorientierung nicht nur ein Programmierkonzept ist, sondern eine Erfolg versprechende Herangehensweise insbesondere an schlecht strukturierte Problemstellungen auch betriebswirtschaftlicher Art darstellt.

Spracharchitektur

Die Sprachelemente der UML, die Sie im Rahmen Ihrer Projekte zur Modellierung einsetzen, sind an sich recht einfach und bestehen im Wesentlichen aus grafischen Darstellungen. Gleichwohl basieren sie auf einer eindeutig definierten Spracharchitektur, um die jederzeitige Konsistenz der Modelle zu gewährleisten.

Die folgenden Ausführungen geben Ihnen einen kurzen Einblick in die Spracharchitektur der UML.

UML-Modelle umfassen grob zwei Arten von Objektmodellen:

- ✓ *Statische Modelle* (Strukturmodelle) beleuchten die Struktur von Objekten in einem System. Dazu gehören ihre Klassen, Schnittstellen, Attribute und Beziehungen zu anderen Objekten.
- ✓ *Dynamische Modelle* (Verhaltensmodelle) zeigen das Verhalten der Objekte in einem System. Hierzu zählen ihre Methoden, Zustände und Interaktionen sowie ihre Zusammenarbeit bei der Lösung einer Problemstellung.

Die Architektur der UML basiert auf einer vierschichtigen Metamodell-Struktur. Im Einzelnen sind dies folgende Schichten (Layer):

- ✓ Objekte,
- ✓ Modell,
- ✓ Metamodell und
- ✓ Meta-Metamodell.

Die folgende Tabelle enthält eine Übersicht, in der die einzelnen Schichten beschrieben werden.

Schicht	Beschreibung	Beispiel
Meta-Metamodell	Infrastruktur für eine Meta-modell-Architektur; definiert die Sprache zur Spezifikation eines Metamodells	Meta-Klasse, Meta-Attribut, Meta-Operation
Metamodell	Instanz eines Meta-Metamodells; definiert die Sprache zur Spezifikation eines Modells	Klasse, Attribut, Operation, Komponente
Modell	Instanz eines Metamodells; definiert eine Sprache zur Beschreibung der Problemdomäne	Auto, Besitzer, Typ, Farbe, Fahren, Bremsen
Objekte	Instanz eines Modells; definiert eine spezifische Problemdomäne	Peters grüner Golf

Tabelle 2.1: Vierschichtige Metamodell-Architektur der UML

Die Schicht des *Meta-Metamodells* bildet die Grundlage der Architektur. Ihre Aufgabe besteht darin, die Sprache zur Spezifikation eines Metamodells zu definieren. Das Meta-Metamodell verkörpert die höchste Abstraktionsstufe im Rahmen der Vier-Schichten-Architektur. Es kann viele verschiedene Metamodelle definieren. Umgekehrt können mit jedem Metamodell mehrere Meta-Metamodelle verbunden sein (z.B. Meta-Klasse, Meta-Attribut, Meta-Operation).

Ein *Metamodell* ist eine Instanz eines Meta-Metamodells. Es definiert eine Sprache zur Spezifikation von Modellen. Metamodelle

sind konkreter als Meta-Metamodelle und besitzen ihrerseits Modelle als Instanzen (z.B. Klasse, Attribut, Operation, Komponente).

Die Instanzen eines Metamodells heißen *Modelle*. Sie haben die Aufgabe, eine Sprache zu definieren, mit der eine Problemdomäne beschrieben werden kann. Modelle sind somit näher an einer konkreten Problemstellung als Metamodelle (z.B. Auto, Besitzer, Typ, Farbe, Fahren, Bremsen).

Die geringste Abstraktionsstufe im Rahmen der vierschichtigen Architektur der UML bilden die *Objekte*. Sie sind Instanzen von Modellen und beschreiben eine spezifische Problemdomäne. Objekte repräsentieren konkrete Elemente der Problemdomäne (z.B. Peters grünen Golf).

Die logische Struktur des UML-Metamodells ist in *Pakete* (Packages) gegliedert. Diese Pakete fassen Metaklassen zusammen, die untereinander starke Gemeinsamkeiten aufweisen und mit Metaklassen in anderen Paketen lose verbunden sind.

- ✓ Das *Foundation Package* stellt die Sprach-Infrastruktur zur Spezifikation der statischen Modellstruktur bereit. Es ist seinerseits aus drei Unterpaketen zusammengesetzt:
- ✓ Das *Core Package* definiert die grundlegenden Konstrukte des Metamodells, die für die Entwicklung von Objektmodellen benötigt werden. Dazu zählen u.a. die Konstrukte Assoziation, Attribut, Beziehung (Abhängigkeit, Generalisierung etc.), Datentyp, Klasse, Komponente, Schnittstelle und Zusicherung. Diese Konstrukte werden in den folgenden Kapiteln noch ausführlich beschrieben.
- ✓ Im *Extension Mechanisms Package* ist festgelegt, wie die standardmäßigen Modellelemente der UML angepasst und erweitert werden können. Obwohl die UML bereits eine Vielzahl von Modellelementen zur Verfügung stellt, reichen diese u.U. nicht aus, um sämtliche Details sehr spezieller Problemdomänen abzubilden und die Anforderungen aller Anwender zu erfüllen. Daher existieren durch das Extension Mechanisms Package

eindeutige Regeln, die eine individuelle und dennoch konsistente Anpassung und Erweiterung der UML ermöglichen.



Für die meisten Anwendungsbereiche ist die Standarddefinition der UML vollkommen ausreichend. Anpassungen und insbesondere Erweiterungen sollten daher nur dann vorgenommen werden, wenn es für die Modellierung absolut unabdingbar ist. Erweiterungen berauben Objektmodelle einer der Stärken der Standardisierung: Sie sind nicht mehr allgemein verständlich, kompatibel und uneingeschränkt wiederverwendbar.

- ✓ Die Datentypen, die zur Definition der UML verwendet werden, sind im *Data Types Package* abgelegt. Diese Datentypen sind nicht zu verwechseln mit jenen, die der Anwender später in seinem Objektmodell verwendet. Das *Data Types Package* enthält Meta-Datentypen für die Definition der UML selbst.
- ✓ Das *Behavioral Elements Package* legt eine Struktur fest, mit der das dynamische Verhalten von Modellen spezifiziert wird. Die verschiedenen Unterpakete (Subpackages) dieses Pakets werden in den folgenden Kapiteln noch näher beleuchtet. Es sind im Einzelnen:
 - ✓ Das *Common Behavior Package* definiert die grundlegenden Konzepte für Verhaltenselemente der UML.
 - ✓ Das *Collaborations Package* beschreibt die Verhaltenskonzepte von Modellobjekten bei der Erfüllung bestimmter Aufgaben.
 - ✓ Das Paket *Use Cases* spezifiziert das Verhalten für Akteure und Anwendungsfälle (Use Cases).
 - ✓ Das *State Machines Package* ist für die Spezifikation des Verhaltens in Zustandsautomaten verantwortlich.
 - ✓ Das *Activity Graphs Package* definiert Verhaltenselemente in Aktivitätsdiagrammen als Spezialfall von Zustandsdiagrammen zur Modellierung von Prozessen.

- ✓ Das *Model Management Package* ist abhängig vom *Foundation Package*. Es definiert die Begriffe Modell, Paket und Subsystem. Pakete werden benutzt, um Modellelemente mit ähnlichen Eigenschaften zu gruppieren. Subsysteme sind spezielle Arten von Paketen, die Einheiten des physischen Systems in einem Objektmodell repräsentieren.

Die genannten Pakete der UML sind noch weiter untergliedert. Eine eingehende Beschreibung würde jedoch den Rahmen dieses Taschenbuchs sprengen. Falls Sie sich für die Feinheiten der UML-Architektur interessieren, nehmen Sie die Spezifikation der Object Management Group zur Hand. Dort finden Sie tiefer gehende Informationen (vgl. auch die Literaturhinweise im Anhang).

Diagrammarten

Die UML stellt Ihnen verschiedene Diagramme zur Verfügung, um die objektorientierte Analyse Ihrer Problemdomäne grafisch darzustellen. Diese Diagramme unterscheiden sich durch den Blickwinkel, aus dem sie die Problemstellung betrachten.



Die Diagramme der UML analysieren eine Problemstellung aus den vier Blickwinkeln interne, externe, statische und dynamische Sicht.

Im Wesentlichen können dabei zwei Kategorien von Sichtweisen unterschieden werden:

- ✓ Externe vs. interne Sicht
 - ✓ Bei der *externen Betrachtung* (Außensicht) ist das zu analysierende System als »black box« zu betrachten. Von Interesse sind dabei lediglich der Input und der Output des Systems. Ein UML-Modell stellt in der externen Sicht also lediglich dar, was ein System (z.B. ein Geschäftsprozess oder ein Objekt) leistet und welche Informationen dazu benötigt

werden. Wie ein Ergebnis erzielt wird, ist in der externen Sicht belanglos.

- ✓ Die *interne Sicht* (Innensicht) zeigt an, wie ein System arbeitet, um zu einem Ergebnis zu kommen. Die »black box« wird hier geöffnet und näher untersucht.
- ✓ Statische vs. dynamische Sicht
 - ✓ Die *statische Sicht* betrachtet die Struktur eines Anwendungsbereichs zu einem bestimmten *Zeitpunkt*. Das System wird quasi »eingefroren« und so in seinen Bestandteilen beleuchtet. Statische Aspekte eines objektorientierten Modells sind häufig auch unabhängig vom betrachteten Zeitpunkt (z.B. Klassendiagramm).
 - ✓ In der *dynamischen Sicht* wird das Verhalten des Systems bzw. seiner Elemente analysiert. Es handelt sich hier also um eine *Zeitraum*-Betrachtung. Von besonderem Interesse sind die Aktionen der Objekte und ihre Zusammenarbeit bei der Lösung einer Aufgabe.

Mit Hilfe dieser vier Sichten ermöglicht die UML dem Modellierer die Wahl verschiedener Blickwinkel auf eine Problemstellung in beliebigen Abstraktionsstufen. Je nach Art des zu analysierenden Systems und der gewünschten Informationen werden die verschiedenen Diagramme der UML eingesetzt. Durch die Kombination interner, externer, statischer und dynamischer Sichten lässt sich das System vollständig analysieren bzw. modellieren.

Wir wollen Ihnen die verschiedenen von der UML zur Verfügung gestellten Diagramme hier vorab schon einmal kurz vorstellen, damit Sie einen ersten Eindruck von den Möglichkeiten der Modellierungssprache bekommen. Im anschließenden zweiten Teil dieses Taschenbuchs werden die Diagramme dann ausführlich anhand anschaulicher Beispiele erläutert.

- ✓ *Anwendungsfalldiagramme* dienen der einfachen Darstellung von Prozessen (so genannten Anwendungsfällen). Sie beschreiben diese Prozesse mit ihren Beziehungen untereinander und zu be-

teiligten Personen, Ereignissen oder anderen Prozessen. Häufig werden für einen Anwendungsfall auch die Begriffe *Use Case* oder *Geschäftsvorfall* verwendet.

- ✓ *Klassendiagramme* beschreiben die statische Struktur eines Objektmodells. Sie zeigen die Eigenschaften und gegenseitigen Beziehungen von Klassen bzw. der aus diesen Klassen erzeugten Objekte.
- ✓ *Verhaltensdiagramme* repräsentieren dynamische Sachverhalte innerhalb eines objektorientierten Modells. Die UML stellt vier verschiedene Arten von Verhaltensdiagrammen zur Verfügung, deren unterschiedliche Betrachtungsweisen sich zu einem objektorientierten Gesamtbild des analysierten Problemereichs ergänzen:
 - ✓ *Sequenzdiagramme*
 - ✓ *Kollaborationsdiagramme*
 - ✓ *Zustandsdiagramme*
 - ✓ *Aktivitätsdiagramme*
- ✓ *Implementierungsdiagramme* unterstützen die Umsetzung objektorientierter Modelle in Programmcode. Sie zeigen die Struktur des Quellcodes eines Programms und der Implementierung zur Laufzeit. Die UML stellt zwei spezielle Ausprägungen von Implementierungsdiagrammen zur Verfügung:
 - ✓ *Komponentendiagramme*
 - ✓ *Verteilungsdiagramme*

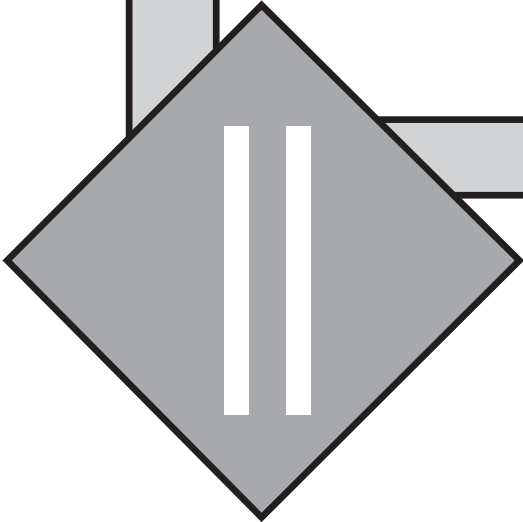
Damit haben Sie die Grundlagen der Objektorientierung und der UML bereits hinter sich. Im zweiten Teil dieses Taschenbuchs gehen wir anschließend ausführlich auf die einzelnen Diagramme der UML ein. Wir werden dabei die angesprochenen vier Sichten aufgreifen und jedes Diagramm anhand anschaulicher Beispiele erläutern.



TEIL

Techniken – Modelle der UML

Der zweite Teil dieses Taschenbuchs bringt Ihnen die einzelnen Diagramme der UML näher. Anhand konkreter Beispiele werden die Diagramme sowohl konzeptionell als auch in ihrer Umsetzung mit einem Modellierungstool erläutert.





KAPITEL

Überblick

Das dritte Kapitel eröffnet den zweiten Teil dieses Taschenbuchs mit einem Überblick über die Diagramme der UML. Dazu wird nach dem Blickwinkel auf die abzubildende Problemstellung in Diagramme für die Außensicht und für die Innensicht differenziert.

3



Überblick

Wie Sie bereits im letzten Kapitel erfahren haben, stellt die UML Diagramme für die objektorientierte Modellierung zur Verfügung, die sich durch ihre Sichtweisen auf die zu analysierende Problemstellung unterscheiden.

Zwei sich ergänzende Blickwinkel sind die *Außensicht* (externe Sicht) und die *Innensicht* (interne Sicht).

Außensicht (Use-Case-Diagramme)

Wie bereits im ersten Teil im Rahmen der Grundlagen der UML erläutert, wird in der Außensicht das zu analysierende System als »black box« betrachtet.

Der Nutzer des Systems braucht nichts über die Abläufe innerhalb des Systems selbst zu wissen. Für ihn ist lediglich von Interesse, *was* das System leistet, welche Art von Leistung er zu erwarten hat. Außerdem muss der Nutzer darüber informiert sein, wie er an diese Leistung gelangen kann, welchen Input das System zu deren Erbringung benötigt. Die Abläufe innerhalb des Systems werden in der Außensicht vernachlässigt.

Zur Modellierung der Außensicht dienen in der UML die *Use-Case-Diagramme* (auch Anwendungsfall-Diagramme genannt). Sie stellen einzelne Geschäftsvorfälle (bei der Geschäftsprozess-Modellierung) oder Anwendungsfälle (bei der Software-Modellierung) grafisch dar.

Elemente von Use-Case-Diagrammen sind *Akteure* und *Anwendungsfälle*.

Akteure



Geschäftsvorfälle werden stets von Akteuren angestoßen.

Dabei muss es sich nicht unbedingt um Personen handeln. Auch externe Ereignisse oder Prozesse können einen Geschäftsvorfall initiieren und somit als Akteure in Erscheinung treten.

Akteure treten in einem Geschäftsprozess nicht in ihrer Eigenschaft als Person (bzw. Ereignis oder Prozess) auf, sondern schlüpfen in eine *Rolle*. Die Rolle repräsentiert in der Regel eine bestimmte Funktion innerhalb des Geschäftsprozesses. Sie kann von verschiedenen Akteuren gleichzeitig oder nacheinander eingenommen werden. Bei einer Hotelbuchung tritt beispielsweise eine Rolle mit der Bezeichnung »Portier« auf. Welcher Mitarbeiter diese Rolle als Akteur ausfüllt, ist für die Beschreibung dieses Systems in der UML unerheblich.

Akteure werden in einem Anwendungsfall-Diagramm wahlweise grafisch oder durch eine textuelle Bezeichnung dargestellt, wie die folgende Abbildung veranschaulicht.

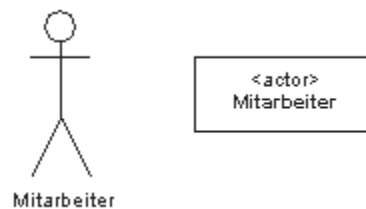


Abbildung 3.1: Darstellung von Akteuren in Anwendungsfall-Diagrammen

Die Abbildung zeigt auf der linken Seite die grafische und auf der rechten Seite die textuelle Darstellung eines Akteurs (Rolle) mit

der Bezeichnung Mitarbeiter. Das grafische Symbol für einen Akteur besteht aus einem »Strichmännchen« mit der entsprechenden Bezeichnung darunter. In der textuellen Darstellung wird lediglich der Rollename, ergänzt durch das Schlüsselwort <actor>, in einem einfachen Rahmen verwendet.

Anwendungsfälle

Anwendungsfälle werden von Akteuren initiiert und bestehen aus einzelnen Aktivitäten. Sie besitzen als Output ein eindeutiges Resultat.

Obwohl es sich bei einem Use Case zunächst um ein Beschreibungsinstrument der Außensicht handelt, ist es hilfreich, Anwendungsfälle im weiteren Verlauf der objektorientierten Analyse schriftlich zu dokumentieren. Dabei sollte der Ablauf des Prozesses möglichst detailliert dargelegt werden. Bedingungen, Regeln, Anforderungen und Alternativen sind in einer solchen Definition ebenfalls zu berücksichtigen.

Für den Geschäftsvorfall einer Hotelbuchung sieht eine schriftliche Dokumentation etwa wie folgt aus:

- ✓ Anwendungsfall: Hotelbuchung
- ✓ Bedingung: Zimmer in Zeitraum und Kategorie verfügbar
- ✓ Alternative: Andere Kategorie anbieten
- ✓ Ablauf:
 - ✓ Verfügbarkeit prüfen
 - ✓ Gast registrieren
 - ✓ Schlüssel aushändigen
 - ✓ Gepäck zum Zimmer transportieren
 - ✓ Rechnung erstellen
 - ✓ Zahlung abwickeln

Die folgende Abbildung zeigt ein vereinfachtes Anwendungsfall-Diagramm einer Hotelbuchung. Alle betroffenen Anwendungsfälle werden in einem Rahmen dargestellt und mit einer möglichst aussagekräftigen Bezeichnung beschriftet. Bei der Verwaltung einer Vielzahl von Anwendungsfällen ist es sinnvoll, diese zusätzlich zu nummerieren.

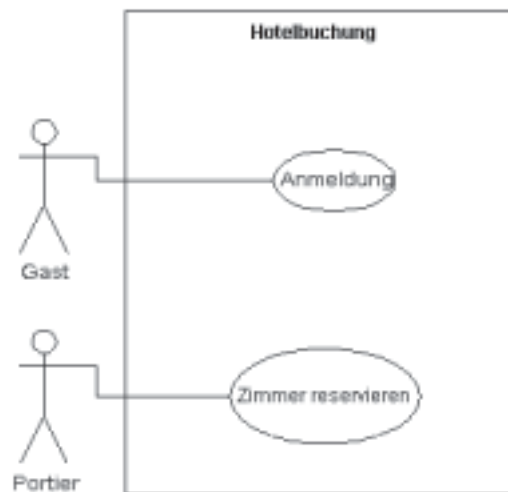


Abbildung 3.2: Vereinfachter Use Case »Hotelbuchung«

Innerhalb des Rahmens stehen die einzelnen Anwendungsfälle. Ihre jeweiligen Beziehungen zu Akteuren sind in der Abbildung durch Verbindungslinien angedeutet. Im Beispiel der obigen Abbildung ist der Akteur Guest mit dem Anwendungsfall Anmeldung verbunden, der Akteur Portier ist an dem Anwendungsfall Zimmer reservieren beteiligt.

Die Aussagekraft von Anwendungsfall-Diagrammen ist allerdings beschränkt. Sie geben nur einen groben Überblick über die Ge-

schäftsvorfälle. Die Abläufe innerhalb eines Prozesses sind nicht im Einzelnen erkennbar. Dafür gibt es mit den Verhaltens-Diagrammen eigene Notationen, die ergänzend zu den Anwendungsfall-Diagrammen eingesetzt werden können.

Anwendungsfall-Diagramme geben einen Überblick über eine Menge von Anwendungsfällen, die untereinander auf verschiedene Art Beziehungen bilden können:

- ✓ Das Grundkonzept der Vererbung aus der Objektorientierung findet auch in Anwendungsfall-Diagrammen Verwendung. Mit Hilfe von *Generalisierungen* können Unterfälle zu Anwendungsfällen gebildet werden. Ein Unterfall beschreibt einen speziellen Anwendungsfall, indem er die Eigenschaften eines generellen Anwendungsfalls erbt und diese erweitert oder überschreibt. Grafisch wird eine solche Beziehung durch einen Pfeil vom speziellen zum generellen Anwendungsfall dargestellt, der mit der Bezeichnung »erbt« oder »extends« versehen wird.
- ✓ *Include*-Beziehungen bezeichnen den Zugriff eines Anwendungsfalls auf einen anderen. Teilprozesse, die in mehreren Geschäftsvorfällen auftreten, können somit für verschiedene Anwendungsfälle wiederverwendet werden, ohne wiederholt beschrieben werden zu müssen. Grafisch wird die Include-Beziehung ebenfalls durch einen Pfeil von dem verwendenden Anwendungsfall zum verwendeten Teilprozess dargestellt. Er erhält die Beschriftung »verwendet« oder »uses«.
- ✓ Die dritte Form der Beziehung zwischen Anwendungsfällen ist die *Extend*-Beziehung. Sie dient dazu, Erweiterungen von Anwendungsfällen darzustellen, die von bestimmten Bedingungen abhängen. Es handelt sich um optionale Verzweigungen im Ablauf eines Geschäftsprozesses.

Die folgende Abbildung veranschaulicht die drei genannten Arten von Beziehungen anhand des Beispiels eines Fahrzeug-Service in einer Kfz-Werkstatt.

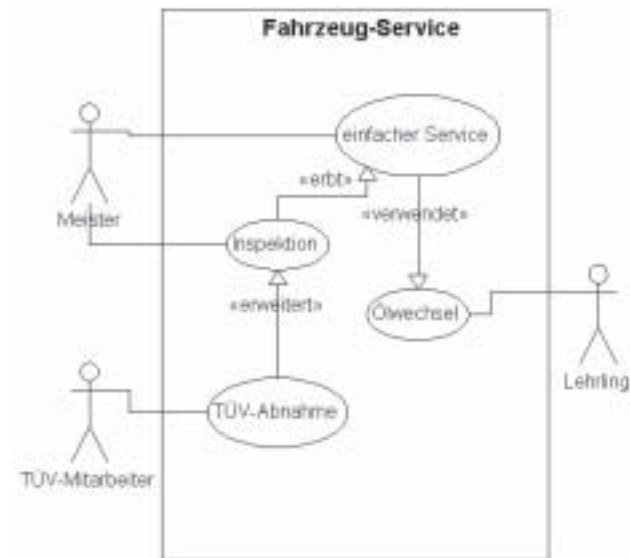


Abbildung 3.3: Anwendungsfall-Diagramm »Fahrzeug-Service«

In dem vereinfachten Beispiel der Abbildung ist der Akteur Meister für die Anwendungsfälle einfacher Service und Inspektion zuständig. Die Rolle Lehrling tritt im Rahmen des Anwendungsfalls Ölwechsel auf und der TÜV-Mitarbeiter nimmt die TÜV-Abnahme vor.

Im Rahmen des einfachen Service wird stets auch ein Ölwechsel vorgenommen. Der Anwendungsfall einfacher Service verwendet also den Teilprozess Ölwechsel. Darüber hinaus kann ein Ölwechsel auch als alleiniger Anwendungsfall oder im Zusammenhang mit anderen Anwendungsfällen vorkommen.

Die Inspektion ist ein Spezialfall des einfachen Service. Über die Leistungen des Letzteren hinaus fallen bei der Inspektion noch zusätzliche Arbeiten an. Insofern ist der Anwendungsfall Inspektion

eine Spezialisierung des Anwendungsfalls einfacher Service. In der objektorientierten Terminologie *erbt* der Anwendungsfall Inspektion die Eigenschaften des Anwendungsfalls einfacher Service und fügt weitere, eigene Eigenschaften hinzu.

Die dritte Art von Beziehungen zwischen Anwendungsfällen zeigt die TÜV-Abnahme. Sie ist nur unter bestimmten Bedingungen im Rahmen einer Inspektion durchzuführen. Nur dann, wenn die Gültigkeit der TÜV-Plakette abgelaufen ist, wird der Anwendungsfall TÜV-Abnahme als *Erweiterung* des Anwendungsfalls Inspektion in den Fahrzeug-Service einbezogen.



Bitte beachten Sie, dass die TÜV-Abnahme ausschließlich eine Erweiterung des Anwendungsfalls Inspektion ist. Nach der Aussage der Abbildung hat sie nichts mit den Anwendungsfällen einfacher Service oder gar Ölwechsel zu tun.

Leider sind die Bezeichnungen der Beziehungen zwischen Anwendungsfällen ein wenig verwirrend. In der Objektorientierung wird eine Vererbungsbeziehung häufig auch als Erweiterungsbeziehung bezeichnet. Die Erweiterung in Use-Case-Diagrammen meint aber gerade keine Vererbung zwischen Anwendungsfällen, sondern lediglich eine optionale Ausweitung eines Anwendungsfalls um weitere Aktivitäten bzw. Prozesse.

Use-Case-Diagramme dienen der Darstellung der Außensicht eines Objektmodells. Einzelne Arbeitsschritte (Teilprozesse), die zur internen Durchführung eines Vorgangs benötigt werden, werden in das Anwendungsfall-Diagramm nicht aufgenommen. In dem Beispiel des Fahrzeug-Service wird das Fahrzeug im Rahmen der Inspektion u.a. einer Sichtprüfung unterzogen, die sich wiederum aus mehreren Arbeitsschritten zusammensetzt. Auch der Ölwechsel besteht aus einzelnen Arbeitsschritten, die für den ausführenden Akteur oder den Betrieb (z.B. bei der Abrechnung) durchaus relevant sind.

Hier wird deutlich, dass es in der Objektorientierung stets auf die Sichtweise ankommt, die man bei der Betrachtung einer Aufgaben-

stellung einnimmt. Wir untersuchen in der obigen Abbildung den Use Case Fahrzeug-Service. Dieser Blickwinkel ist entscheidend für die Definition, welche (Teil-) Prozesse als intern und welche als extern zu bezeichnen sind.

Ein Fahrzeug-Service kann in unserem Beispiel aus einem einfachen Service, einer Inspektion oder lediglich aus einem Ölwechsel oder einer TÜV-Abnahme bestehen. Die Beziehungen zwischen diesen Anwendungsfällen sind in der Abbildung dargestellt. Alle weitergehenden Informationen über die Abläufe und Zusammenhänge innerhalb der genannten Anwendungsfälle sind für die Außensicht im Rahmen des Blickwinkels »Fahrzeug-Service« unwichtig.

Das ändert sich, sobald Sie Ihren Blickwinkel verschieben und den Fahrzeug-Service nicht mehr aus der Sicht des Außenstehenden betrachten. »Zoomen« Sie etwa in den Fahrzeug-Service hinein und betrachten beispielsweise die Inspektion als neuen Use Case, so besteht Ihre Außensicht nunmehr aus den verschiedenen Anwendungsfällen, die im Rahmen der Inspektion eine Rolle spielen. Ihr Diagramm besteht dann u.a. aus den Anwendungsfällen »Bremsen prüfen«, »Lenkung prüfen«, »Zündkerzen wechseln« und deren Beziehungen.



Wie den Bildausschnitt einer Kamera können Sie in der Objekt-orientierung Ihren Blickwinkel auf eine Problemstellung variieren und erhalten so unterschiedliche Objektmodelle. Die Kunst der objektorientierten Analyse besteht daher nicht nur in dem korrekten Einsatz objektorientierter Konzepte. Ebenso wichtig ist die Auswahl eines geeigneten Blickwinkels auf die zu lösende Aufgabe und die sinnvolle Abgrenzung der Problemstellung.

Für das Beispiel des Weckers, anhand dessen wir Ihnen in diesem zweiten Teil des Buchs die einzelnen Diagramme der UML erläutern wollen, gibt das Use-Case-Diagramm nicht allzu viel her. Der Vollständigkeit halber wollen wir Ihnen aber dennoch eine Möglichkeit vorschlagen.

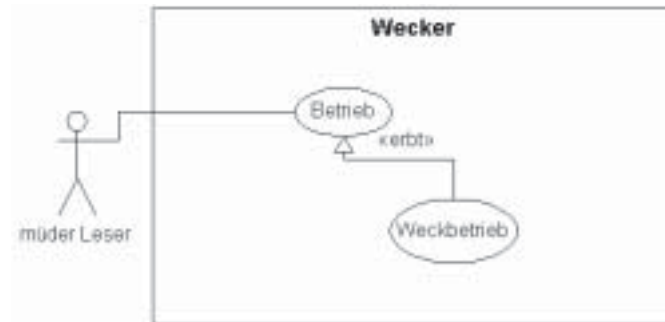


Abbildung 3.4: Use Case »Wecker«

Die Abbildung zeigt den Akteur müder Leser, der mit dem Anwendungsfall Betrieb verbunden ist. Der zweite abgebildete Anwendungsfall Weckbetrieb ist ein Spezialfall von Betrieb. Ersterer erbt also die Eigenschaften des Letzteren und ergänzt sie um eigene Eigenschaften.

Wie gesagt, das Beispiel ist nicht sehr aussagekräftig. Im Zusammenhang mit der eingehenden Beschreibung der weiteren Diagramme wird der Wecker Ihnen jedoch die Konzepte der UML anschaulich verdeutlichen.

Innensicht

Für die Kommunikation zwischen Anwender und Entwickler ist die Außensicht, die alle inneren Details vernachlässigt, Grundlage, um Elemente des Systems zu identifizieren. Betrachten wir ein System mit Hilfe von Anwendungsfällen, sehen wir Systemstrukturen und deren Abhängigkeiten, ohne zu wissen, welche Abläufe sich genau dahinter verbergen.

Um ein System zu verstehen und zu konstruieren, legen wir in der Innensicht fest, wie die bisher als black-box betrachteten Zusam-

menhänge im Inneren aussehen. Wir schauen in sie hinein und machen Sie damit zur white-box.

Die UML stellt, wie wir in Teil I dargestellt haben, eine Reihe von Diagrammen zur Verfügung, mit denen eine solche Innensicht abgebildet wird. In diesem Abschnitt wollen wir diese Sicht nicht vollständig modellieren. Wir greifen nur notwendige Elemente aus ausgewählten UML-Diagrammen heraus, die es aber dennoch ermöglichen, ein System in seinem Aufbau und seiner Funktionsweise aus fachlicher Sicht zu verstehen und abzubilden.

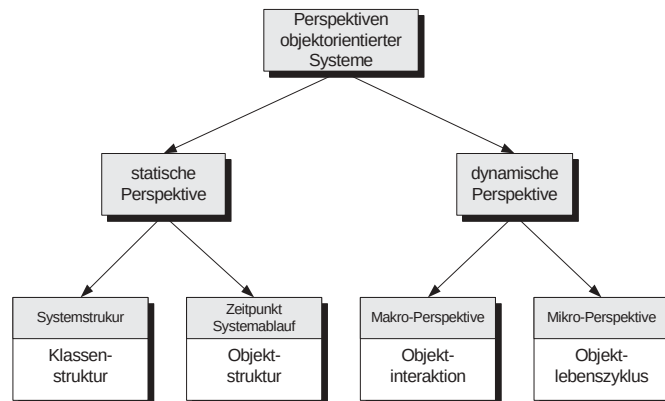
Die aufgabenorientierte Sicht der Anwendungsfälle dient als Aufhänger, um die inneren Strukturen in einem Lernprozess zu verstehen. Die Erfahrung zeigt, dass es sinnvoll ist, ein System aus verschiedenen Blickwinkeln zu betrachten, um es wirklich zu verstehen.

Jeder sieht's anders

Bereits im ersten Teil haben wir gesehen, dass Entwickler und Anwender bei ihrer subjektiven Betrachtungsweise, die von unterschiedlichen Betrachtungspositionen, aber auch von unterschiedlichem Denkverhalten stammen, Zusammenhänge häufig sehr unterschiedlich sehen.

Im Prozess der Systementwicklung geht es primär darum, diese verschiedenen Sichtweisen aufeinander abzustimmen, so dass ein Modell und darauf basierend Software entsteht, die von den Beteiligten akzeptiert wird.

Die UML bietet den verschiedenen Beteiligten unterschiedliche Diagramme an, die jeweils unterschiedliche formale Aspekte eines Modells betonen. Betrachten wir den Auftraggeber eines Systems, so wird der primär ökonomische Interessen am System und an der Außensicht besitzen. Ihm gibt das Anwendungsfall-Diagramm eine verständliche Gesamtübersicht.

**Abbildung 3.5:** White-Box-Perspektiven

Betrachten wir ein objektorientiertes System aus vier verschiedenen Blickrichtungen, die jeweils andere Aspekte desselben Systems hervorheben, hat unser Gehirn ausreichend Futter, um ein Verständnis der Zusammenhänge zu erzeugen.

Eine erste Unterscheidung der Perspektiven erfolgt nach zeitlichen Kriterien. Zunächst betrachtet die statische Sicht Systemzustände, in denen die Zeit nicht explizit Bestandteil des Modells ist. Auf der anderen Seite werden dynamische Aspekte eines objektorientierten Systems betrachtet, die Abläufe in Abhängigkeit von der Größe Zeit darstellen. In der statischen Perspektive unterscheidet sich ein Blick auf die zeitlich invariante Systemstruktur von einem Blick auf den Systemzustand zu einem bestimmten Zeitpunkt, der als Schnappschuss oder Momentaufnahme bezeichnet werden kann.

Während die Klassenstruktur die zeitlich invariante Systemstruktur darstellt, existiert zu jeder Zeit, in der das System aktiv ist, eine konkrete Zustandsstruktur aller bestehenden Objekte. Die dynamische Sicht unterscheidet zwei Granularitätsstufen der Betrachtung. Neben einer auf das Verhalten eines einzelnen Objekts beschränk-

ten Sicht, die durch den Begriff der dynamischen Mikro-Perspektive beschrieben werden kann, stellt die dynamische Makro-Perspektive das Zusammenspiel der Objekte auf der Systemebene dar.

Vom Groben zum Detail oder doch umgekehrt?

Vor jeder Überlegung stellen sich den Projektverantwortlichen einige grundsätzliche Fragen:

- ✓ *Was will ich modellieren? Welchen Kern hat mein System?* Hier wird festgelegt, ob beispielsweise ein Workflow-Management-System, ein Buchhaltungssystem, eine Lagerverwaltung o.Ä. Gegenstand des Projektes ist.
- ✓ *In welchem Umfang will ich modellieren?* Die Systemgrenzen müssen festgelegt werden. Bezogen auf die Unternehmung müssen Sie festlegen, ob die gesamte Unternehmung oder eine Organisationseinheit Sparte, Abteilung, Filiale den Umfang begrenzen. Grundsätzlich lässt sich die Forderung nach einem umfassenden, die Gesamtunternehmung abbildenden Modell der Geschäftsprozesse stellen. Eine solche Lösung ist häufig so komplex, dass sie kaum handhabbar ist. Der Projektansatz greift Inseln heraus (inhaltlich und/oder organisatorisch) und kommt dort sehr viel schneller wegen der geringeren Komplexität zu Modellierungsergebnissen. Schwierig ist in diesem Ansatz die Abstimmung mit anderen Projekten.
- ✓ *Wie tief will ich modellieren?* Die UML kann sowohl als kognitives Ordnungsinstrument, das zur Darstellung von Geschäftsprozessen eingesetzt wird, als auch weiter in die Tiefe gehend zur Darstellung eines fachlichen und technischen Entwurfs von Software-Systemen eingesetzt werden.

Greifen wir das einleitende Beispiel einer Bank noch einmal auf. Wollen wir alle Geschäftsaktivitäten der Bank beschreiben, oder geht es lediglich um Kreditgeschäfte oder um Finanztransaktionen auf Spar- und Girokonten? Sollen lediglich Geschäftsprozesse dargestellt werden oder eine Software zur Abwicklung der Finanztransaktionen realisiert werden?

Nachdem so Inhalt und Grenzen des Systems vereinbart sind, stellt sich die Frage nach der Vorgehensweise. In welchen konkreten Schritten bauen wir unser objektorientiertes System auf ?

Die UML gibt zunächst keine Information, welche Schritte in welcher Reihenfolge abzuarbeiten sind. Die Vorläufer der UML enthielten solche Vorgehensmodelle. Da die Amigos sich jedoch nicht auf ein gemeinsames Vorgehen einigen konnten, ist die UML mehr eine Sammlung von Abbildungstechniken als eine vollständige Methode, die auch den Weg der Modellierung vorgibt.

Die Tatsache, dass kein allgemeingültiges Vorgehensmodell in der UML spezifiziert ist, ist nicht als Nachlässigkeit der Autoren zu deuten. Vielmehr ist es so, dass jedes Projekt sein eigenes Vorgehen benötigt. Je nachdem, ob ein Entwicklungsteam nach einem strengen Plan Schritt für Schritt in einem deterministischen Vorgehen von der Planung über das Fachkonzept, den technischen Entwurf zur Implementierung schreitet oder den Entwicklungsprozess je nach Erfordernis in einem evolutionären Prozess flexibel anpasst und keine strengen Abfolgen vorgibt. Beide Vorgehensweisen sind mit der UML vereinbar.

Dementsprechend existieren unterschiedliche Vorgehensmodelle zur Gestaltung objektorientierter Systeme.

Mittlerweile gibt es von den Amigos den *Unified Process*, der ein mögliches Vorgehen definiert und von der Firma Rational im *RUP* (*Rational Unified Process*) auf die Modellierungswerkzeuge der Firma Rational angepasst wurde. Ein von der objektorientierten Modellierung unabhängiges Vorgehen beschreibt das V-Modell, das im öffentlichen Sektor eine weite Verbreitung gefunden hat.

Sind Systeminhalt und Systemgrenzen einmal festgelegt, stellt sich die Frage, an welchem Punkt wir beginnen. Gehen wir in unserem Beispiel davon aus, dass unsere Bankmitarbeiter durch ein neues Software-System unterstützt werden sollen, das die Finanztransaktionen von Spar- und Girokonten aller Kunden der betrachteten Bank abwickelt.

Grundsätzlich existieren zwei Herangehensweisen, um das Modell zu erstellen. Wir können von den groben Zusammenhängen ausgehen und dann das Gesamtsystem immer weiter verfeinern, bis wir eine Granularitätsstufe erreicht haben, die es uns ermöglicht, das Ganze in Software zu gießen. Dieses Vorgehen nennt man Top-Down-Strategie. Wir gehen von einem Finanztransaktionssystem aus und zerlegen es in die Kundenkonten, die Finanztransaktionen ermöglichen, die Liquiditätsüberwachung und die beteiligten Kunden und Sachbearbeiter mit ihren Aufgaben. Wir legen fest, dass jede Finanztransaktion von einem zuständigen Bankmitarbeiter genehmigt sein muss. Weiterhin hat jede Transaktion einen Einfluss auf die Liquidität unserer Bank. Im nächsten Schritt greifen wir dann die Kundenkonten heraus, von denen aus eine Finanztransaktion möglich ist. Dort spielen Zahlungsein- und -ausgänge die entscheidende Rolle. Betrachten wir ein Girokonto, so existiert dort auch die Möglichkeit zur Überweisung eines Betrages.

Der umgekehrte Weg wird im Bottom-Up-Vorgehen beschritten. Wir greifen Teilaspekte des Systems heraus, entwickeln isolierte Teilmodelle und fügen sie später zu einem Gesamtmodell zusammen. So könnten wir unabhängig voneinander die Transaktionen von Sparkonten und die von Girokonten als Modell entwickeln, die in einem zweiten Schritt in den Gesamtkontext der Finanztransaktionen eingeordnet werden, der dann auch die Liquiditätsreservenüberwachung der Bank mit einbezieht.

UML-Kerndiagramme im Überblick

Die vier grundlegenden Perspektiven stellen wir anhand eines Ausschnitts der Finanztransaktionen dar.

Zum grundlegenden Verständnis objektorientierter Systeme ist, ähnlich wie in der Systemtheorie, die Kenntnis nur weniger Konzepte und Begriffe notwendig. Das Verständnis technischer objektorientierter Systeme erfordert hingegen eine Erweiterung der elementaren objektorientierten Konzepte und Begriffe.

Da die erstellten Teilmodelle zumindest in der Analysephase vorrangig zur Kommunikation zwischen Entwicklern und Anwendern

eingesetzt werden, ist für das Verständnis nicht zwingend die Kenntnis der über die objektorientierten Grundkonzepte hinausgehenden technikorientierten Details notwendig.

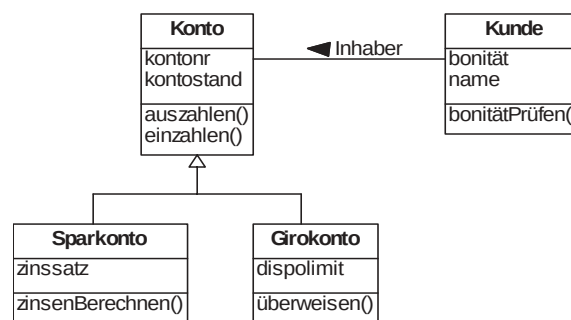


Abbildung 3.6: Klassendiagramm »Kontoanwendung«

Zunächst wird die *statische strukturorientierte Perspektive* modelliert, die die Abbildung in Form eines *Klassendiagramms* veranschaulicht. Klassen erscheinen wie bereits dargestellt als dreifach unterteilte Rechtecke, die von oben nach unten den Klassennamen, die Zustandsattribute und die Verhaltensweisen enthalten. Derartige Klassen zeigen die zeitlich invariante Struktur von Systemelementen auf. Beziehungen, die zwischen den Exemplaren bestimmter Klassen existieren, werden durch einfache Verbindungslinien dargestellt, Vererbungsbeziehungen als Pfeil, wobei der Pfeil von der erbenden auf die vererbende Klasse zeigt. In dem gewählten Beispiel stellt die Klasse **Konto** die vererbende Klasse dar und die Klassen **Sparkonto** und **Girokonto** verkörpern die erbenden Klassen.

Alle Definitionsbestandteile der Oberklasse **Konto** – Zustand und Verhalten – werden an die Unterklassen **Sparkonto** und **Girokonto** weitergegeben und erscheinen in den Unterklassen, als seien sie dort definiert.

Die einfache Verbindungslinie zwischen den Klassen **Konto** und **Kunde** symbolisiert eine Beziehung zwischen Exemplaren der Klas-

sen Konto und Kunde. Dabei besteht genau dann eine Beziehung zwischen einem Exemplar der Klasse Kunde und einem Exemplar der Klasse Konto, wenn der Kunde Inhaber eines Kontos ist.

Die dargestellte Klassenstruktur besitzt zu jedem Zeitpunkt der Existenz eines Objektsystems eine konkrete Ausprägung, wie die folgende Abbildung für das Kontenbeispiel als Ausschnitt mit sechs Exemplaren der Klassen Sparkonto, Girokonto und Kunde zeigt. Jedes Rechteck steht für ein Objekt, das als Exemplar einer der drei Klassen zugeordnet ist.

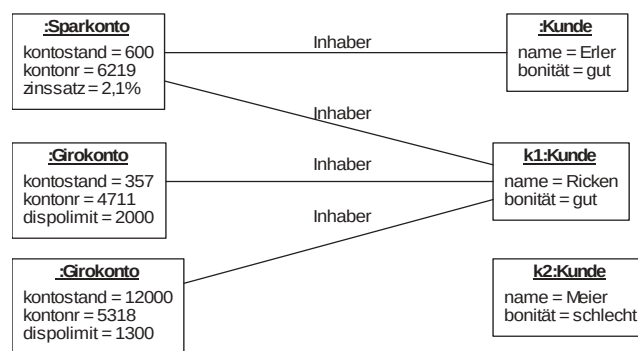


Abbildung 3.7: Objektdiagramm »Kontoanwendung«

Jedes Objekt ist eindeutig durch die Darstellung als Rechteck identifizierbar. Zusätzlich kann ein unterstrichener Objektname und darüber hinaus der zugehörige Klassenname durch einen Doppelpunkt von der Objektbezeichnung getrennt angegeben werden.

Die ebenfalls aus dem Klassendiagramm bekannte Beziehung zwischen den Objekten stellt auch hier Verbindungslinien her. Der zeitpunktabhängige Zustand manifestiert sich in den Attributen, die die Konten und Kunden-Objekte enthalten.

Während der Zustandsraum aus der zeitlich invarianten Klassenstruktur unveränderlich ist, können sich die konkreten Zustände durch Wertzuweisungen ändern. So könnte der Zinssatz des Sparkonto-Objekts 6219 zu einem anderen Zeitpunkt von 2,1% auf 2,4% erhöht werden. Die zeitliche Veränderung der Beziehungen zwischen Objekten hingegen beschränkt sich zunächst auf ihre Existenz oder Nicht-Existenz. Dabei müssen alle Beziehungen zwingend mit den Strukturvorgaben des Klassenstrukturdiagramms übereinstimmen.

Die im Objektdiagramm dargestellte Momentaufnahme im Systemablauf stellt neben anderen Objekten die Kunden Erler und Ricken durch eine Verbindungslinie als gemeinschaftliche Inhaber des Girokontos 6219 dar. Das Objektdiagramm zeigt also die in einem Klassenstruktur-Diagramm abgebildeten möglichen Beziehungsstrukturen in einer konkreten Ausprägung zu einem bestimmten Zeitpunkt als Beziehung zwischen Objekten.

Neben der statischen strukturorientierten Perspektive verdeutlichen die dynamischen Teilmodelle Systemabläufe. Die erste *dynamische Sicht* modelliert das *Objekt-Lebenszyklusmodell*. Das Kontenbeispiel in der folgenden Abbildung definiert zugelassene Zustände und Zustandsübergänge eines Objekts von seiner Entstehung (schwarzer Punkt) bis zu seiner Vernichtung (schwarz umrandeter schwarzer Punkt). Die Kästchen stellen Zustände dar, die mit bestimmten Aktionen verbunden sein können. Die Verbindungspfeile repräsentieren Zustandsübergänge, die durch bestimmte Ereignisse, die mit Bedingungen versehen sein können, ausgelöst werden. Ein Konto kann zwei verschiedene Zustände besitzen. Es kann einen positiven Kontostand aufweisen (oberes Kästchen) oder einen negativen Saldo (unteres Kästchen) enthalten (siehe Abbildung 3.8).

Der dargestellte Objekt-Lebenszyklus modelliert mögliche Veränderungen des Zustands eines Objekts.

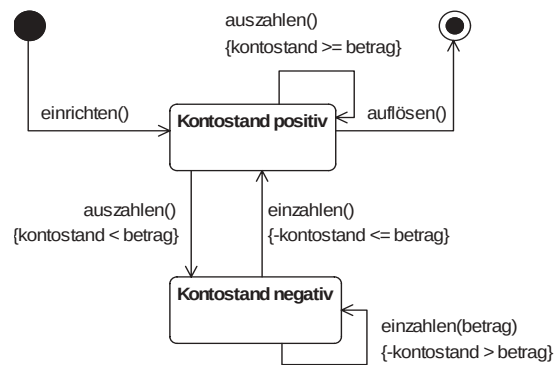


Abbildung 3.8: Zustandsdiagramm »Kontoanwendung«

Die zweite dynamische Perspektive fokussiert mögliche Interaktionen zwischen den relevanten Objekten des Anwendungsbereichs. Die folgende Abbildung stellt die *Objek-Interaktion* zwischen den Kontenarten Girokonto und Sparkonto in einem *Sequenz-Diagramm* dar (siehe Abbildung 3.9).

Gestrichelte senkrechte Linien repräsentieren die drei an der Interaktion beteiligten Objekte der Klassen Kunde, Girokonto und Sparkonto. Die Verbindungspfeile deuten eine Interaktion zwischen Objekten an, wobei die Interaktion ausgelöst wird durch das Objekt, das den Ursprung des Pfeils darstellt. Im Sinne einer Auftragsvergabe beauftragt das Objekt der Klasse Kunde das Objekt der Klasse Girokonto, eine Überweisung vorzunehmen, bei der ein bestimmter Betrag an ein bestimmtes Gegenkonto überwiesen wird. Die Durchführung der Überweisung obliegt dann dem Objekt der Klasse Girokonto selbst.

Das auslösende Objekt interagiert mit dem Objekt, das durch die Pfeilspitze gekennzeichnet wird. Die Zeitspanne, in der ein Objekt aktiv ist, wird durch einen Längsbalken über dem gestrichelten Objektsymbol abgebildet. Die Interaktionen erfolgen chronologisch von oben nach unten.

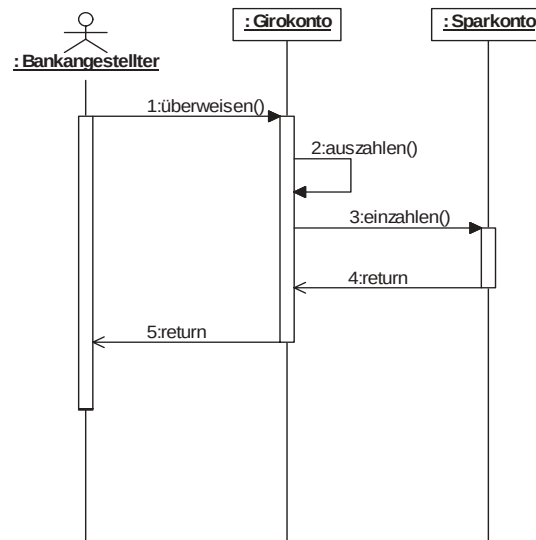


Abbildung 3.9: Sequenzdiagramm »Kontoanwendung«

An dieser Stelle diene die Behandlung einiger Teilmodelle der UML lediglich der Illustration der Perspektiven und stellt keinesfalls eine erschöpfende Behandlung dar.

Der objektorientierte Ansatz ist bis hierher in seiner historischen Entstehung sowie aus konzeptioneller Sicht dargestellt.



KAPITEL

Diagrammarten

Im vierten Kapitel wollen wir uns die wichtigsten Diagramme etwas genauer ansehen. Anhand konkreter Beispiele stellen wir Ihnen die verschiedenen Diagramme konzeptionell im statischen und im dynamischen Objektmodell vor und zeigen Ihnen deren Anwendung.

4



Diagrammarten

In den vorangehenden Kapiteln haben Sie bereits die statischen und die dynamischen Eigenschaften von Objekten kennen gelernt. Für jede dieser beiden Arten von Eigenschaften stellt die UML verschiedene Diagramme zur Verfügung, mit denen diese Eigenschaften näher untersucht und beschrieben werden können.



Neben der Außen- und der Innensicht lassen sich die Diagramme der UML auch in eine statische und eine dynamische Sicht unterscheiden.

Im Rahmen der Beschreibung der Spracharchitektur der UML haben wir diese beiden Sichtweisen bereits kurz angesprochen:

- ✓ *Statische Modelle* (Strukturmodelle) beleuchten die Struktur von Objekten in einem System. Dazu gehören ihre Klassen, Schnittstellen, Attribute und Beziehungen zu anderen Objekten.
- ✓ *Dynamische Modelle* (Verhaltensmodelle) zeigen das Verhalten der Objekte in einem System. Hierzu zählen ihre Methoden, Zustände und Interaktionen sowie ihre Zusammenarbeit bei der Lösung einer Problemstellung.

In den folgenden Abschnitten werden die UML-Diagramme der statischen und der dynamischen Modelle näher erläutert.

Statik: Klassen- und Objektdiagramme

Klassendiagramme beschreiben die statische Struktur eines objektorientierten Modells, die über einen längeren Zeitraum unverändertlich bleibt. Dabei stehen der Aufbau und die Beziehungen von Objekten im Vordergrund der Betrachtung.

Klasse – Bauplan für Exemplare

Über den Aufbau einer Klasse haben Sie bereits mit den Grundlagen der Objektorientierung sowie bei der Einführung in die UML einiges erfahren. Eine Klasse beschreibt den Aufbau von Objekten, indem sie Name, Zustand und Verhalten für Objekte festlegt.

Eine Klasse beschreibt die Gemeinsamkeiten von Objekten, deren Zustand den gleichen Aufbau besitzt und deren Verhaltensweisen identisch sind.

Das bedeutet konkret, dass die Attribute, nicht aber die Attributwerte, sowie die Operationen von Objekten, die in einer gemeinsamen Klasse beschrieben werden, identisch sind.



Stellen Sie sich eine Klasse als eine Art Schablone vor, die den Aufbau zugehöriger Objekte beschreibt. Sie ist somit ein Bauplan, nach dem ein Objekt gefertigt wird. Dabei legt die Klasse den Zustandsraum (Summe der Attribute) sowie sein Verhalten (Summe der Operationen) fest.

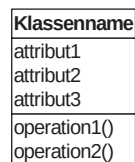


Abbildung 4.1: Klassenstruktur UML-Darstellung

In der UML symbolisiert ein Rechteck eine einfache Klasse. Die Abbildung zeigt, dass die grafische Notation aus drei Segmenten besteht.

- ✓ Das erste Segment beinhaltet den Namen der Klasse. Über diesen Namen ist die Klasse eindeutig identifizierbar. Das bedeutet, dass es in einem objektorientierten Modell niemals zwei

Klassen mit einem identischen Namen gibt. Wir werden später sehen, dass Klassen zu Paketen zusammengefasst werden können. Somit können Klassen in verschiedenen Paketen gleiche Namen besitzen. Ihre Identität kann über die Paketzugehörigkeit dann eindeutig festgestellt werden.

- ✓ Das zweite Segment nimmt die Attribute einer Klasse auf. Sie beschreiben den Zustand (statische Eigenschaften) der aus ihr abgeleiteten Objekte. Aus einer Klasse abgeleitete Objekte bezeichnet man als Exemplare.
- ✓ Im dritten Segment definieren wir die Operationen, die das Verhalten der Exemplare festlegen. Sie legen das dynamische Verhalten der abgeleiteten Objekte fest.
- ✓ Die Segmente zwei (Attribute) und/oder drei (Operationen) können auch ausgeblendet werden, so dass eine Klasse nur mit ihrem Namenssegment angezeigt wird.

Ein Beispiel für eine solche Schablone ist die Klasse `Fahrzeug`, die als konzeptionelle Beschreibung im Zustands-Segment die Attribute `besitzer`, `typ`, `farbe` und `geschwindigkeit` enthält und im Verhaltens-Segment die Operationen `fahren()` und `bremsen()` vereinbart.

Fahrzeug
besitzer
typ
farbe
geschwindigkeit
fahren()
bremsen()

Abbildung 4.2: Klasse `Fahrzeug`: UML-Darstellung

Ein weiteres Beispiel ist die konzeptionelle Beschreibung eines Weckers in einer Klasse.

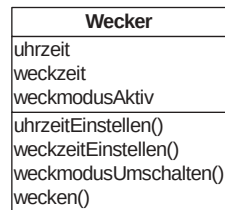


Abbildung 4.3: Klasse Wecker: UML-Darstellung

- ✓ Die Klasse Wecker enthält als Attribute die `uhrzeit`, die `weckzeit` und `weckmodusAktiv`.
- ✓ Die aktuelle Zeit der Attribute `uhrzeit` und `weckzeit` kann über die Operationen `zeitEinstellen()` und `weckzeitEinstellen()` manipuliert werden.
- ✓ Die Operation `weckmodusUmschalten()` dient zur Festlegung, ob der Wecker zur eingestellten Weckzeit klingeln soll oder nicht. Der Aktivierungsstatus wird im Attribut `weckmodusAktiv` festgehalten.
- ✓ Die Operation `wecken()` wird aufgerufen, wenn `weckmodusAktiv` aktiv geschaltet ist und die `weckzeit` erreicht ist. Mit anderen Worten, der Wecker klingelt.



Beachten Sie, dass die Attribute `uhrzeit`, `weckzeit` und `weckmodusAktiv` in der Klassendarstellung, die das Konzept Wecker beschreibt, keine Attributwerte enthalten. Sie beschreiben den Zustandsraum, also den strukturellen Aufbau des Zustands, ohne konkrete Werte für die Attribute festzulegen. Erst wenn aus der Klasse ein Objekt erzeugt wird, erhält jedes Attribut in diesem neu erzeugten Exemplar einen konkreten Wert.

Die UML schreibt einige formale Kriterien fest, die unerlässlich sind, um allgemein verständliche und zueinander kompatible objektorientierte Modelle erstellen zu können. Darüber hinaus exis-

tieren Konventionen, die von der Gemeinschaft der Modellierer akzeptiert sind:

- ✓ Der Klassenname steht zentriert und fett im ersten Segment des Klassensymbols.
- ✓ Klassennamen beginnen mit einem Großbuchstaben.
- ✓ Attribute und Operationen werden mager gedruckt.
- ✓ Die Bezeichnungen für Attribute und Operationen beginnen mit einem Kleinbuchstaben.
- ✓ Operationen unterscheiden sich von Attributen dadurch, dass sie hinter ihrem Namen eine öffnende und eine schließende runde Klammer besitzen. Innerhalb der Klammern können weitere Angaben zur Operation stehen.
- ✓ Operationen werden mit Verben bezeichnet, um den aktiven Charakter zu unterstreichen.
- ✓ Die Namen abstrakter Klassen werden kursiv gedruckt.
- ✓ Bei Attributen oder Operationen, deren Namen sich aus mehreren Wörtern zusammensetzen, schreiben Sie die Wörter ohne Leerzeichen hintereinander, wobei jedes neue Wort mit einem Großbuchstaben beginnt. So bezeichnen Sie ein Attribut, das die Anzahl der Räder eines Fahrzeugs bezeichnet, mit `anzahlRäder`, oder die Operation zum Einstellen der Uhrzeit in der Klasse `Wecker` als `uhrzeitEinstellen()`.

Objekt – das konkrete Exemplar einer Klasse

Eine Klasse wird immer mit dem Ziel modelliert, direkt oder indirekt Vorlage oder Schablone für Objekte zu sein, die aus ihrer Strukturbeschreibung erzeugt werden (siehe Abbildung 4.4).

Ein Objekt besitzt ebenfalls ein rechteckiges Symbol. Es unterscheidet sich vom Klassensymbol dadurch, dass der Objektname im ersten Segment unterstrichen ist.

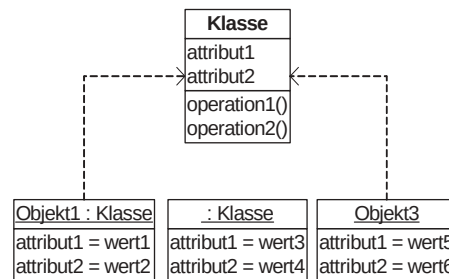


Abbildung 4.4: Aus Klassen erzeugte Objekte: UML-Darstellung

Die einem Objekt zugeordnete Klasse kann auf zwei verschiedene Arten kenntlich gemacht werden. Die erste Möglichkeit besteht in einem gestrichelten Pfeil, der vom Objekt ausgehend auf die Klasse verweist. Die zweite Alternative ist die Nennung des Klassennamens im ersten Segment des Objektsymbols. Dabei wird der Klassenname unabhängig davon, ob das Objekt benannt ist, von einem Doppelpunkt eingeleitet.

Den gestrichelten Pfeil oder alternativ die Nennung des Klassennamens bezeichnet man als *Realize*-Beziehung, da das Objekt als konkrete Ausprägung die Struktur der Klasse realisiert.

Im zweiten Segment legt ein Objekt zu den in der Klasse definierten Attributen Attributwerte in der Form `attribut = wert` fest.

So dient die Klasse *Fahrzeug* dazu, konkrete Fahrzeuge zu beschreiben, wie meinen roten Golf `meinGolf` oder den silberfarbenen Mercedes (hier als nicht benanntes Objekt dargestellt) (siehe Abbildung 4.5).

Den Attributen `besitzer`, `typ`, `farbe` und `geschwindigkeit` werden konkrete Werte, die in jedem Objekt verschieden sein können, zugeordnet.

Die Klasse *Wecker* dient als Schablone für alle Wecker dazu, einen alten schwarzen Wecker, `meinSchwarzerWecker`, zu beschreiben.

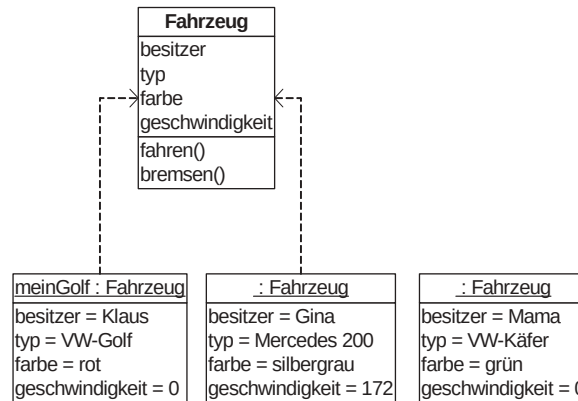


Abbildung 4.5: Aus Klassen erzeugte Objekte (Exemplare): Fahrzeuge

Auch hier finden wir die in der Klasse definierten Attribute `uhrzeit`, `weckzeit` und `weckmodusAktiv` in jedem aus der Klasse abgeleiteten Objekt mit den zugeordneten konkreten Werten wieder.

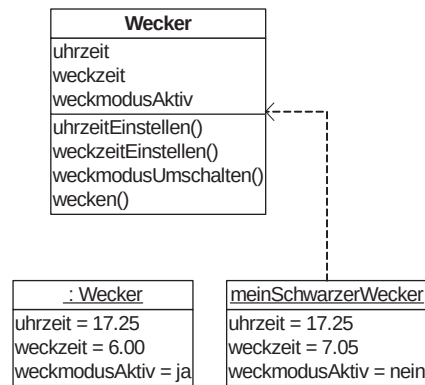


Abbildung 4.6: Aus Klassen erzeugte Objekte (Exemplare): Wecker

Die Operationen werden nicht genannt, da sie vollständig innerhalb der Klasse definiert und daher für alle Objekte identisch sind. Sie können sich daher in der zugehörigen Klasse über die verfügbaren Operationen eines Objektes informieren.



Klassen definieren ihre Attribute und Operationen, so dass die aus diesen Klassen erzeugten Objekte neben ihrer Klassenzugehörigkeit nur noch die konkreten Attributwerte festlegen. Die Operationen sind vollständig in der Klasse vereinbart und dort einsehbar.

Neben den gewöhnlichen Klassen existieren spezielle Formen von Klassen.

Schnittstellen (*interfaces*) beschreiben nur das von außen sichtbare Element einer Klasse und müssen konkretisiert werden. Weiter unten werden Schnittstellen konzeptionell erläutert und im nächsten Teil in einem Beispiel umgesetzt.

Im Kontext der Vererbung greifen wir so genannte *abstrakte Klassen* auf, aus denen keine Objekte erzeugt werden können.

Parametrisierbare Klassen (*Template, Parameterized Class*) sind Schablonen, die generische Parameter enthalten. Aus diesen generischen Klassen können gewöhnliche Klassen erzeugt werden, die dann die generischen Parameter durch aktuelle Parameter konkretisieren. Die Sprache C++ unterstützt solche Templates.

Metaklassen hingegen sind Klassen, deren erzeugte Objekte auch wieder Klassen sind. Metaklassen sind ein theoretisch hoch interessantes Thema, da sie menschliches Denken gut nachbilden können. So können selbstreferenzielle oder autopoietische Systeme nachempfunden werden. In der praktischen Modellierung mit UML spielen sie meist eine untergeordnete Rolle.

Attribut – identitätslose Eigenschaft

Attribute legen den Zustandsraum einer Klasse fest. Sie definieren eine beliebige Anzahl beschreibender Eigenschaften und Beziehun-

gen, die die Klasse und damit auch die abgeleiteten Objekte näher beschreiben.

Attribute einer Klasse besitzen keinen konkreten Wert. Wohl aber können Sie für Attribute einen bestimmten Wertebereich festlegen. Diese Festlegung entspricht im Wesentlichen der Festlegung eines Datentyps in einer Programmiersprache.

Grundsätzlich lassen sich zwei verschiedene Arten von Attributen unterscheiden. Zum einen gibt es beschreibende Attribute, die selbst keine Identität besitzen. Sie sind jedoch einem Objekt, das eine Identität besitzt, zugeordnet und beschreiben es näher. Das Attribut *farbe* eines Autos mit dem Attributwert *rot* hat keine Identität, beschreibt jedoch den roten Golf, der als Objekt eine Identität besitzt, näher. Solche Attribute lassen sich mit den klassischen Datentypen *Integer* (Ganzzahl), *Real* (Fließkommazahl), *String* (Zeichenfolge) oder *Boolean* (Wahrheitswert) auf einen bestimmten Wertebereich einschränken.

Typ	Beispielwerte	Operationen
Integer	1, -3, 234, 1567, ...	*, +, -, /
Real	2.5, -3.265, 500.0, ...	*, +, -, /
String	"Meier", "Chiron", ...	toUpper(), concat()
Boolean	true, false	and, or, xor, not

Tabelle 4.1: Basisdatentypen

Die Tabelle zeigt die Basisdatentypen mit einigen Beispielwerten und wichtigen Operationen, die mit diesen Typen durchgeführt werden können. Sie sind in der *OCL* (*Object Constraint Language*), die zur Beschreibung von Zusicherungen dient, festgelegt. So stehen bei den Zahlen mathematische Operationen zur Verfügung, bei den Strings Operationen zur Umwandlung von Klein- in Großbuchstaben und Verkettung und bei Wahrheitswerten logische Operatoren.

Die zweite Art von Attributen sind Attribute, die wie das Objekt, das sie als Attribut beschreiben, ebenfalls eine Identität besitzen. Somit ist auch das Attribut ein Objekt, das zur näheren Beschreibung eines anderen Attributes dient. Objekte, die zu ihrer eigenen Beschreibung fremde Objekte nutzen werden wir im Zusammenhang der Assoziationen noch genauer unter die Lupe nehmen.

Betrachten wir die Klasse Fahrzeug, so können wir die Attribute besitzer, typ und farbe als Zeichenfolgen (String) und die geschwindigkeit als ganzzahligen Wert (Integer) festlegen. Sobald ein Objekt mit konkreten Werten belegt wird, müssen diese Werte dem vereinbarten Datentyp formal entsprechen.

Fahrzeug
besitzer : String
typ : String
farbe : String
geschwindigkeit : Integer = 0
fahren()
bremsen()

Abbildung 4.7: Datentypen für Attribute: Fahrzeug

Zusätzlich können so genannte Initialwerte festgelegt werden. Im Fall des Attributs geschwindigkeit wird der Initialwert mit dem Wert 0 festgelegt. Das bedeutet, dass, sobald ein Objekt aus dieser Klasse abgeleitet wird, der Wert des Attributes geschwindigkeit mit 0 definiert wird.

Darüber hinaus können für Attribute, wie für viele andere Elemente in der UML, aber auch Zusicherungen, so genannte *Constraints* vereinbart werden. Constraints beschreiben Bedingungen, die immer erfüllt sein müssen. Dazu steht in UML die bereits erwähnte OCL zur Verfügung.



Die OCL wird in vielen Projekten durch Ausdrücke einer objektorientierten Programmiersprache wie C++, Java oder C# ersetzt. Oft werden auch frei formulierte Zusicherungen definiert.

Fahrzeug
besitzer : String
typ : String
farbe : String
geschwindigkeit : Integer = 0
anzahlRäder {>0} : Integer
fahren()
bremsen()

Abbildung 4.8: Zusicherung: Attribut Räder

Zusicherungen werden innerhalb geschweifter Klammern hinter dem Attributnamen notiert. So legt das neu definierte Attribut `anzahlRäder {>0} : Integer` fest, dass die Anzahl der Räder (`anzahlRäder`) als ganzzahliger Wert (`: Integer`) immer größer als der Wert 0 (`{>0}`) sein muss. Diese Bedingung muss zu jeder Zeit der Existenz eines Auto-Objektes erfüllt sein. Weitere Zusicherungen stellen wir Ihnen im Abschnitt über Assoziationen vor.

Operationen – Verhalten eines Objektes

Neben den Attributen unterliegen auch die Operationen einer formalen Darstellung. Wir haben gesehen, dass Attribute innerhalb eines vorgegebenen Wertebereichs für jedes Objekt einen beliebigen Wert annehmen können. Auch die Operationen können variabel ausgestattet werden, indem variable Parameter definiert werden können. Zusätzlich kann die Operation nach Ausführung einen Wert als Antwort zurückliefern. Formal sieht das folgendermaßen aus.

Klassenname
attribut1
attribut2
attribut3
operation1() : RückTyp1
operation2(Par1 : Typ1, Par2 : Typ2) : RückTyp2

Abbildung 4.9: Operationen: UML-Darstellung

Die Spezifikation einer Operation `operation(Par1 : Typ1, Par2 : Typ2) : Rück Typ3` zeigt ähnlich einer Funktion oder Prozedur, die als Operation an die aus dieser Klasse erzeugten Objekte gebunden ist, die Form der Übergabe- und Rückgabeparameter.

Innerhalb der runden Klammern (`Par1 : Typ1, Par2 : Typ2`) definieren Sie eine beliebige Anzahl von Parametern, die mit einem Namen gefolgt von einer Typangabe nach einem Semikolon `Par1 : Typ1` vereinbart sind. Legen Sie mehrere Übergabeparameter fest, so werden die Vereinbarungen durch ein Komma getrennt.

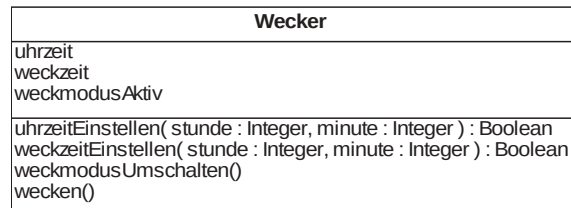
Neben den Übergabeparametern, die eine Operation beim Aufruf erhält, können Sie auch einen Rückgabeparameter vereinbaren, der ein Ergebnis des Operationsaufrufs zurückliefert. Den Typ des Rückgabeparameters `operation(...) : RückTyp3` legen Sie im Anschluss an den Bereich der Übergabeparameter getrennt durch einen Doppelpunkt fest.



Definieren Sie Attributtypen und Operationsparameter nicht wahllos, sondern denken Sie dabei vom Ergebnis her. In der Analysephase, in der es um die Darstellung der fachlichen Inhalte geht, stehen Datentypen und Operationsparameter der Übersichtlichkeit oft im Wege. Nutzen Sie ein Klassendiagramm, um mit den Fachexperten die fachlichen Inhalte zu visualisieren, reichen in der Regel Attributnamen und Operationsnamen. In der Entwurfsphase dagegen, die als technische Spezifikation die Implementierung vorbereitet, sind sie sehr hilfreich, da sie eine automatisierte Überführung des UML-Modells in eine Programmiersprache erleichtern.

Das folgende Beispiel definiert Übergabe- und Rückgabeparameter für die der Klasse `Wecker` zugeordneten Objekte (siehe Abbildung 4.10).

So müssen wir dem Objekt vom Typ `Wecker` beim Stellen der Uhrzeit oder der Weckzeit die neuen Werte für die Stunde und die Minute ...(`stunde : Integer, minute : Integer`) mitteilen. Beide Übergabeparameter sind vom Typ `Integer`, können also ganze Zahlen an die Operation übergeben.

**Abbildung 4.10:** Spezifikation von Operationen: Wecker

Die Operationen geben als Antwort einen Wert vom Typ ...() : Boolean zurück, der anzeigt, ob die Zeit erfolgreich umgestellt wurde (true) oder nicht (false).



Einige Tools zur UML-Modellierung erlauben die Festlegung von Attributen und Operationen direkt in der Syntax einer objektorientierten Programmiersprache wie Java, C++ oder C#. Dadurch wird der nahtlose Übergang vom Entwurf mit UML in eine Programmiersprache erleichtert.

Sichtbarkeit – wer sieht was von wo?

Grundsätzlich gilt bei einer strengen Auslegung der objektorientierten Prinzipien die Regel, dass auf die Attribute einer Klasse von außen ausschließlich indirekt über die Operationen zugegriffen werden darf. So rufen wir die Operation `uhrzeitEinstellen(...)` auf, die dann eine Wertänderung des Attributs `uhrzeit` vornimmt. Folglich greifen wir indirekt über die Operation `uhrzeitEinstellen(...)` auf das Attribut `uhrzeit` zu.

Die UML gibt dem Programmierer die Möglichkeit, die äußere Sichtbarkeit von Attributen und Operationen selbst zu bestimmen. Damit befindet sich die UML in einem Boot mit den objektorientierten Programmiersprachen Java, C++ und C#. Aus Effizienzgründen ist es in manchen Fällen angebracht, das Geheimnisprin-

zip, das aus konzeptionellen Gründen sinnvoll ist, im konkreten Entwurf aufzuweichen.



Bei der Definition von Sichtbarkeiten sollte grundsätzlich das Geheimnisprinzip Anwendung finden. Nur für begründete Ausnahmefälle, wie die Komplexität und Aufblähung des Codes oder bei Effizienzproblemen, die gut begründet sein müssen, sollten Sie Attribute nach außen sichtbar machen.

Die UML definiert vier verschiedene Sichtbarkeitsstufen.

Sichtbarkeit	Bedeutung	UML-Symbol
public	öffentlich; für alle Klassen sichtbar und benutzbar	+
package	paketweit; für die Klasse selbst sowie für alle Klassen desselben Paketes sichtbar und benutzbar	~
protected	geschützt; für die Klasse selbst sowie für alle Unterklassen sichtbar und benutzbar	#
private	privat; nur für die Klasse selbst sichtbar und benutzbar	-

Tabelle 4.2: Sichtbarkeiten von Attributen und Operationen

Beachten Sie, dass die Sichtbarkeit in den einzelnen Programmiersprachen abweichend von der UML-Spezifikation festgelegt sein kann, auch wenn dort dieselben Sichtbarkeitsbezeichnungen verwendet werden.



Für die Sichtbarkeiten gilt ebenso wie für die Vereinbarung von Typen, dass sie im Analysemodell eher stören, während sie den automatisierten Übergang vom UML-Modell zum Programmcode erleichtern.

Führen wir die Sichtbarkeiten für die Attribute und Operationen der Klasse `Fahrzeug` ein und legen fest, dass das Attribut `besitzer` als *private* (-) definiert und damit ausschließlich innerhalb von `Fahrzeug`-Objekten verfügbar ist.

Fahrzeug
-besitzer : String
#typ : String
+farbe : String
#geschwindigkeit : int = 0
+anzahlRäder {>0} : int
+fahren()
+bremsen()

Abbildung 4.11: Sichtbarkeiten: Fahrzeug

Die Attribute `typ` und `geschwindigkeit` sind als *protected* (#) vereinbarte Attribute auch in möglichen Unterklassen von Fahrzeugen wie beispielsweise PKW oder Motorrad sichtbar.

Die Attribute `farbe` und `anzahlRäder` sind als *public* (+) vereinbarte Attribute ebenso wie die Operationen `fahren()` und `bremsen()` uneingeschränkt verfügbar.

Klassenattribute und Klassenoperationen – Arbeiten ohne Objekte

Gewöhnliche Attribute und Operationen werden in einer Klasse vereinbart und stehen nach der Erzeugung den gebildeten Objekten zur Verfügung. Die Attribute und Operationen sind an ein Objekt gebunden und nur über diese verfügbar.

Für Attribute gilt, dass jedes aus einer Klasse erzeugte Exemplar seine individuellen Attributwerte festlegen kann. So hat das Attribut `Farbe` eines PKW für jeden PKW einen individuellen Wert. Es macht auch nur Sinn, über das jeweilige PKW-Objekt auf das `Farbe`-Attribut zuzugreifen.

Wird dagegen ein Attribut als Klassenattribut definiert, so ist dieses Attribut über die Klasse erreichbar. Es ist an die Klasse und nicht etwa an die aus ihr erzeugten Objekte gebunden. Sie stehen zwar jedem Objekt der Klasse zur Verfügung, besitzen aber, da sie an die Klasse gebunden sind, nur einen einzigen Wert, der damit für alle Objekte identisch ist.

Klassenoperationen und Klassenattribute werden in der UML-Notation wie gewöhnliche Attribute und Operationen in den üblichen Segmenten des Klassenrechtecks notiert. Zur Unterscheidung werden sie jedoch unterstrichen dargestellt.

Die folgende Abbildung zeigt am Beispiel des Weckers, wie ein Klassenattribut und eine Klassenoperation vereinbart und genutzt werden können.

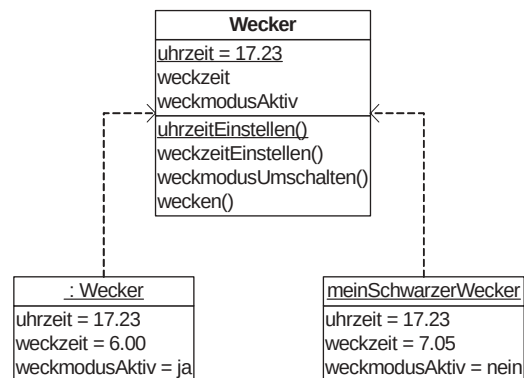


Abbildung 4.12: Klassenattribut und Klassenoperation: Wecker

- ✓ Die Klasse Wecker vereinbart ein Klassenattribut `uhrzeit`, das unterstrichen dargestellt ist.
- ✓ Damit ist das Attribut `uhrzeit` an die Klasse Wecker gebunden und über diese auch dann verfügbar, wenn noch kein Wecker-Objekt erzeugt ist.

- ✓ Das Klassenattribut `uhrzeit` ist für die Objekte `:Wecker` und `meinSchwarzerWecker` wie die gewöhnlichen Attribute verfügbar.
- ✓ Im Unterschied zu gewöhnlichen Attributen wie die `weckzeit`, die individuell bei jedem `Wecker` unterschiedlich sein kann, wirkt sich die Änderung der `uhrzeit` auf alle aus der Klasse `Wecker` abgeleiteten Objekte gleichermaßen aus.
- ✓ Die Operation `uhrzeitStellen()` ist als Klassenoperation definiert und zur Kennzeichnung im Operations-Segment der Klasse `Wecker` unterstrichen dargestellt.
- ✓ Die Operation `uhrzeitStellen()` kann unabhängig von der Existenz abgeleiteter Objekte über die Klasse `Wecker` aufgerufen werden, um den Wert des Klassenattributes `uhrzeit` zu verändern.
- ✓ Die Operation `uhrzeitStellen()` lässt sich ebenso über die aus der Klasse `Wecker` erzeugten Objekte `:Wecker` und `meinSchwarzerWecker` aktivieren.

Somit ist gewährleistet, dass alle Uhren dieselbe Uhrzeit besitzen und die Uhrzeit auch gestellt werden kann, wenn im System noch kein `Wecker`-Objekt erzeugt wurde, da die Uhrzeit in der Klasse hinterlegt ist.



Verwenden Sie Klassenattribute, wenn ein Attributwert für alle aus der Klasse gebildeten Objekte identisch ist und auch unabhängig von der Existenz von Exemplaren sinnvoll mit Werten belegt werden kann. Vereinbaren Sie Klassenoperationen für Aufgaben, die unabhängig von der Existenz von Objekten durchgeführt werden können. Eine sinnvolle Aufgabe ist beispielsweise die Manipulation von Klassenattributen.

Schnittstelle – verkümmerte Klassen

Schnittstellen (Interfaces) lassen sich salopp als verkümmerte Klassen beschreiben, da sie lediglich einen Schnittstellennamen und

die Aufrufkonventionen von Operationen definieren. Die Arbeitsweise der Operationen sowie Attribute vereinbart die Schnittstelle im Gegensatz zur Klasse nicht.



Wir kommen im nächsten Teil bei der Implementierung eines UML-Modells auf die Schnittstellen zurück und werden dort verdeutlichen, dass Schnittstellen als Ersatz für die Mehrfachvererbung eine wichtige Rolle spielen.

Die UML stellt zwei verschiedene Notationsformen für Schnittstellen zur Verfügung: die Lollipop-Darstellung und die klassenähnliche Notation.

Bei der Lollipop-Notation wird die Schnittstelle als nicht ausgefüllter Kreis dargestellt, der mit den Klassen, die die Schnittstelle implementieren, über eine durchgezogene Linie verbunden ist.

Das Implementieren einer Schnittstelle bedeutet, dass die Klasse sich verpflichtet, die »verkümmerten« Operationen zu übernehmen und vollständig mit der Definition der Aufgabenlösung zu beschreiben.

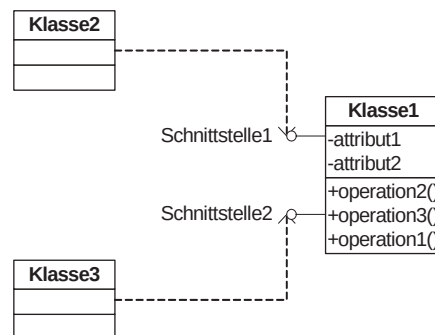


Abbildung 4.13: Schnittstellen: UML-Darstellung (Lollipop)

- ✓ Schnittstelle1 und Schnittstelle2 vereinbaren die Aufrufkonventionen für operation1(), operation2() und operation3(). Diese Vereinbarung ist in der Lollipop-Darstellung nicht sichtbar.
- ✓ Klasse1 implementiert beide Schnittstellen und verpflichtet sich damit, operation1(), operation2() und operation3() zu übernehmen und vollständig zu definieren.
- ✓ Klasse2 und Klasse3 stehen in einer Abhängigkeitsbeziehung zur Schnittstelle1, da sie Objekte des Schnittstellentyps nutzt. Die durch die gestrichelten Pfeile dargestellte Abhängigkeit besagt, dass eine Änderung von Schnittstelle1 eine Änderung in Schnittstelle2 nach sich ziehen könnte und eine Änderung in Schnittstelle2 sich auf Klasse3 auswirkt.

Alternativ kann auch die klassenähnliche Darstellung von Schnittstellen gewählt werden. Eine Schnittstelle ist mit dem Stereotyp <<interface>> gekennzeichnet und enthält keine Attribute.



Ein Stereotyp kann für viele UML-Elemente definiert werden und legt Erweiterungen bestehender Elemente fest oder dient einfach nur zur Klassifikation von Elementen. Im Fall der Schnittstelle definiert das Stereotyp die Erweiterungen (Operationen nur mit ihren Aufrufkonventionen zu beschreiben und Verbot der Definition von Attributen) im Vergleich zu einer gewöhnlichen Klasse.

In der klassenähnlichen Darstellung von Schnittstellen sind auch die vereinbarten Aufrufkonventionen der Operationen erkennbar (siehe Abbildung 4.14).

Klassen, die eine Schnittstelle implementieren, die klassenähnlich dargestellt wird, verweisen über einen gestrichelten Pfeil, der eine dreieckige nicht ausgefüllte Spitze besitzt, auf die zugehörige Schnittstelle.

- ✓ Schnittstelle1 vereinbart die Aufrufkonventionen von operation1() und operation2().

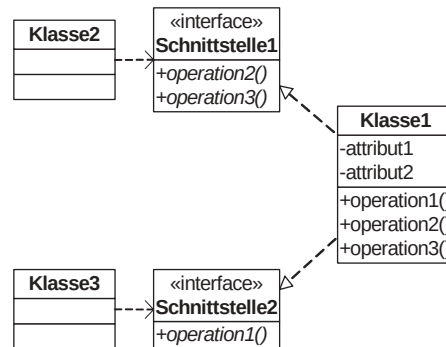


Abbildung 4.14: Schnittstellen: UML-Darstellung (klassenähnlich)

- ✓ Schnittstelle2 legt die Aufrufkonvention für operation1() fest.
- ✓ Da Klasse1 beide Schnittstellen implementiert, definiert sie alle drei Operationen vollständig.

Das Konzept von Schnittstellen wird deutlich im Kontext der Implementierung eines UML-Modells.

Vererbung – verteilte Baupläne

Das bereits in den Grundlagen vorgestellte Konzept der Vererbung vereinbart eine Weitergabe der Attribute und Operationen von einer Ober- an eine Unterklasse.



Die Beziehung zwischen Ober- und Unterklasse kann als Spezialisierung oder Generalisierung aufgefasst werden. Diese Sicht wird häufig in der fachlichen Analyse eingenommen. Aus technischer Sicht kann die Vererbung aber auch als Erweiterungsmechanismus genutzt werden, der semantisch mit einer Spezialisierung

lisierung nichts zu tun hat. Demnach werden Klassenhierarchien nicht nach kognitiven Assoziationen, sondern nach konstruktiver Effizienz gebildet.

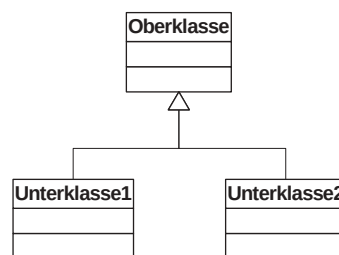


Abbildung 4.15: Vererbung: UML-Darstellung

Die UML stellt eine solche Vererbungsbeziehung als Pfeil dar, der von der Unterklasse auf die Oberklasse zeigt. Die Vererbungsrichtung läuft also entgegengesetzt zur Pfeilrichtung.

Betrachten wir die Vererbung mit den entsprechenden Attributen und Operationen, so werden die Attribute `attribut1`, `attribut2` genauso wie die Operationen `operation1` und `operation2` an die Unterklassen `Unterklasse1` und `Unterklasse2` weitergegeben (siehe Abbildung 4.16).

Die Unterklassen `Unterklasse1` und `Unterklasse2` definieren zusätzliche Attribute und Operationen. So enthält die `Unterklasse1` neben den geerbten Attributen `attribut1` und `attribut2` das Attribut `attribut3` und über die geerbten Operationen `operation1` und `operation2` die Operation `operation3`. Analog vollzieht sich die Vererbung in `unterklasse2`.

Betrachten wir das Objekt `Objekt1`, so zeigt sich, dass die von Oberklasse geerbten und die in `Unterklasse1` neu definierten Attribute nicht zu unterscheiden sind.

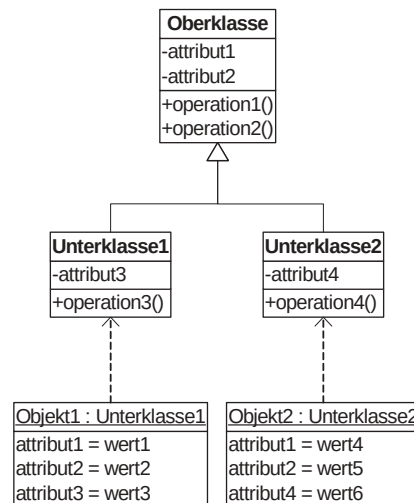


Abbildung 4.16: Vererbung mit Objekten: UML-Darstellung

Die Attribute `attribut1`, `attribut2` und `attribut3` stehen also in den Objekten gleichberechtigt nebeneinander. Das Gleiche gilt für die Operationen `operation1`, `operation2` und `operation3`.

Beschreiben wir einen Wecker, so können wir seine Attribute und Operationen wie bisher einfach benennen. Eine andere Möglichkeit, die dem menschlichen Denken eher entspricht, ist die assoziative Ähnlichkeitsbeschreibung. Dabei hilft es Menschen, neue Erkenntnisse immer auf bekannte Sachverhalte zurückzuführen.

Im Falle des Weckers erscheint uns dieses Vorgehen trivial. Dennoch verdeutlicht es das Prinzip. Versetzen wir uns in die Lage eines Kindes, das zwar eine Uhr, aber keinen Wecker kennt. Um dem Kind das Konzept Wecker zu vermitteln, greifen wir auf das uns am ähnlichsten erscheinende bekannte Konzept – die Uhr – zurück und beschreiben davon ausgehend die Unterschiede und Erweiterungen, um aus dem Konzept Uhr das Konzept Wecker abzuleiten.

Ein Kind kann mit der Aussage: »Ein Wecker funktioniert wie eine Uhr, nur dass er zu einer festgelegten Uhrzeit zusätzlich klingeln kann«, sehr schnell das neue Konzept in den Gesamtzusammenhang seines Wissens einordnen. Damit haben wir seine Vorstellung eines Weckers auf die Vorstellung einer Uhr zurückgeführt.

Übernehmen wir diese Vorstellung in das Klassendiagramm, nutzen wir die Vererbungsbeziehung, um den Zusammenhang von Uhr und Wecker darzustellen.

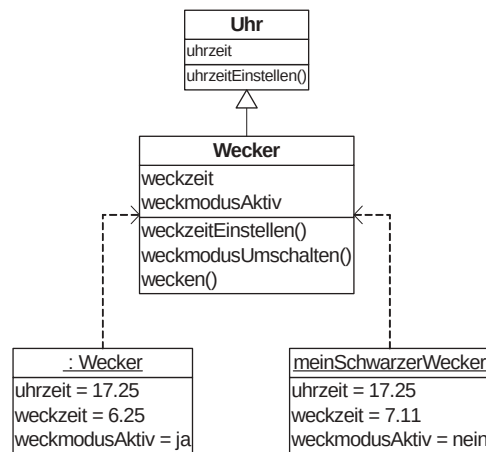


Abbildung 4.17: Vererbung: Uhr – Wecker

Die Abbildung zeigt die Vererbungsbeziehung zwischen den Klassen **Uhr** und **Wecker**, wobei die **Uhr** die vererbende Oberklasse und der **Wecker** die erbende Unterklasse ist. In der Klasse **Wecker** entspringt genau ein Vererbungspfeil, so dass hier eine Einfachvererbung vorliegt.

- ✓ Die Klasse **Uhr** gibt das Attribut `uhrzeit` sowie die Operation `uhrzeitEinstellen()` über die Vererbungsbeziehung (entgegen der Pfeilrichtung) an die Klasse **Wecker** weiter.

- ✓ Die Klasse `Wecker` definiert zusätzlich zu dem geerbten Attribut `uhrzeit` die Attribute `weckzeit` und `weckmodusAktiv`.
- ✓ Darüber hinaus vereinbart die Klasse `Wecker` über die geerbte Operation `uhrzeitEinstellen()` hinaus die Operationen `weckzeitEinstellen()` und `weckmodusUmschalten()`.
- ✓ Der `Wecker` besitzt damit die vollständige Funktionalität einer Uhr, die um die Weckfunktionen erweitert wurde.
- ✓ Die aus der Klasse `Wecker` erzeugten Objekte `:Wecker` und `meinSchwarzerWecker` können folglich mit den Attributen `uhrzeit`, `weckzeit` und `weckmodusAktiv()` beschrieben werden. Darüber hinaus besitzen sie die Operationen `uhrzeitEinstellen()`, `weckzeitEinstellen()` und `weckmodusUmschalten()`.



Besitzt eine Klasse genau eine direkte Oberklasse, liegt eine Einfachvererbung vor. Sie erkennen die Einfachvererbung im Klassendiagramm daran, dass von einer Klasse nur ein Vererbungs-pfeil ausgeht (d.h. seinen Ursprung hat).

Erweitern wir das Beispiel um ein Funkuhr, einen Radiowecker und einen Funkwecker, so erhalten wir eine mehrstufige Vererbungshierarchie. Das bedeutet, dass Attribute und Operationen über mehr als eine Stufe an mindestens zwei Klassen weitergegeben werden (siehe Abbildung 4.18).

- ✓ Die Klassen `Uhr` und `Wecker` sind aus dem vorhergehenden Beispiel unverändert übernommen.
- ✓ Die Klasse `Radiowecker` erbt, obwohl sie nicht in einer direkten Vererbungsbeziehung zur Klasse `Uhr` steht, indirekt über die Klasse `Wecker` auch die Uhrzeit-Funktionalität der Klasse `Uhr`.
- ✓ Die Klasse `Wecker` ist damit gleichzeitig Unterklasse der Oberklasse `Uhr` wie auch Oberklasse der Unterklasse `Radiowecker`.

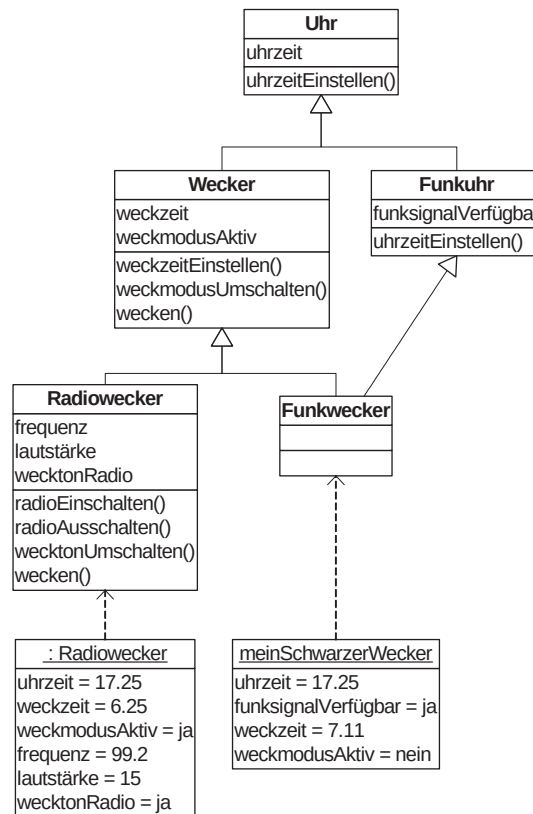


Abbildung 4.18: Mehrfachvererbung: Radiowecker – Funkwecker

- ✓ Die Klasse Radiowecker enthält damit die Attribute uhrzeit (aus der Klasse Uhr über Wecker an Radiowecker weitergegeben), weckzeit und weckmodusAktiv (aus der Klasse Wecker an Radiowecker vererbt) und die in der Klasse Radiowecker selbst definierten Attribute frequenz, lautstärke und wecktonRadio.
- ✓ Die Attribute frequenz und lautstärke halten die eingestellten Optionen für den Radiobetrieb fest.

- ✓ Das Attribut `wecktonRadio` enthält die Information, ob der Wecker mit Klingeln oder dem eingestellten Radiosender weckt.
- ✓ Wie die Attribute werden auch die Operationen an die Klasse Radiowecker weitergegeben. Damit enthält sie die Operationen `uhrzeitEinstellen()` (aus der Klasse `Uhr`), `weckzeitEinstellen()`, `weckmodusUmschalten()` und `wecken()` (aus der Klasse `Wecker`) sowie die selbst vereinbarten Operationen `radioEinschalten()`, `radioAusschalten()`, `wecktonUmschalten()` und `wecken()`.
- ✓ Die in der Klasse `Radiowecker` vereinbarte Operation `wecken()` überschreibt die geerbte Operation `wecken()` der Klasse `Wecker`.
- ✓ Die im `Radiowecker` spezialisierte Operation `wecken()` prüft zunächst, ob die Option `wecktonRadio` gesetzt ist. Ist das der Fall, aktiviert sie die Operation `radioEinschalten()`. Ist die `Radio`-Option nicht gesetzt, so aktiviert sie die überschriebene Operation `wecken()`, die sie aus der Klasse `Wecker` geerbt hat. In diesem Fall klingelt der Wecker.



Mehrstufige Vererbung liegt vor, wenn in einer Vererbungshierarchie mindestens eine Klasse existiert, die sowohl von einer Oberklasse Attribute und Operationen erbt als auch die geerbten und neu vereinbarten eigenen Attribute und Operationen an eine Unterklasse weitergibt.

Die Operationsnamen `uhrzeitEinstellen()` und `wecken()` sind mehrfach verwendet worden. Diese Polymorphie ist nach objektorientierten Grundsätzen ausdrücklich zugelassen. Da in diesem Fall die betroffenen Klassen in einer Vererbungsbeziehung stehen, ersetzt die aktuelle Version die geerbte Version der Operation. Dieses Ersetzen bezeichnet man als *Redefinition* einer Operation. Nach außen stehen ausschließlich die spezialisierten Versionen der Operationen zur Verfügung. Innerhalb der Operationen können jedoch auch überschriebene Operationen verwendet werden.



Die *Redefinition* (Überschreiben) einer geerbten Operation ist erlaubt, da die Polymorphie (Vielgestalt) Grundkonzept der Objektorientierung ist. Überschriebene Operationen können intern, d.h. innerhalb von Operationen dieser Klasse weiter verwendet werden.

Neben einer mehrstufigen Vererbungsbeziehung und überschriebenen Operationen zeigt die Abbildung auch die Mehrfachvererbung, die bei der Klasse *Funkwecker* gegeben ist. Der *Funkwecker* erbt Attribute und Operationen von seinen beiden direkten Oberklassen *Wecker* und *Funkuhr*.



Mehrfachvererbung liegt dann vor, wenn eine Unterklasse mehr als eine direkte Oberklasse besitzt. Im UML-Klassendiagramm erkennen Sie Mehrfachvererbung daran, dass von einer Unterklasse mindestens zwei Vererbungspfeile ausgehen (d.h., sie entspringen der Unterklasse).

Die Mehrfachvererbung gibt dem Modellierer einige Probleme auf, die wir anhand des Weckerbeispiels exemplarisch aufarbeiten wollen.

- ✓ Die Klasse *Funkuhr* ist eine Spezialisierung der Klasse *Uhr*, da sie eine spezielle Art zum Einstellen der Uhrzeit besitzt. Damit ist die Klasse *Uhr* in der umgekehrten Sicht eine Generalisierung der Klasse *Funkuhr*.
- ✓ Sowohl die Klasse *Uhr* als auch die Klasse *Funkuhr* vereinbaren eine Operation `zeitEinstellen()`. Die *Funkuhr* überschreibt die Operation `zeitEinstellen()`, die sie aus der Klasse *Uhr* geerbt hat. Sie passt die Operation also an die speziellen Bedürfnisse einer *Funkuhr* an.
- ✓ Bei einer »normalen« *Uhr* nehmen wir die Zeiteinstellung von Hand vor. Für das Modell bedeutet dies, dass die Werte vom Benutzer vorgegeben werden.

- ✓ Bei einer Funkuhr wird geprüft, ob ein Funksignal, das die Uhrzeit übermittelt, empfangen werden kann. Ist das der Fall, so wird die Uhr in einem periodischen Abstand nach dieser Funkzeit gestellt.
- ✓ Das Attribut `funksignalVerfügbar` enthält die Information, ob ein Funksignal anliegt.
- ✓ Liegt kein Funksignal an, so soll der Benutzer die Möglichkeit erhalten, die Uhr von Hand zu stellen. In dem Fall wird intern innerhalb der spezialisierten Operation `zeitEinstellen()` eines Funkuhr-Objekts die geerbte und redefinierte Operation `zeitEinstellen()` der Klasse `Uhr` aktiviert.
- ✓ Die Klasse `Funkwecker` besitzt mit den Klassen `Funkuhr` und `Wecker` zwei direkte Oberklassen. Damit ist der Tatbestand einer Mehrfachvererbung erfüllt.
- ✓ Das Attribut `uhrzeit` wird über die Klasse `Wecker` und zusätzlich über die Klasse `Funkuhr` an die Klasse `Funkwecker` vererbt. Damit ist es nach den Grundsätzen der Vererbung in der Klasse `Funkwecker` doppelt verfügbar.
- ✓ Das Attribut `uhrzeit` hat einen Konflikt verursacht, der individuell gelöst werden muss. Im Fall des Funkweckers ist nur eine Uhrzeit gegeben, so dass in diesem konkreten Fall nur ein Attribut `uhrzeit` in der Klasse `Funkwecker` verfügbar gemacht wird.
- ✓ Die Klasse `Funkwecker` erbt die Operation `uhrzeitEinstellen()` ebenfalls über beide Vererbungspfade, so dass sie nach Vererbungsregeln zweimal vereinbart ist.
- ✓ Der Konflikt wird in diesem Fall dadurch verschärft, dass zwei unterschiedliche Versionen der Operation `uhrzeitEinstellen()` vorliegen. Über den `Wecker` gelangt die Ursprungsversion der Klasse `Uhr` und über die `Funkuhr` die spezialisierte Variante, die das Anlegen eines Funksignals berücksichtigt, in die Klasse `Funkuhr`.

- ✓ Da ein Funkwecker sich bei Einstellen der Uhrzeit nicht von einer Funkuhr unterscheidet, streichen wir die Ursprungs-Variante. Theoretisch denkbar wäre auch, dass beide Varianten der Operation verfügbar sind oder nur die Ursprungsversion.

Im Falle der Mehrfachvererbung können Konflikte auftreten, die eine individuelle Konfliktlösungsstrategie erfordern und nicht generell gelöst werden können. Durch die Mehrfachvererbung wird ein Modell kompliziert, und aus der grafischen Darstellung sind für das Verständnis notwendige Konfliktlösungen nicht mehr ersichtlich. Aus diesem Grund verzichten viele Modellierer auf den Einsatz der Mehrfachvererbung und ersetzen ihn durch den Einsatz von Schnittstellen oder eine Kombination aus Assoziation und Einfachvererbung.



Konflikte treten bei der Mehrfachvererbung auf, wenn Attribute und Operationen mehrfach an eine Unterklasse vererbt werden. Hier muss individuell geklärt werden, ob das betroffene Attribut bzw. die betroffene Operation einmal oder mehrfach in der Klasse enthalten ist. Verschärft wird der Konflikt, wenn die Ursprungsversion einer Operation mit der spezialisierten Operation über verschiedene Vererbungspfade an die Unterklasse weitergegeben. Auch hier gibt es keine allgemein gültige Konfliktlösung. Es mag Fälle geben, in denen nur die spezialisierte Operation verfügbar ist. In anderen Fällen mag es sinnvoll sein, dass beide Operationen aufgerufen werden können, oder in Ausnahmefällen kann es sogar vorkommen, dass nur die Ursprungsversion der Operation verfügbar gemacht wird.

Aus dem gleichen Grund verzichten einige Programmiersprachen wie Java und C# auf die Implementierung der Mehrfachvererbung. C++ hingegen erlaubt Mehrfachvererbung.



Wollen Sie unabhängig von der Wahl der Programmiersprache sein, empfiehlt es sich, die Mehrfachvererbung zu vermeiden und im Bedarfsfall durch Schnittstellen oder eine Kombination aus Assoziationen und Einfachvererbung nachzubilden.

Betrachten wir die Vererbung als eines der zentralen Konzepte der Objektorientierung an unserem Fahrzeug.

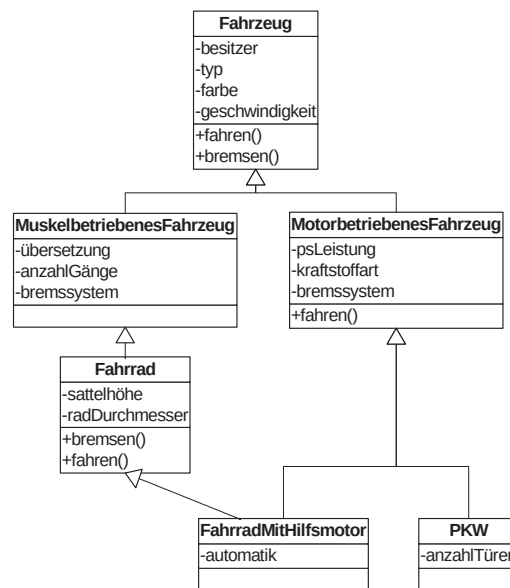


Abbildung 4.19: Mehrfachvererbung Fahrzeuge

- ✓ Die Klasse Fahrzeug mit den Attributen besitzer, typ, farbe und geschwindigkeit sowie den Operationen fahren() und bremsen() besitzt mit den Klassen MuskelbetriebenesFahrzeug und MotorbetriebenesFahrzeug zwei direkte Unterklassen.
- ✓ Muskelbetriebene Fahrzeuge definieren zusätzlich zu den geerbten Attributen und Operationen die Attribute übersetzung, anzahlGänge und bremssystem.
- ✓ Das Fahrrad erweitert Muskelbetriebene Fahrzeuge um die Attribute sattelhöhe und radDurchmesser. Die Operationen bremsen() und fahren() werden speziell für Fahrräder angepasst.

- ✓ Motorbetriebene Fahrzeuge vereinbaren zusätzlich die Attribute `psleistung`, `kraftstoffart` und `bremssystem`. Die Operation `fahren()` wird spezialisiert und auf motorgetriebene Fahrzeuge angepasst.
- ✓ Der PKW als spezielles motorgetriebenes Fahrzeug vereinbart zusätzlich noch das Attribut `anzahlTüren`.
- ✓ Die Klasse `FahrradMitHilfsmotor` besitzt eine Mehrfachvererbungsstruktur und übernimmt Attribute und Operationen sowohl von der Oberklasse `Fahrrad` als auch von der Oberklasse `MotorbetriebenesFahrzeug`.
- ✓ Damit enthält die Klasse `FahrradMitHilfsmotor` über den linken Vererbungsweig die Attribute:
 - ✓ `besitzer`, `typ`, `farbe` und `geschwindigkeit` aus der Klasse `Fahrzeug`,
 - ✓ `übersetzung`, `anzahlGänge` und `bremssystem` aus der Klasse `MuskelbetriebenesFahrzeug`,
 - ✓ `sattelhöhe` und `radDurchmesser` aus der Klasse `Fahrrad`.
- ✓ Über den rechten Vererbungsweig erhält sie die Attribute:
 - ✓ `besitzer`, `typ`, `farbe` und `geschwindigkeit` aus der Klasse `Fahrzeug`
 - ✓ `psleistung`, `kraftstoffart`, `bremssystem` aus der Klasse `MotorbetriebenesFahrzeug`
- ✓ Dazu kommt das selbst definierte Attribut `automatik`.

Die Konfliktlösung der über beide Äste vererbten und damit mehrfach vorhandenen Attribute könnte folgendermaßen aussehen:

- ✓ Die Attribute `besitzer`, `typ`, `farbe` und `geschwindigkeit` sollten nur einmal in der Klasse `FahrradMitHilfsmotor` vorhanden sein.
- ✓ Das Attribut `bremssystem` sollte doppelt in der Klasse `FahrradMitHilfsmotor` vorhanden sein, da sowohl eine Rücktrittbremsanlage für Fahrräder als auch die Bremssysteme eines motorbetriebenen Fahrzeugs eingebaut werden.

Damit haben wir gesehen, dass es auch sinnvoll sein kann im Einzelfall über zwei Wege vererbte Attribute in einer Unterklasse zuzulassen. Zur Unterscheidung ist es theoretisch denkbar die Attribute umzubenennen oder immer mit der zusätzlichen Klassenangabe ihrer Herkunft aufzurufen.

Betrachten wir die Operationen der Klasse `Fahrrad` mit `Hilfsmotor`, so entstehen durch die Mehrfachvererbung ebenfalls Konflikte, die individuell gelöst werden müssen.

- ✓ Über den linken Vererbungsweig erbt die Klasse `Fahrrad` mit `Hilfsmotor` die Operationen `fahren()` und `bremsen()` aus der Klasse `Fahrrad`, die die Ursprungsvarianten der Klasse `Fahrzeug` überschrieben haben.
- ✓ Über den rechten Vererbungsweig werden die Operationen `bremsen()` als Ursprungsvariante aus der Klasse `Fahrzeug`, sowie `fahren()`, die eine redefinierte Variante in der Klasse `MotorbetriebenesFahrzeug` vereinbart ist, vererbt.

Damit ist sowohl die Operation `fahren()` als auch die Operation `bremsen()` doppelt an die Klasse `FahrradMitHilfsmotor` vererbt. Eine Konfliktlösung könnte hier folgendermaßen aussehen:

- ✓ Die Operation `fahren()` wird in doppelter Ausführung in der Klasse `FahrradMitHilfsmotor` verfügbar gemacht, da das `Fahrrad` mit `Hilfsmotor` sowohl mit `Motor-` als auch mit `Muskelkraft` gefahren werden kann.
- ✓ Die Operation `bremsen()` wird ebenfalls doppelt in die Klasse integriert, da die doppelt vorhandenen Bremssysteme individuell gesteuert werden müssen.



Es gibt Fälle, in denen es sinnvoll ist, Attribute und Operationen, die durch Mehrfachvererbung doppelt in einer Klasse vorkommen, in diese Klasse mehrfach zu integrieren. Jedes Attribut und jede Operation muss individuell geprüft werden, ob ein mehrfaches Aufnehmen inhaltlich sinnvoll erscheint oder nicht.

Um die Übersichtlichkeit Ihrer Klassenstrukturen in Klassendiagrammen zu erhalten, sollten Sie nach Möglichkeit auf die Mehrfachvererbung verzichten. Ebenfalls unübersichtlich werden zu tiefe Vererbungshierarchien, da der Betrachter entlang einer Vererbungslinie die Attribute und Operationen einer Klasse zusammenstellen muss. Als Anhaltspunkt für die Modellierung sollten wir eine drei-stufige bis vierstufige Hierarchie als sinnvolle Tiefe ansehen. Sie sollte in der Regel deutlich unterhalb des semantischen Differenzials von 7 liegen.



In komplexen Modellen ist es oft nur noch schwer nachzuvollziehen, woher einzelne Attribute und Operationen in tiefen Vererbungshierarchien stammen. Vielstufige und multiple Vererbungsstrukturen sollten Sie insbesondere dann, wenn sie in Kombination auftreten, möglichst vermeiden oder nur äußerst sparsam und mit Bedacht modellieren.

Diskriminatoren für Vererbungshierarchien

Auch Vererbungsbeziehungen können in der UML mit Zusicherungen (Constraints) versehen werden, die nähere Informationen über die Art der Beziehungen enthalten. Vordefiniert sind die folgenden Klassifizierungen:

- ✓ *disjoint*: Vererbungsbeziehungen sind standardmäßig disjoint. Dies bedeutet, dass grundsätzlich davon ausgegangen wird, dass keine Überschneidungen in den Klassendefinitionen vorliegen. Konsequenz daraus ist, dass im Falle der Mehrfachvererbung mehrfach definierte Attribute und Operationen aufgrund der durch disjoint festgeschrieben semantischen Unterschiedlichkeit auch unterschiedliche Inhalte verkörpern.
- ✓ *overlapping*: Die Zusicherung overlapping deutet darauf hin, dass Klassen überschneidend definiert sind.
- ✓ *complete*: Zu einer Klasse sind in einem objektorientierten Modell alle Unterklassen enthalten. Die Festlegung weiterer Unterklassen ist nicht beabsichtigt. Dabei müssen nicht alle im

Modell enthaltenen Klassen in einem Klassendiagramm abgebildet sein.

- ✓ *incomplete*: Das Modell ist noch nicht vollständig. Es sind zwar bereits einige Unterklassen vorhanden, doch ist angesichts der gegebenen Problemstellung damit zu rechnen, dass noch weitere Unterklassen gebildet werden.



Die Zusicherungen *complete* und *incomplete* besagen nichts darüber, ob ein Klassendiagramm alle Klassen anzeigt, sondern ob das Modell an sich vollständig ist.

Abstrakte Klassen – Bauplan ohne Bauen

Wird eine Klasse als abstrakt definiert, so können aus ihr niemals Objekte erzeugt werden. Es stellt sich die Frage, welche Existenzberechtigung eine Klasse dann noch hat:

- ✓ Dient eine Klasse nur dazu, als künstlich gebildete Oberklasse die Gemeinsamkeiten zugehöriger Unterklassen zu beschreiben, so kann sie abstrakt vereinbart werden.
- ✓ Klassen, die identitätslose Eigenschaften beschreiben, können ebenfalls abstrakt definiert werden. Als Beispiel sei hier die Uhrzeit genannt, die ja keine Identität und damit auch keinen Lebenszyklus mit Anfangs- und Endzeitpunkt besitzt.
- ✓ Gemäß UML-Notation wird der Klassenname einer abstrakten Klasse kursiv dargestellt.

Das folgende Beispiel zeigt eine abstrakte Klasse, die ausschließlich dazu vorgesehen ist, die Gemeinsamkeiten ihrer Unterklassen zu vereinbaren (siehe Abbildung 4.20).

- ✓ Jedes Konto, egal ob Sparkonto oder Girokonto, besitzt die Attribute `kontonr` und `kontostand`.
- ✓ Jedes Konto können Sie durch `auszahlen()` erleichtern und durch `einzahlen()` füllen.

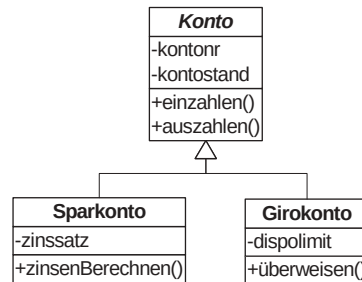


Abbildung 4.20: Abstrakte Klasse: Konto

- ✓ Gehen Sie zur Bank und wollen Sie ein Konto eröffnen, so fragt der Mitarbeiter, was für ein Konto es denn sein soll, ein Sparkonto oder ein Girokonto.
- ✓ Da die Klasse *Konto* ausschließlich dazu genutzt wird, die Gemeinsamkeiten aller Konten zu beschreiben, wird sie als abstrakte Klasse vereinbart. Der Klassenname *Konto* erscheint daher kursiv.

Assoziation – Attribute mit Identität

Eine Assoziation beschreibt eine fachliche Beziehung zwischen Klassen, die zwischen den aus diesen Klassen gebildeten Objekten realisiert werden. Eine Assoziation ist eine strukturelle Beziehung, über die die verbundenen Objekte miteinander kommunizieren. Man spricht auch davon, dass eine Assoziation Objekte miteinander bekannt macht. Die entstandene Verbindung wird auch als *link* bezeichnet. Sie werden technisch als Attribute implementiert, so dass man sie von gewöhnlichen Attributen, die als identitätslose Eigenschaften ein Objekt beschreiben, als Attribute mit Identität unterscheiden kann.



Attribute und Beziehungen beschreiben gemeinsam den Zustand eines Objektes. Beziehungen werden genau wie Attribute auch vererbt.

Einfache binäre Assoziation

In einer binären Assoziation sind genau zwei Objekte miteinander fachlich verbunden. Dabei können sowohl zwei Objekte von verschiedenen Klassen als auch zwei Objekte, die aus derselben Klasse erzeugt wurden, miteinander verbunden sein.



Abbildung 4.21: Assoziation: UML-Darstellung

Eine Assoziation wird im Klassendiagramm als einfache Verbindungslinie eingezeichnet, die im Fall der binären Assoziation zwei Klassensymbole miteinander verbindet.

- ✓ Die Assoziation erhält einen Namen und kann mit weiteren Attributen versehen werden.
- ✓ Die beiden Sterne unterhalb der Verbindungslinie kennzeichnen die *Kardinalitäten* der Assoziation. Sie geben an, wie viele Objekte der einen Klasse mit einem Objekt der anderen Klasse in Verbindung stehen. Im Falle des Sterns kann ein Objekt mit beliebig vielen Objekten der anderen Klasse fachlich verbunden sein.
- ✓ Eine Assoziation besitzt einen Assoziationsnamen (Assoziation).
- ✓ Die Leserichtung wird neben dem Assoziationsnamen als Pfeilspitze markiert, so dass Sie auch entgegen der gewohnten Leserichtung einen Assoziationsnamen von rechts nach links interpretieren können.
- ✓ Weiterhin nimmt jede Klasse in einer Assoziation eine bestimmte Rolle (Rolle1, Rolle2) ein, deren Bezeichnung neben der Klasse am entsprechenden Ende der Assoziation notiert werden kann.

Weitere hier nicht abgebildete Elemente von Assoziationen sind gerichtete, abgeleitete und qualifizierte Assoziationen, auf die wir später noch eingehen. Darüber hinaus können Assoziationen geordnet sein, wenn am Ende einer Assoziation der Zusatz *{ordered}* notiert ist.



Hüten Sie sich davor, jede noch so unbedeutende Assoziation mit allen Elementen vollständig zu spezifizieren. Gerade in der Analysephase, in der Sie das Modell nutzen, um mit den Anwendern die fachlichen Inhalte zu klären, gehen Übersichtlichkeit, Einfachheit und Klarheit vor Vollständigkeit. Modellieren Sie stets nach dem Grundsatz: »So viel wie nötig, aber nicht mehr.«

Die folgende Abbildung verdeutlicht dies am Beispiel der Assoziation Bankkunde - Girokonto.

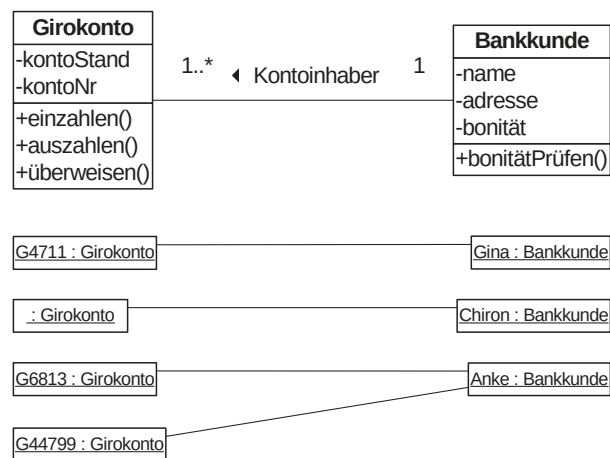


Abbildung 4.22: Binäre Assoziation: Bankkunde – Girokonto

- ✓ Im oben abgebildeten Klassendiagramm besitzt die Klasse Bankkunde eine Verbindungslinie (Assoziation) zur Klasse Girokonto.
- ✓ Dabei tritt der Bankkunde in der Rolle des Inhabers auf. Den Assoziationsnamen Kontoinhaber – in der Abbildung gekennzeichnet durch die Beschriftung der Verbindungslinie, lesen Sie von links nach rechts.
- ✓ Die Ziffern an den Klassensymbolen legen die Kardinalitäten fest. Sie besagen, dass ein Bankkunde ein oder mehrere (1..*) Girokonten als Inhaber führen kann, und umgekehrt, dass ein Girokonto genau einen (1) Kontoinhaber besitzt.
- ✓ Das Girokonto-Objekt G4711 ist dem Bankkunden Gina zugeordnet.
- ✓ Der Bankkunde Chiron ist Inhaber des :Girokonto.
- ✓ Die Girokonten G6813 und G44799 werden von der Bankkundin Anke geführt.



Wenn Sie mit der Entity-Relationship-Modellierung im Bereich der relationalen, postrelationalen oder objektrelationalen Datenbanken vertraut sind, finden Sie mit den Assoziationen der UML ein Konstrukt vor, das den Relationen ähnlich ist. Allerdings werden die Beziehungen im objektorientierten Modell völlig anders realisiert.

Für die Exemplare der beiden Klassen bedeutet dies, dass jedes Objekt der Klasse Girokonto mit genau einem (1) Objekt der Klasse Bankkunde verbunden ist und über diese Verbindung kommunizieren kann. Jedes Objekt der Klasse Bankkunde steht umgekehrt mit einem oder mehreren (1..*) Objekten der Klasse Girokonto in Verbindung.

Die Kardinalität definiert also im Rahmen einer Assoziation, wie viele Objekte der einen Klasse ein bestimmtes Objekt der anderen Klasse kennen. Die UML lässt folgende Kardinalitäten zu.

Kardinalität	Bedeutung
1	Verbindung zu genau einem Exemplar der verbundenen Klasse
0..1	Verbindung mit keinem bis maximal einem Exemplar der verbundenen Klasse
*	Verbindung mit keinem, einem oder mehreren Exemplar(en) der verbundenen Klasse
2..*	Verbindung mit zwei bis vielen Exemplaren der verbundenen Klasse
5	Verbindung mit genau fünf Exemplaren der verbundenen Klasse
1, 5, 8	Verbindung mit einem, fünf oder acht Exemplar(en) der verbundenen Klasse
1..7, 10, 13..*	Die Anzahl der zugeordneten Exemplare der verbundenen Klasse muss in jedem Fall von null, acht, neun und zwölf verschieden sein

Tabelle 4.3: Bedeutung von Kardinalitäten

Der Bezugspunkt von Kardinalitäten wird oft verwechselt. Die Anzahl der an einer Beziehung beteiligten Objekte wird immer neben dem Klassensymbol angegeben. Das bedeutet, dass ein Exemplar der Ausgangsklasse mit einer Anzahl von Exemplaren der gegenüberliegenden Klasse, neben der auch die zugehörige Kardinalität notiert ist, verbunden ist.

Das bedeutet bezogen auf das Beispiel Girokonto, dass ein Bankkunde mit 1..* Girokonten verbunden und umgekehrt ein Girokonto mit genau 1 Bankkunden assoziiert sein muss.

Sollten Sie bei den folgenden Beispielen mit der Deutung der Kardinalitäten Schwierigkeiten haben, fertigen Sie ruhig ein Objekt-

diagramm an, das beispielhaft die angegebenen Kardinalitäten umsetzt.

Assoziationsklasse

Ein objektorientiertes Modell stellt stets eine subjektive Sicht auf eine Problemstellung dar, wie unsere Überlegungen zum objektorientierten Denken deutlich darlegen. Welche Elemente und Beziehungen in das Modell einfließen, hängt daher von der persönlichen individuellen Sicht der Beteiligten ab.

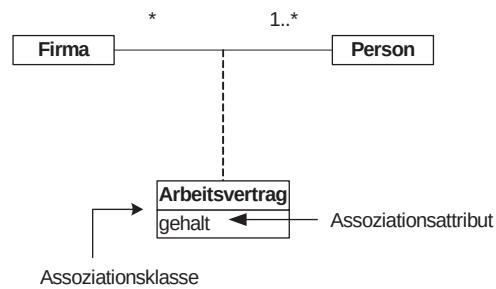
Darüber hinaus bietet die Objektorientierung und insbesondere die UML zahlreiche Modellierungsalternativen, um Sachverhalte darzustellen. So besteht auch die Möglichkeit, Assoziationen um Klasseigenschaften zu erweitern. Dies ist dann sinnvoll, wenn eine detaillierte Beschreibung der Beziehung zweier Klassen notwendig erscheint.

Eine solche *assoziative Klasse* oder *Assoziationsklasse* besitzt in ihrer Spezifikation neben beschreibenden Attributen und Operationen auch die Informationen über die Beziehung zu den beteiligten Klassen. Die Attribute einer Assoziationsklasse werden Assoziationsattribute genannt, da sie eine Assoziation näher beschreiben.

Eine Assoziationsklasse wird im Klassendiagramm wie eine normale Klasse dargestellt. Einzige Besonderheit ist eine gestrichelte Verbindungslinie zu der Assoziation, die sie näher beschreibt.

Im dargestellten Beispiel erläutert die Assoziationsklasse Arbeitsvertrag die Assoziation zwischen einer Firma und einer mit ihr verbundenen Person. Das Attribut *gehalt* stellt in diesem Fall ein Assoziationsattribut dar, das den Arbeitsvertrag genauer spezifiziert (siehe Abbildung 4.23).

Die Abbildung zeigt, dass eine Firma 1..* Personen beschäftigt und umgekehrt jede Person an * Firmen über einen Arbeitsvertrag gebunden ist.

**Abbildung 4.23:** Assoziationsklasse: Arbeitsvertrag

Eine assoziative Klasse kann aufgelöst werden in eine gewöhnliche Klasse. Dies ist notwendig, um ein Modell für die Implementierung in einer Programmiersprache vorzubereiten, die keine derartigen Konstruktionen kennt.

- ✓ Entfernen Sie die Assoziation von Firma zu Person.
- ✓ Definieren Sie eine Assoziation von Firma zu Arbeitsvertrag.
- ✓ Vereinbaren Sie eine Verbindung von Person zu Arbeitsvertrag.
- ✓ Die Kardinalität neben der Firma wird mit 1 notiert.
- ✓ Ebenso beträgt die Kardinalität neben der Klasse Person 1.
- ✓ Die Kardinalität am Ende der Assoziation zwischen Firma und Arbeitsvertrag ist nun mit 1..* neben der Klasse Arbeitsvertrag notiert.
- ✓ Die Kardinalität am Ende der Assoziation zwischen Person und Arbeitsvertrag beträgt * und wird am Assoziationsende neben der Klasse Arbeitsvertrag notiert.

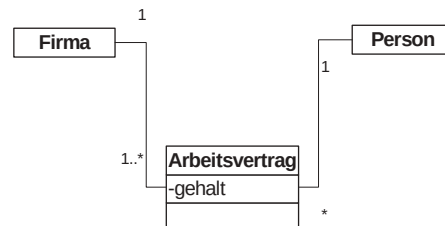


Abbildung 4.24: Aufgelöste Assoziationsklasse: Arbeitsvertrag

Somit werden in einer Firma 1..* Arbeitsverträge geschlossen. Jeder Arbeitsvertrag kann hingegen genau einer Firma zugeordnet werden. Eine Person kann * (beliebig viele oder keinen) Arbeitsverträge besitzen, auf der anderen Seite ist jeder Arbeitsvertrag genau einer Person zugeordnet.

Damit ist im Ergebnis dieselbe Beziehung ausgedrückt, die auch in der Assoziationsklasse vereinbart war.

Eine Assoziationsklasse kann in eine gewöhnliche Klasse überführt werden, indem die zwischen den Klassen ursprünglich vereinbarte Assoziation in zwei Assoziationen aufgespalten wird. Gehen Sie in folgenden Schritten vor:

- ✓ Entfernen Sie die ursprüngliche Assoziation.
- ✓ Definieren Sie von jeder ursprünglich assoziierten Klasse eine Assoziation zur ursprünglichen Assoziationsklasse.
- ✓ Die ursprünglichen Kardinalitäten wandern von den ursprünglichen Enden an die Enden der neu gebildeten neben die ursprüngliche Assoziationsklasse.
- ✓ Die Kardinalitäten neben den ursprünglich assoziierten Klassen betragen immer 1.

Reflexive Assoziation

Ein Sonderfall der binären Assoziation ist die rekursive oder *reflexive Assoziation*, die eine Verbindung von Objekten beschreibt, die aus derselben Klasse erzeugt sind.

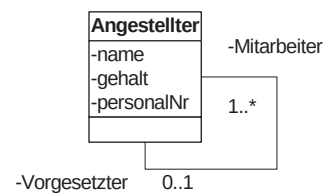


Abbildung 4.25: Reflexive Assoziation: Angestellter

Die vorstehende Abbildung zeigt als Beispiel einer reflexiven Assoziation die Klasse `Angestellter`. Objekte dieser Klasse können sowohl in der Rolle eines Mitarbeiters als auch in der Rolle eines Vorgesetzten auftreten.

Dementsprechend liegen Start- und Endpunkt der Assoziation in derselben Klasse `Angestellter`. Die Klasse bezieht sich auf sich selbst. Interpretieren wir die Assoziation, so kann ein Angestellter in seiner Rolle als Vorgesetzter 1..* Angestellte als Mitarbeiter führen. Auf der anderen Seite ist jeder Angestellte in seiner Rolle als Mitarbeiter 0..1 Vorgesetzten unterstellt.

Mehrgliedrige Assoziation

Die bisherigen Ausführungen beschränken sich auf binäre Assoziationen, die jeweils zwei Objekte miteinander verbinden. Darüber hinaus ermöglichen mehrgliedrige Assoziationen oder n-äre Assoziationen eine gleichzeitige Kommunikationsbeziehung zwischen drei oder mehr Objekten.

Die Abbildung zeigt das Beispiel einer ternären (dreigliedrigen) Assoziation anhand einer Flugbuchung. Das Beispiel zeigt keine

Operationen, da sie für das Verständnis keine Rolle spielen. Die Beziehung stellt die Buchung eines Platzes in einem Flugzeug für einen Passagier dar. Dazu werden die Flugnummer, das Datum des Fluges sowie die Anzahl der Sitzplätze in dem eingesetzten Flugzeugtyp benötigt. Jeder Platz hat eine Platznummer, befindet sich in einer bestimmten Klasse (First-, Business-, Economy-Class) und wird zu einem festgelegten Preis angeboten. Ein Passagier bucht über seinen Namen. Darüber hinaus besitzt er einen Status (etwa Vielflieger). Die Kardinalitäten betragen jeweils 1, was bedeutet, dass genau ein Passagier einen Platz in einem Flugzeug bucht.

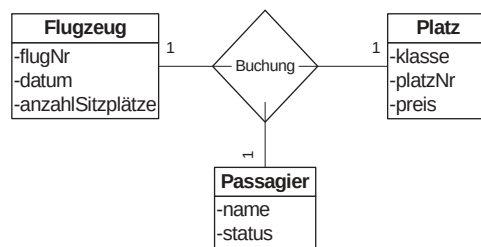


Abbildung 4.26: Ternäre Assoziation: Flugbuchung

Die drei Klassen Flugzeug, Platz und Passagier sind durch die Beziehung Buchung miteinander verbunden. Die Abbildung zeigt, dass von einer Raute ausgehende Verbindungslinien zu den Klassen führen, die an der Assoziation beteiligt sind. Im Zentrum der Raute wird der Name der Assoziation notiert.

Auch eine mehrgliedrige Beziehung lässt sich mit Hilfe einer Assoziationsklasse näher beschreiben. Dazu führt eine gestrichelte Linie von der Raute zu der entsprechenden Assoziationsklasse (siehe Abbildung 4.27).

Im obigen Beispiel verbindet die gestrichelte Linie die ternäre oder dreigliedrige Assoziation über die Assoziationsraute mit der Assoziationsklasse Buchung. Für jede Buchung werden das Buchungsdatum, der Status, ob der Passagier die Buchung bestätigt hat und der Sonderwunsch nach Diät-Essen hinterlegt.

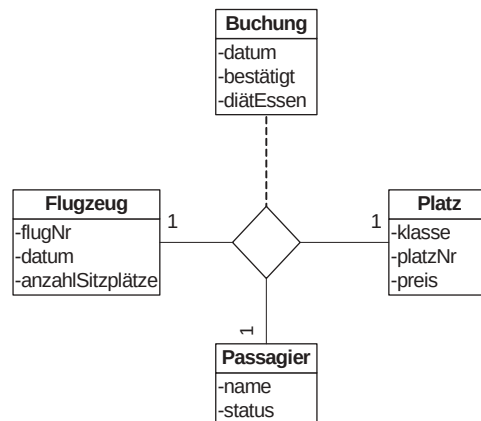


Abbildung 4.27: Ternäre Assoziation mit Assoziationsklasse: Flugbuchung

Auch eine Assoziationsklasse, die eine mehrgliedrige Verbindung näher erläutert, kann in eine gewöhnliche Klasse überführt werden. Dazu übernimmt die bisherige Assoziationsklasse die Zusammenführung der an der Assoziation beteiligten Klassen.

Qualifizierte Assoziation

Eine qualifizierte Assoziation stellt die Verbindung zwischen den Objekten der beteiligten Klassen über ein qualifizierendes Attribut her. Gewöhnlich werden Objekte per direktem Verweis einander zugeordnet. Dies geschieht über eine Verweisliste, die alle verbundenen Objekte referenziert.

Im Fall der qualifizierenden Assoziation dient der Wert eines qualifizierenden Attributs dazu, die zugeordneten Objekte in Gruppen zu unterteilen. Die Objekte, die denselben Attributwert des qualifizierenden Attributs besitzen, gehören einer gemeinsamen Partition an.

Das qualifizierende Attribut wird am Assoziationsende neben dem Zielobjekt notiert. Das qualifizierende Attribut ist Element der Assoziation und nicht, wie fälschlicherweise oft behauptet wird, Attribut der Klasse.

Das folgende Beispiel verdeutlicht die Zusammenhänge, die abstrakt erklärt verwirrend erscheinen.

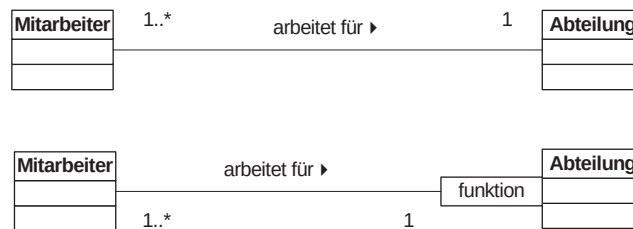


Abbildung 4.28: Qualifizierte Assoziation: Mitarbeiter

- ✓ Die Abbildung enthält die alternative Darstellung der Beziehung von Mitarbeiter zu einer Abteilung, für die er arbeitet.
- ✓ Dabei gehört jeder Mitarbeiter einer Abteilung an, während jede Abteilung einen oder mehrere Mitarbeiter beschäftigt.
- ✓ Das qualifizierende Attribut *funktion*, das neben dem Rechteck der Klasse *Abteilung* abgebildet ist, kann beispielsweise die Mitarbeiter einer Abteilung in die Funktionspartitionen *Leitung*, *Verwaltung* und *Sachbearbeitung* unterteilen. *Leitung*, *Verwaltung* und *Sachbearbeitung* sind Werte des zur individuellen Assoziation gehörenden qualifizierten Attributes *funktion*.
- ✓ Somit übt jeder Mitarbeiter in einer Abteilung eine bestimmte Funktion (*Leitung*, *Verwaltung*, *Sachbearbeitung*) aus. Diese individuelle Funktion können auch andere Mitarbeiter der Abteilung besitzen, die dann mit ihm eine gemeinsame Untermenge (Partition) der Abteilungszugehörigkeiten bilden.

Abgeleitete Assoziation

Assoziationen, die aus anderen Assoziationen berechnet werden können, bezeichnet man als abgeleitete Assoziationen. Abgeleitete Assoziationen werden nicht implementiert. Sie werden bei Bedarf ad hoc aus gewöhnlichen Assoziationen hergeleitet.

In der UML-Notation werden abgeleitete Assoziationen durch einen Schrägstrich vor dem Assoziationsnamen gekennzeichnet.

Die drei Assoziationen zwischen den Klassen Mannschaft, Fußballliga und Spieler verdeutlichen den Zusammenhang.

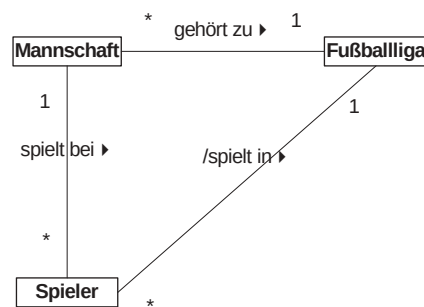


Abbildung 4.29: Abgeleitete Assoziation: Fußball

- ✓ Ein Spieler spielt bei einer Mannschaft, während einer Mannschaft mehrere Spieler angehören.
- ✓ Jede Mannschaft spielt in einer Fußballliga, während in einer Fußballliga mehrere Mannschaften gegeneinander antreten.
- ✓ Die Information, in welcher Fußballliga ein bestimmter Spieler gegen den Ball tritt, lässt sich jederzeit über die Assoziationen spielt bei und gehört zu ermitteln.
- ✓ Die abgeleitete Assoziation spielt in muss nicht physisch implementiert sein. Sie kann ad hoc berechnet werden, indem der Pfad ausgehend von einem Spieler über seine Mannschaft zur Fußballliga der Mannschaft verfolgt wird.

- ✓ Somit spielt jeder Spieler in genau einer Fußballliga. Jede Liga ermöglicht vielen Kickern das Fußballvergnügen.



Ob eine im UML-Modell als abgeleitete Assoziation definierte Beziehung auch im späteren Programm ad hoc hergeleitet wird oder ob sie als gewöhnliche Assoziation realisiert wird, hängt vom Berechnungsaufwand ab.

Gerichtete Assoziation

Eine gerichtete Assoziation liegt vor, wenn nur ein Objekt einer Assoziation das oder die anderen Objekt(e) der Beziehung kennt. Das bedeutet, dass eine gerichtete Assoziation nur einseitig navigierbar ist. Damit ist ein Objekt der Beziehung »blind« und weiß nicht um die Beziehung.

Die Navigationsrichtung einer Assoziation wird durch eine Pfeilspitze am Assoziationsende notiert.

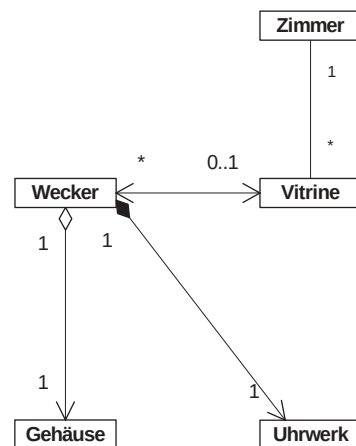


Abbildung 4.30: Gerichtete Assoziationen: Wecker

- ✓ Ein Wecker-Objekt kennt das Gehäuse-Objekt, das ihm zugeordnet wird.
- ✓ Der Wecker kann auf das Gehäuse-Objekt zugreifen. Einen derartigen Zugriff bezeichnet man als Navigation in Pfeilrichtung.
- ✓ Der Wecker kann ebenfalls auf das Uhrwerk zugreifen, da die Komposition in Richtung Uhrwerk navigierbar ist.
- ✓ Gehäuse und Uhrwerk haben keine strukturelle Möglichkeit, auf den Wecker zuzugreifen, da keine Navigationsmöglichkeit vorgesehen ist. Die Verbindungen bezeichnet man auch als einseitig gerichtet.
- ✓ Die Assoziation zwischen Wecker und Vitrine ist zweiseitig navigierbar, so dass ein Wecker-Objekt auf ein zugeordnetes Vitrinen-Objekt und umgekehrt auch das Vitrinen-Objekt auf das Wecker-Objekt zugreifen kann.
- ✓ Die Assoziation zwischen Vitrine und Zimmer enthält keine Informationen über die Navigierbarkeit. Das kann entweder als beidseitig navigierbar oder als nicht festgelegte Navigierbarkeit interpretiert werden.

Zweiseitig navigierbare Assoziationen werden zur Implementierung in zwei einseitige, gerichtete Assoziationen zerlegt, die jedoch voneinander abhängig sind. Das bedeutet, dass zu jedem Verweis, der bei einem Wecker-Objekt auf ein Vitrinen-Objekt verweist, ein korrespondierender Rückverweis im Vitrinen-Objekt vereinbart sein muss.

Die Abbildung verdeutlicht das Aufbrechen der zweiseitigen, gerichteten Assoziation in zwei einseitig gerichtete Assoziationen (siehe Abbildung 4.31).

Von den gerichteten Assoziationen deutlich abzugrenzen ist die ungerichtete Variante, die auch als unspezifiziert interpretiert werden kann. In diesem Fall ist die Navigierbarkeit noch nicht festgelegt und muss spätestens bei der Implementierung vereinbart werden.

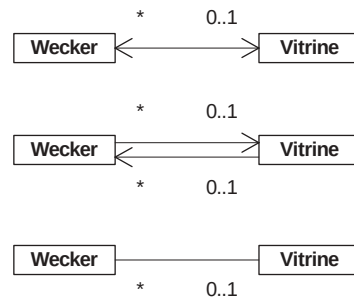


Abbildung 4.31: Gerichtete Assoziation: Wecker

Zusicherungen

Zusicherungen haben wir bereits in anderen Zusammenhängen wie z.B. für Attribute, Operationen kennen gelernt. Auch Assoziationen können durch Zusicherungen genauer spezifiziert werden. Auch im Kontext von Assoziationen dienen Zusicherungen (Constraints) dazu, Einschränkungen oder Bedingungen zu formulieren, die während der gesamten Existenz einer Objektverbindung erfüllt sein müssen.

Im Bereich der Assoziationen finden insbesondere die zwei Zusicherungen *or* und *subset* eine wichtige Rolle.

- ✓ Die Zusicherung *or* (or-Restriktion, oder-Einschränkung) besagt, dass zu einem Zeitpunkt nur eine einzige Beziehung der durch *or* gekennzeichneten Assoziationen realisiert werden darf. Im deutschen Sprachraum finden wir anstelle des Begriffs *or* häufig den Begriff *oder*.
- ✓ Die Zusicherung *subset* (subset-Restriktion) bildet eine Abhängigkeitsbeziehung. Die abhängige Assoziation enthält eine Teilmenge der Objektverbindungen, die in der korrespondierenden Assoziation vereinbart sind.

Die beiden folgenden Beispiele verdeutlichen diesen Zusammenhang.

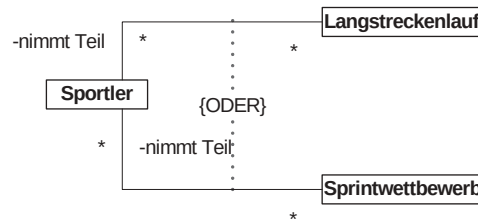


Abbildung 4.32: Zusicherung: »oder«

Ein Sportler nimmt entweder an Langstreckenläufen oder an Sprintwettbewerben teil, da bei einem Sportfest die Sprintwettbewerbe und die Langstreckenwettbewerbe gleichzeitig stattfinden. Das bedeutet, dass ein Sportler, der an einem Langstreckenlauf teilnimmt, nicht an einem Sprintwettbewerb teilnehmen kann und umgekehrt ein Sportler, der bei einem Sprintwettbewerb an den Start geht, zu Langstreckenläufen nicht zugelassen wird. Genau genommen handelt es sich aus logischer Sicht um eine Exklusiv-oder-Aussage, da das Vorhandensein einer Objektbeziehung die Objektbeziehungen der jeweils anderen Assoziation ausschließt.

Die Zusicherung subset verdeutlichen wir anhand einer Fußballmannschaft. Eine Fußballmannschaft besteht aus 11 Spielern. Im Klassendiagramm drücken wir das durch eine Assoziation zwischen den Klassen **Mannschaft** und **Fußballspieler** aus.

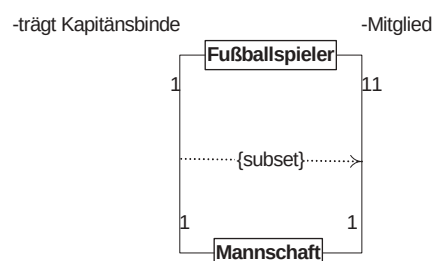


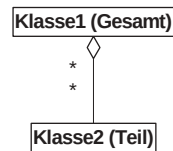
Abbildung 4.33: Zusicherung: »subset«

- ✓ Jeder Mannschaft ordnet die Assoziation Mitglied genau 11 Fußballspieler zu. Jeder Fußballspieler kann in der umgekehrten Sicht nur Mitglied genau einer Mannschaft sein.
- ✓ Eine weitere Assoziation, die die Klassen Fußballspieler und Mannschaft verbindet, ist die Kapitänsrolle, die in jeder Mannschaft ein Spieler wahrnimmt.
- ✓ Gäbe es die subset-Vereinbarung zwischen den Assoziationen Mitglied und Kapitän nicht, so wäre es möglich, dass ein Spieler von Bayer 04 Leverkusen als Kapitän des FC Bayern München fungiert. Da diese Zuordnung keinen Sinn ergibt, schränken wir die möglichen Kombinationen mit einer Zusicherung weiter ein.
- ✓ Die Subset-Zusicherung, die als gestrichelter Pfeil die Assoziation Kapitän mit der Assoziation Mitglied verbindet, besagt, dass die Objektverbindungen, die die Kapitäneigenschaft festlegen, auch zu der Menge der Mitgliedschaftszuordnungen gehören müssen. Mit anderen Worten: Kapitän kann nur sein, wer auch Mitglied einer Mannschaft ist.

Aggregation – Teile eines Ganzen

Eine *Aggregation* ist eine besondere Form der Assoziation. Dabei handelt es sich um eine fachliche Verbindung, in der sich ein Objekt aus einem oder mehreren anderen Objekten zusammensetzt. Folglich wird diese Beziehung häufig auch als *Teil-Ganzes-Beziehung*, *Gesamt-Teil-Struktur* oder *Whole-Part-Beziehung* bezeichnet.

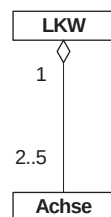
Grundsätzlich definiert eine Aggregation neben ihrer Semantik auch eine Rangordnung von Objekten. Die in der Hierarchie höheren Objekte setzen sich aus den Objekten mit niedrigerem Rang zusammen. Umgekehrt sind die in der Hierarchie niedrigeren Objekte Teil der Objekte mit höherem Rang. Die folgende Abbildung zeigt die UML-Darstellung einer Aggregation.

**Abbildung 4.34:** Aggregation: UML-Darstellung

Eine Aggregation verbindet, wie die einfache Assoziation, zwei Klassen miteinander, wobei sie in einer nicht ausgefüllten Raute endet. Dabei steht die Raute immer am Ende des Ganzen, das man auch Aggregat nennt. Die Teilobjekte nehmen in einer Aggregation Aufgaben des Aggregats wahr. Das bedeutet, dass das Aggregat Aufgaben an die Teile delegiert.

Alle Regeln der Assoziationen gelten auch für die Aggregation, da sie lediglich eine Spezialform der Assoziation ist. Die Kardinalitäten legen fest, aus wie vielen Teilen ein Aggregat besteht (Kardinalität neben dem Teil) bzw. zu wie vielen Aggregaten ein Objekt gehört (Kardinalität neben dem Aggregat).

Das folgende Beispiel verdeutlicht den Zusammenhang von Kardinalität und Aggregation.

**Abbildung 4.35:** Aggregation: LKW

Ein LKW besteht neben anderen Teilen auch aus mehreren Achsen.

- ✓ Die Raute an der Klasse LKW symbolisiert, dass die Klasse LKW das Aggregat (Ganzes) darstellt.
- ✓ Das andere Ende der Aggregation, die Klasse Achse, symbolisiert die Teile.
- ✓ Die Kardinalität auf Seiten der Teile-Klasse legt fest, dass ein LKW mindestens zwei, höchstens jedoch fünf Achsen besitzt. Damit werden über die Aggregation jedem LKW zwei bis fünf Achsen zugeordnet.
- ✓ Die Kardinalität eins, die neben dem Aggregat LKW notiert ist, legt fest, dass auf der anderen Seite eine Achse zu einem Zeitpunkt nur in genau einen LKW eingebaut sein kann.



Bei der Zuordnung von materiellen Gegenständen fällt es dem Menschen leicht, zwischen einer Assoziation und einer Aggregation zu unterscheiden. Sehr viel schwieriger empfinden wir die Unterscheidung zwischen Aggregation und Assoziation bei abstrakten Gegenständen. Bei abstrakten Gegenständen ist das subjektive Empfinden, ob es sich um eine Aggregation oder eine einfache Assoziation handelt, sehr unterschiedlich.

Mehrstufige Aggregation

Ähnlich wie bei den Vererbungsbeziehungen können wir auch mehrstufige Aggregationen bilden. Nehmen wir als Beispiel ein Buch, das aus mehreren Kapiteln besteht, die wiederum aus mehreren Abschnitten bestehen. Damit kann ein Teil-Objekt selbst wiederum Aggregat sein, das sich aus weiteren Objekten zusammensetzt.

Die UML-Notation dieser mehrstufigen Aggregation für ein Buch könnte folgendermaßen aussehen.

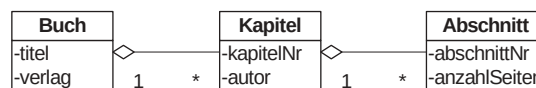


Abbildung 4.36: Aggregationshierarchie: Buch

Die Kardinalitäten könnten bei entsprechend sinnvoller Interpretation auch anders vereinbart werden. Das hängt vom Modellierungsziel ab.

- ✓ Die Objekte der Klasse Buch besitzen die Attribute `titel` und `verlag`.
- ✓ Ein Buch setzt sich aus mehreren Kapiteln zusammen. Die Attribute `kapitelNr` und `autor` charakterisieren jedes Kapitel.
- ✓ Eine Aggregation verbindet ein Kapitel-Objekt mit genau einem Buch-Objekt. Die Raute neben der Klasse Buch bestimmt die Klasse Buch als Aggregat- und die Klasse Kapitel als Teil-Klasse.
- ✓ Dabei wird ein Kapitel genau einem Buch zugeordnet und umgekehrt besitzt ein Buch beliebig viele Kapitel.
- ✓ Ein Kapitel setzt sich wiederum aus Abschnitten zusammen, die mit den Attributen `abschnittNr` und `anzahlSeiten` beschrieben sind.
- ✓ Die Aggregation, die die Klasse Abschnitt mit der Klasse Kapitel verbindet, legt fest dass Kapitel-Objekte als Aggregate beliebig viele Abschnitts-Objekte enthalten können. Jedes Abschnitts-Objekt ist dagegen Teil genau eines Kapitels.
- ✓ Ein Kapitel-Objekt ist damit gleichzeitig Teil eines Buch-Objektes und Aggregat, das als Teile Abschnitts-Objekte enthält.



Ob die Objekte einer Klasse Aggregate, Teile oder beides sind, ist nur in Beziehung zu den über eine Aggregationsbeziehung verbundenen Teilen festzulegen. Damit kann ein Objekt gleichzeitig Teil eines anderen Objektes und Aggregat sein, dass Teil-Objekte enthält.

Geteilte Aggregation

Bisher haben wir ausschließlich Teile betrachtet, die höchstens in einem Aggregat als Teil vorkamen. Es ist jedoch auch denkbar, dass ein Objekt Teil mehrerer Aggregate ist. Dabei kann es sein, dass die Aggregate Objekte derselben Klasse sind oder auch dass die Aggregate aus unterschiedlichen Klassen erzeugt wurden.

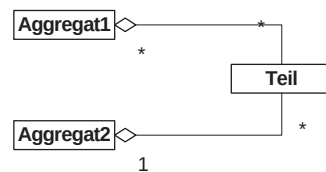


Abbildung 4.37: Geteilte Aggregation: UML-Darstellung

Die oben abgebildete UML-Notation zeigt beide Varianten in einer Grafik.

- ✓ Ein Teil-Objekt kann Teil mehrerer Objekte vom Typ Aggregat 1 sein.
- ✓ Darüber ist ein Teil-Objekt auch Teil eines Aggregat2-Objektes.

Mehrteilige Aggregation

In den vorangegangenen Abschnitten haben wir die verschiedenen Varianten betrachtet, die sich für Teile im Umfeld von Aggregationen ergeben. Auch ein Aggregat kann sich nicht nur aus einem oder mehreren Teil-Objekten einer einzigen Klasse zusammensetzen. Es ist ebenso denkbar, dass ein Aggregat-Objekt Teil-Objekte verschiedener Klassen zusammenfasst.

Besteht ein Objekt aus mehr als einem Teil-Objekt, so sprechen wir von einer mehrteiligen Aggregation. Auch hier besteht die Möglichkeit, dass die in einem Aggregat enthaltenen Objekte von der-

selben Klasse stammen, wie das Buch aus mehreren Abschnitten besteht oder aus mehreren Teil-Objekten verschiedener Klassen stammt.

Betrachten wir einen PKW, der aus verschiedenen Einzelteilen zusammengesetzt wird.

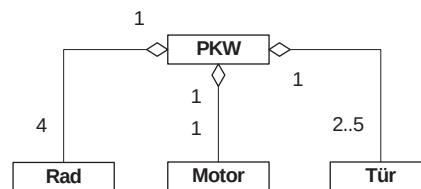


Abbildung 4.38: Mehrteilige Aggregation: PKW

Ein PKW besteht aus vier Rädern, wobei wir hier vom Ersatzrad einmal absehen. Diese Aggregation allein reicht schon aus, um die Beziehung als mehrteilig zu qualifizieren. In diesem Fall stellen die vier Räder mehrere Teil-Objekte derselben Klasse Rad dar.

Weiterhin setzt sich ein PKW aus einem Motor und zwei bis fünf Türen, je nach Bauart, zusammen. Auch die Tür-Objekte allein stellen bereits eine mehrteilige Aggregation dar. Insgesamt sind hier beide Formen der mehrteiligen Aggregation gegeben, da sowohl mehrere Elemente derselben Klasse als auch Objekte verschiedener Klassen als Teile im Aggregat PKW enthalten sind.

Eine alternative Betrachtung eines PKW, die von einem Autokonstrukteur stammen könnte, der Bauteilgruppen und Bauteile plant und realisiert, könnte eine mehrstufige und mehrteilige Aggregation ergeben.

- ✓ Der PKW setzt zunächst aus den Bauteilgruppen Chassis und Karosserie zusammen.

- ✓ Das Chassis besteht wiederum aus vier Rädern und einem Motor. Die Karosserie enthält Türen.
- ✓ Damit sind die Klassen Chassis und Karosserie sowohl Teile als auch Aggregate in einer mehrstufigen Aggregations-Hierarchie.

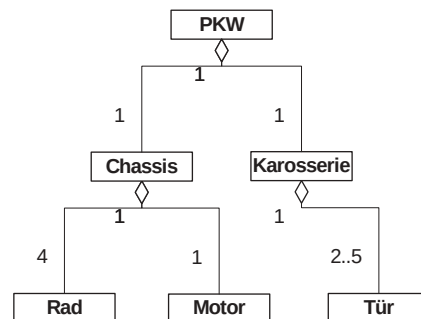


Abbildung 4.39: Mehrstufige Aggregation: PKW

Komposition – die strenge Aggregation

Die *Komposition* stellt eine spezielle Form der Aggregation dar. Somit gelten im Prinzip alle Regeln, die für die Aggregation vereinbart sind, auch für die Komposition, es sei denn, die Aggregation definiert etwas anderes.



Obwohl die Komposition einen semantischen Unterschied zur Aggregation aufweist, bezeichnen wir das Ganze wie in der objektorientierten Fachwelt üblich weiter als Aggregat, auch wenn es sinnvoll wäre, das Ganze in einer Kompositionsbeziehung als *Kompositum* zu bezeichnen.

Im Unterschied zu einer Aggregation sind in einer Kompositionsbeziehung die Teile vom Aggregat existenzabhängig. Das bedeutet, dass die Teile ohne das Aggregat nicht existieren können. Aus diesem semantischen Unterschied ergeben sich auch formale Unterschiede.

- ✓ Die Kardinalität auf Seiten des Aggregats beträgt in einer Komposition immer 1. Ein Teilobjekt gehört immer zu genau einem Aggregat.
- ✓ Die Teilobjekte sind abhängig von ihrem Aggregat. Existiert kein Aggregat, so existiert auch kein Teilobjekt.
- ✓ Umgekehrt kann es ein Aggregat geben, ohne dass zwingend die mittels Komposition zugeordneten Teile bestehen müssen. Die Teile können auch noch später hinzugefügt werden.
- ✓ Eine geteilte Aggregation ist nicht zulässig, da damit die Existenzabhängigkeit unterlaufen würde.
- ✓ Wird ein Aggregatobjekt vernichtet, muss zwingend auch das zugehörige Teilobjekt zerstört werden.
- ✓ Wird ein Aggregat kopiert, müssen auch die per Komposition zugeordneten Teilobjekte kopiert werden, da sonst eine nicht zulässige geteilte Aggregation der Teile mit Original und Kopie des Aggregats entstünden.

Diese Aspekte müssen insbesondere bei der Implementierung objektorientierter Modelle sauber umgesetzt werden.



In einer Komposition sind die Teile abhängig von ihrem Aggregat. Ohne das Ganze können die Teile nicht existieren. Das Aggregat kann aber umgekehrt sehr wohl ohne seine Teile bestehen.

Die Notation der Komposition entspricht der einer Aggregation, mit dem Unterschied, dass die Raute auf der Seite des Aggregats ausgefüllt ist.

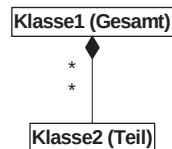


Abbildung 4.40: Komposition: UML-Darstellung

Kardinalitäten werden ebenfalls dargestellt. Beachten Sie, dass geteilte Kompositionen wegen der Existenzabhängigkeit nicht modelliert werden dürfen. Mehrstufige und mehrteilige Kompositionen stellen dagegen kein Problem dar.

Das folgende Beispiel stellt ein Gebäude dar, das sich aus Treppen und Zimmern zusammensetzt.

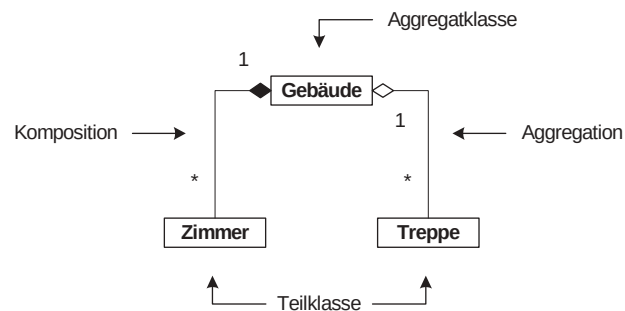


Abbildung 4.41: Komposition: Gebäude

- ✓ Zwischen der Klasse Gebäude und der Klasse Treppe besteht eine Aggregation.
- ✓ Ein Gebäude kann beliebig viele Treppen enthalten, muss aber nicht zwingend, wie im Fall eines ebenerdigen, einstöckigen Gebäudes ohne Keller, eine Treppe besitzen.

- ✓ Eine Treppe kann ohne ein Gebäude existieren. Sie kann im Freien stehen und zu einem höher gelegenen Platz oder zu einer Aussichtsplattform führen. Vielleicht hilft Ihnen auch die Vorstellung, dass eine in ein Gebäude eingebaute Wendeltreppe aus Holz dort wieder ausgebaut werden kann.
- ✓ Die Teilobjekte vom Typ Treppe sind somit nicht existenzabhängig von einem Gebäude.
- ✓ Eine Komposition verbindet Teilobjekte der Klasse Zimmer mit dem Aggregatobjekt der Klasse Gebäude.
- ✓ Mit dem Abriss eines Gebäudes ist zwingend auch die Zerstörung der in dem Gebäude enthaltenen Zimmer verbunden. Somit sind die Zimmer als Teilobjekte von dem Gebäude, in dem sie sich befinden, existenzabhängig.
- ✓ Diese existenzielle Beziehung kann nur gewährleistet sein, wenn die Kardinalität auf Seiten des Aggregats wie im Fall des Gebäudes 1 beträgt. Ein Zimmer kann niemals keinem oder mehreren Gebäuden zugeordnet werden.

Bei der Abbildung von Kompositionen materiell vorkommender Objekte wie Gebäuden und Zimmern fällt uns Menschen die Darstellung und auch die Einsicht in die Existenzabhängigkeit sehr leicht. Betrachten wir immaterielle Objekte, so bedarf es immer einer genauen Prüfung, ob eine Assoziation, Aggregation oder Komposition vorliegt.

Während die Unterscheidung zwischen den Alternativen Assoziation oder Aggregation eher akademischer Natur ist, die kaum ernst zu nehmende Konsequenzen nach sich zieht, ist die Unterscheidung zwischen Aggregation und Komposition insbesondere zur Vorbereitung der Implementierung sehr sorgfältig zu treffen. Dabei stehen die Ziele des Modells (das beabsichtigte Systemverhalten) im Vordergrund. Das subjektive Empfinden des Modellierers ist eher nachrangig als Entscheidungskriterium heranzuziehen.

Betrachten wir abschließend noch zwei Beispiele, die die Schwierigkeit der Entscheidungsfindung zwischen Aggregation und Komposition verdeutlichen.

Geht es darum, einen Auftrag zu den in ihm enthaltenen Auftragspositionen in Beziehung zu setzen, so ist es ziemlich eindeutig, dass es sich um eine Komposition handelt. Das Modellierungsziel ist hier eindeutig und auch aus verschiedenen Perspektiven mit den gleichen Zielen verbunden.

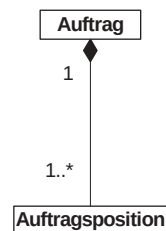


Abbildung 4.42: Komposition: Auftrag

Egal, ob Auftraggeber, Auftragnehmer, Manager oder Controller einen Auftrag aus ihrer speziellen Sicht beschreiben, für alle gilt, dass eine Auftragsposition Bestandteil genau eines Auftrags sein muss und dass eine Auftragsposition per Definition ohne Auftrag keinen Sinn ergibt. Aus der anderen Sicht ergibt auch ein Auftrag ohne Auftragsposition keinen Sinn, so dass zumindest eine Auftragsposition enthalten sein sollte.

Greifen wir den Zusammenhang des Buches auf, das sich aus Kapiteln und Abschnitten zusammensetzt, so werden Ihnen jetzt vielleicht erhebliche Zweifel kommen, ob es sich um eine Aggregation oder Komposition handelt.

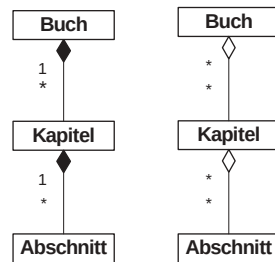


Abbildung 4.43: Aggregation oder Komposition: Buch

Der Leser eines Buches wird sagen, dass ein Kapitel ohne Buch für ihn überhaupt nicht denkbar ist. Genauso wenig kann er sich vorstellen, dass ein Abschnitt ohne Kapitel Sinn hat.

- ✓ Folglich modelliert er die Objekte der Klasse Abschnitt als existenzabhängige Teile der Klasse Kapitel und stellt sie somit als Komposition dar.
- ✓ Die Kapitel wiederum stellt er als existenzabhängige Elemente eines Buch-Objekts da.
- ✓ Zwingend müssen demnach die Kardinalitäten auf Seiten der Aggregate Kapitel und Buch 1 betragen.
- ✓ Zugestehen mag er vielleicht, dass Bücher nicht zwingend Kapitel und Kapitel nicht unbedingt Abschnitte enthalten müssen – vielleicht handelt es sich um einen Gedichtband – aber einen oder mehrere enthalten können. Die Kardinalität auf Seiten der Klassen Kapitel und Abschnitt ist somit beliebig (*).

Fragen wir den Roman- oder Fachbuch-Autor, argumentiert der wahrscheinlich ganz anders. Er sagt: »Wenn ich ein Buch schreibe, habe ich eine Idee, die ich zu Papier bringen möchte. Mit dem ersten Abschnitt fange ich an und irgendwann ist dann das erste Kapitel fertig. Erst wenn mehrere Kapitel fertig sind, überlege ich, ob

ich einen Verlag suche, der das Ganze als Buch verlegt. Die Hälfte all meiner Kapitel und Abschnitte liegt jedoch bis heute – und wahrscheinlich auch in Zukunft – unvollendet in der Schublade.«

- ✓ Folglich gilt für ihn die Eigenständigkeit seiner Arbeitsergebnisse in Form von Abschnitten und Kapiteln.
- ✓ Es besteht bei ihm immer die Absicht, Abschnitte zu Kapiteln zusammenzufassen und Kapitel in einem Buch zusammenzustellen und das Werk einem Verleger anzubieten. Also stellt er den Zusammenhang als Aggregation zwischen den Klassen Abschnitt und Kapitel sowie zwischen Kapitel und Buch dar.
- ✓ Da er oft Kapitel und Abschnitte, die schon in einem Buch veröffentlicht wurden, auch in andere Bücher integriert, kann aus seiner Sammlung ein Abschnitt mehreren Kapiteln oder ein Kapitel durchaus mehreren Büchern zugeordnet werden – gerade bei Autoren von Fachbüchern eine gängige Praxis. Folglich sind alle Kardinalitäten des Modells beliebig (*).

Wer hat nun Recht mit seiner Darstellung? Der Modellierer, der die Komposition vorzieht, oder derjenige, der eine Aggregation bevorzugt? Sie werden in jedem Projekt solche oder ähnliche Entscheidungen treffen müssen. Dabei geht jedoch weniger um die subjektive Einschätzung, die ein Modellierer hat, sondern um die Ziele, die mit der Modellierung verfolgt werden.

Geht es um eine Software, die Literaturstellen verwaltet, so werden dort Abschnitte immer in Kapitel und diese immer im Kontext von Büchern auftauchen. Die Komposition ist in diesem Fall die einzige sinnvolle Wahl.

Handelt es sich jedoch um eine Autorenumgebung, die den Autor von Fachbüchern unterstützt bei der Erstellung von Büchern und Artikeln, sollten Abschnitte und Kapitel als selbständige Einheiten verwaltet werden, so dass die Modellierung einer Aggregation zwischen Abschnitten, Kapiteln und Büchern geboten ist.



Leiten Sie die Assoziationsart (einfache Assoziation, Aggregation, Komposition) und die Kardinalitäten immer aus den Modellzielen ab, die sich aus den Anwendungszielen bzw. dem daraus abgeleiteten Systemverhalten ergeben.

Abschließend erweitern wir das Wecker-Beispiel um einige Elemente, die in Form von Assoziationen, Aggregationen und Kompositionen dem Modell hinzugefügt werden.

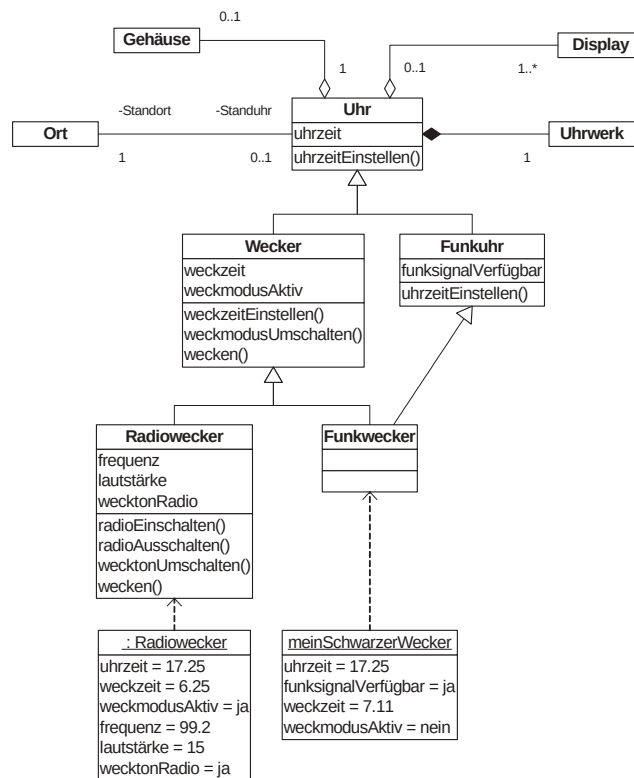


Abbildung 4.44: Assoziation: Wecker

Stellen Sie sich vor, dass das Wecker-Beispiel aus der Sicht eines Uhren- und Weckersammlers erweitert wird. Er stellt seine Uhren in mehreren Vitrinen in seinem Haus aus. Dabei interessiert er sich ausschließlich für funktionstüchtige Uhren. Folglich muss die Uhr ein Uhrwerk besitzen, und auch Display und Gehäuse als Teile, aus denen die Uhr zusammengesetzt ist, sind für unseren Sammler bedeutend.

- ✓ Jede Uhr der Sammlung wird an einem Ort in den Vitrinen des Sammlers ausgestellt. Diese Zuordnung stellen wir als einfache Assoziation zwischen den Klassen `Ort` und `Uhr` dar.
- ✓ Jede Uhr steht an einem Ort, aber nicht jeder Platz in den Vitrinen ist mit einer Uhr belegt.
- ✓ An den Assoziationsenden sind die Rollen, die die beteiligten Objekte in der Verbindung einnehmen, vermerkt. Ein `Ort` wird zum Standort und die `Uhr` zum Exponat.
- ✓ Ein Objekt der Klasse `Uhr` muss, um in die Sammlung aufgenommen zu werden, ein funktionsfähiges Uhrwerk besitzen. Jedes Uhrwerk ist in eine Uhr eingebaut und jede hier interessierende Uhr enthält genau ein Uhrwerk. Diese existenzielle Abhängigkeit stellt eine Komposition dar.
- ✓ Sammlerstücke sollten mindestens ein `Display` besitzen und umgekehrt kann ein `Display` in eine `Uhr` eingebaut sein. Es muß jedoch nicht eingebaut sein, da unser Sammler auch Ersatzdisplays in der Schublade liegen hat. Folglich ist dieser Zusammenhang als Aggregation modelliert. Auf der anderen Seite kann es auch sein, dass eine Uhr mehrere Anzeigen besitzt, wie es bei Uhren der Fall ist, die eine Zifferblattdarstellung mit Zeigern und zusätzlich eine Ziffernanzeige besitzen.
- ✓ In den Vitrinen stehen Uhren mit und ohne Gehäuse. Damit kann ein Uhrwerk in ein Gehäuse eingebaut sein, muss es aber nicht (0..1). Für unseren Sammler besitzen Gehäuse keinen Reiz. Fehlen sie, ist es ihm egal und er kümmert sich auch nicht um ihre Ersatzbeschaffung.

Assoziation oder Vererbung?

Einfachvererbung, Mehrfachvererbung sowie Assoziation, Aggregation und Komposition helfen den Betrachtern eines Klassendiagramms, die einzelnen Elemente auch mit Hilfe ihrer Beziehungen zu erkennen. Das Auswahlproblem des Modellierers bei der Wahl zwischen Assoziation, Aggregation und Komposition haben wir bereits diskutiert. Darüber hinaus besteht mit einigen Einschränkungen sogar die Möglichkeit, eine Vererbungsbeziehung durch eine Kombination aus Assoziation und Zugriffsoperationen zu ersetzen.

Betrachten wir wieder das Beispiel des Weckers. Der Funkwecker als Unterklasse steht in einer Mehrfachvererbungsbeziehung zu den Oberklassen Wecker und Funkuhr. Gesetzt den Fall, Sie möchten die Programmiersprache Java oder C# zur Umsetzung einer Verwaltungssoftware für unseren Uhrensammler einsetzen, so sind Sie gezwungen, ein auf die Einfachvererbung beschränktes Modell zu entwickeln.

Dass ein Funkwecker sowohl Eigenschaften eines Weckers als auch die einer Funkuhr besitzt, steht dabei außer Diskussion. Das Ziel der Umstrukturierung besteht in diesem Fall darin, die Mehrfachvererbung durch eine Kombination aus Einfachvererbung und Assoziation nachzubilden.

- ✓ Die Nachbildung funktioniert prinzipiell so, dass im Falle von zwei gemeinsamen Oberklassen eine Vererbungsbeziehung entfernt wird.
- ✓ Die Attribute und speziellen Operationen, die jetzt fehlen, werden in einer neu gebildeten zusätzlichen Klasse definiert.
- ✓ Die neu gebildete Klasse wird über eine Assoziation, Aggregation oder Komposition allen Klassen zugeordnet, deren Funktionalität durch Entfernen der Vererbungsbeziehung beschnitten wurde. Beachten Sie dabei, dass auch Beziehungen wie Assoziation, Aggregation und Komposition vererbt werden.

Im folgenden Beispiel wird die Vererbungsbeziehung von der Funkuhr zum Funkwecker durch die Klasse Funkempfänger ersetzt.

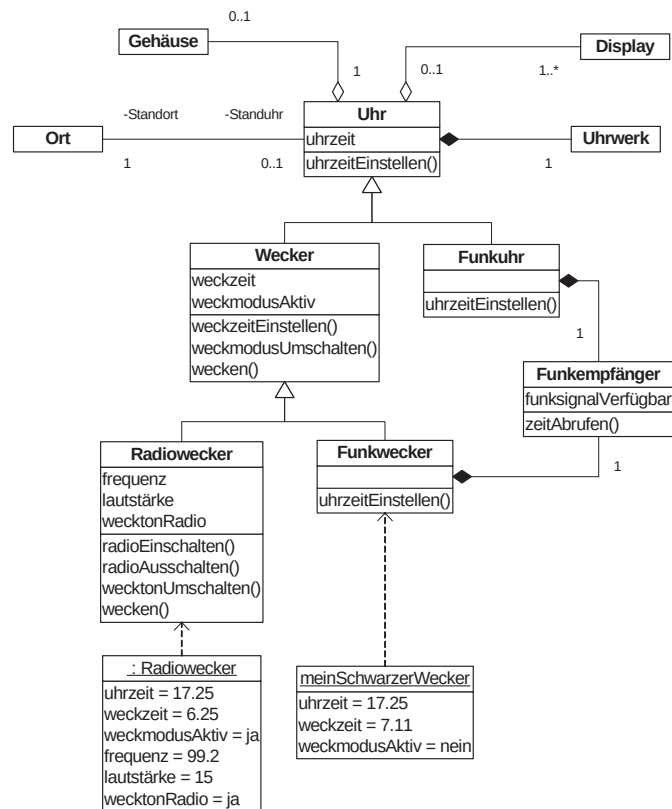


Abbildung 4.45: Assoziation statt Vererbung: Funkuhr

- ✓ Die Vererbungsbeziehung zwischen den Klassen Funkuhr und Funkwecker wird entfernt.
- ✓ Bilden Sie eine neue Klasse Funkempfänger. Dieser Funkempfänger erfüllt nun die Zusatzaufgaben, die ein Funkwecker zusätzlich zu einem gewöhnlichen Wecker besitzt. Er kann über ein anliegendes Funksignal die Zeit abrufen und im Attribut funksignalVerfügbar festhalten, ob ein Funksignal anliegt.

- ✓ Über eine Composition erhalten dann alle Uhren die Funkuhr-Funktionalität zurück, die Ihnen durch die gekappte Vererbungsbeziehung entzogen wurde. In diesem Fall stellt der Funkempfänger ein Bauteil dar, das in die Klasse Funkuhr und Funkwecker eingebaut wird.
- ✓ Damit der Wecker die Funktionalität nutzen kann, definiert er eine Funktion `uhrzeitEinstellen()`, die intern die Operation `zeitEinstellen()` des Funkempfängers aktiviert. Der Auftrag zur Zeitermittlung wird also an das Funkempfänger-Bauteil delegiert.
- ✓ Sollte kein Funksignal anliegen, so teilt die Operation `zeitEinstellen()` seinen erfolglosen Versuch der Funkuhr mit. Alternativ nutzt sie dann die in der Oberklasse Uhr definierte und überschriebene Operation `zeitEinstellen()`, die das »Zeiteinstellen von Hand« beschreibt.



Ziehen Sie eine Kombination aus Einfachvererbung und Assoziation mit einer zusätzlich gebildeten Klasse der Mehrfachvererbung vor, da die Mehrfachvererbung neben konzeptionellen Problemen nicht in allen Programmiersprachen verfügbar ist. Die weit verbreiteten Sprachen C# und Java unterstützen keine Mehrfachvererbung.

Pakete – Ordnung schaffen

Pakete dienen zur Gliederung und Strukturierung von Problemstellungen. Dabei sollte jedes Paket ein konsistentes (Teil-)Modell bilden. Das bedeutet, dass ein Paket immer nach bestimmten sinnvollen Gliederungskriterien zusammengefasste Elemente enthalten sollte und nicht wahllos zusammengewürfelte Klassen.

Vollständigkeit eines Paketes bedeutet dennoch keinesfalls, dass ein Paket isoliert von anderen Paketen zu sehen ist. In der Regel greifen Klassen eines Paketes auf Klassen eines anderen Paketes über Vererbungsbeziehungen und Assoziationen zu.

Klassennamen dürfen sich in verschiedenen Paketen wiederholen, da der vollständige qualifizierte Name einer Klasse sich aus dem Paketnamen und dem Klassennamen bildet. In der Kombination aus Paketname und Klassennamen ist eine Klasse dennoch eindeutig zu identifizieren.

Paketname::Klassename
-attribut1
-attribut2
+operation1()
+operation2()

Abbildung 4.46: Paketzugehörigkeit: UML-Darstellung

Die UML erlaubt mehrere Darstellungen der Paketzugehörigkeit. Die obige Abbildung zeigt die Notation eines vollständig qualifizierten Klassennamens, der aus dem Namen des Paketes, gefolgt von zwei Doppelpunkten und dem Klassennamen im ersten Segment – dem Namenssegment – einer Klasse gebildet wird.

Pakete eignen sich dazu, sehr komplexe Modelle mit einer unübersichtlichen Anzahl von Klassen – eine solche Unübersichtlichkeit ist bereits bei 30 Klassen erreicht – in mehrere Subsysteme zu zerlegen. Damit bleiben die Teilmodelle überschaubar. Wichtig ist in jedem Fall, durch die Organisation in Paketen logische Strukturen zu verdeutlichen und so das Verständnis des Modells dem Betrachter zu vereinfachen.

Dabei eignen sich sowohl fachliche als auch technische Kriterien zur Bildung von Subsystemen. In der Phase der fachlichen Analyse gebildete Pakete nach fachlichen Kriterien können in dem nachfolgenden Schritt der technischen Konstruktion in weitere Unterpakete, die Konstruktionsinformationen der Benutzungsoberfläche und der Datenhaltung unterscheiden, unterteilt werden.

Auch zwischen Paketen können Vererbungs- und Abhängigkeitsbeziehungen bestehen, die sich aus den Beziehungen ihrer Elemente ableiten. Erbt eine Klasse, die zu Paket3 gehört, von einer Klasse

aus Paket2, so gelten auch die Pakete als Ober- und Unterpaket, die in einer Vererbungsbeziehung stehen.

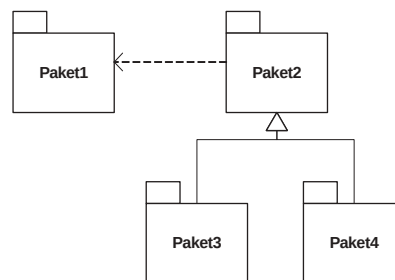


Abbildung 4.47: Abhängigkeiten zwischen Paketen: UML-Darstellung

Somit zeigt schon die Paketsicht, in welchen Paketen Vererbungsbeziehungen zwischen Klassen und Schnittstellen vorliegen.

Das Gleiche gilt auch für die gestrichelt dargestellten Abhängigkeiten, die der folgende Abschnitt gesondert behandelt.

Abhängigkeiten – Änderungen beherrschen

Eine weitere Form der Beziehung zwischen den Elementen eines objektorientierten Modells ist die *Abhängigkeit*. Abhängigkeit bedeutet dabei, dass sich eine Änderung des unabhängigen Elementes auf das abhängige Element auswirken kann, aber nicht zwingend auswirken muss. Aus konstruktiver Sicht ist der Modellierer angehalten, bei der Änderung eines unabhängigen Elementes immer alle abhängigen Elemente auf deren Kompatibilität zum geänderten Element zu prüfen.



Abhängigkeiten dienen dem Modellierer dazu, des möglichen Rattenschwanzes von Änderungen, die eine Änderung eines Modellelements nach sich zieht, Herr zu werden.

Die folgende Abbildung veranschaulicht die UML-Notation einer Abhängigkeit.

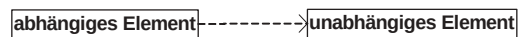


Abbildung 4.48: Abhängigkeit: UML-Darstellung

Dabei verbindet ein gestrichelter Pfeil immer ein abhängiges Element mit einem unabhängigen Element. Der Pfeil entspringt dem abhängigen Element und verweist auf das unabhängige Element.

Abhängigkeitsbeziehungen können zwischen verschiedenen Elementen eines Modells bestehen.

- ✓ Abhängigkeit zwischen zwei Klassen.
- ✓ Abhängigkeit zweier Pakete. Stehen zwei Klassen, die verschiedenen Paketen angehören in einer Abhängigkeitsbeziehung, so gelten auch die korrespondierenden Pakete als voneinander abhängig.
- ✓ Abhängigkeit einer Operation von einer Klasse.
- ✓ Abhängigkeit einer Klasse von einer Schnittstelle.



Beachten Sie, dass sich Abhängigkeiten immer auf die tatsächlich genannten Elemente beziehen. Hängt eine Klasse von einer anderen ab, so gilt die Abhängigkeit aus konstruktiver Sicht für die Klassendefinition. Sie gilt jedoch nicht für die Exemplar-Objekte dieser Klassen aus inhaltlicher Sicht.

Die UML stellt einige *Stereotypen* zur Klassifikation von Abhängigkeitsbeziehungen zur Verfügung.

- ✓ *Access* bezeichnet eine Abhängigkeitsbeziehung zweier Klassen, die es der abhängigen Klasse erlaubt, auf die öffentlichen Elemente der unabhängigen Klasse zuzugreifen.

- ✓ *Friend* definiert eine Abhängigkeit zwischen einer Operation, einer Klasse oder einem Paket und einem Element eines anderen Paketes. Dabei hat das abhängige Element ungeachtet der Sichtbarkeitsdefinition auf die Eigenschaften des unabhängigen Elementes Zugriff.
- ✓ *Import* erlaubt dem abhängigen Element einer Abhängigkeitsbeziehung die Verwendung unabhängiger Elemente in einem fremden Paket. Der Geltungsbereich eines Paketes wird also um die unabhängigen Elemente erweitert. Damit werden die unabhängigen Elemente dem Geltungsbereich des abhängigen Elementes hinzugefügt.

Die folgende Abbildung zeigt die UML-Darstellung der Abhängigkeits-Stereotypen.

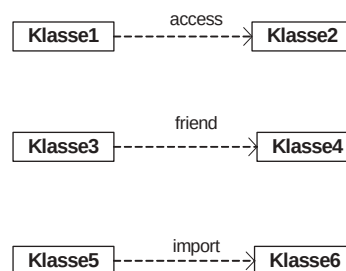


Abbildung 4.49: Abhängigkeit zwischen Klassen

Die Stereotypen sind oberhalb des gestrichelten Abhängigkeitspfeils angeordnet und konkretisieren damit die Art der Abhängigkeit der mit dem Pfeil verbundenen Elemente.

Die Abhängigkeit einer Operation von einer Klasse stellt die nächste Abbildung wie folgt dar.



Abbildung 4.50: Abhängigkeit Klasse und Operation

Der gestrichelte Pfeil entspringt direkt im Namen der abhängigen Operation (`operation2()`) und verweist auf die unabhängige Klasse (`Klasse1`).

Die Abhängigkeit einer Klasse von einer Schnittstelle zeigt die folgende Darstellung.

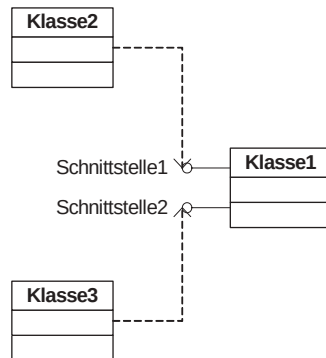


Abbildung 4.51: Schnittstellen: Abhängigkeitsbeziehung

Ein Kreis, den eine Linie mit der zugehörigen Klasse verbindet, gilt als Schnittstellensymbol. Der Abhängigkeitspfeil entspringt einer abhängigen Klasse und verweist auf die unabhängige Schnittstelle.

In der Abbildung verweist das abhängige Element `Klasse1` auf die unabhängige Schnittstelle1. `Klasse3` ist abhängig von Schnitt-

stelle2. Beide Schnittstellen gehören zu Klasse2. Sie sind dort implementiert.

Zwischen Paketen bestehen die Abhängigkeiten, die auch zwischen den in den Paketen enthaltenen Elementen bestehen. Das bedeutet, dass die Abhängigkeiten, die zwischen den Elementen – also Klassen und Schnittstellen – eines Paketes bestehen, automatisch auf die Ebene der Pakete übertragen werden.

Die folgende Abbildung zeigt diesen Zusammenhang exemplarisch an zwei Paketen, die insgesamt fünf Klassen enthalten, auf.

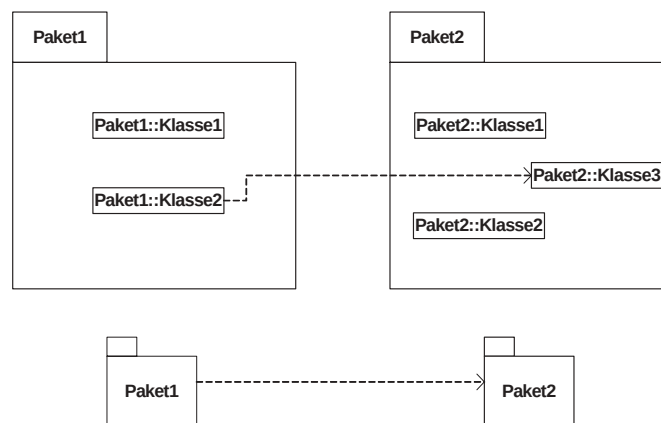


Abbildung 4.52: Klassen- und Paketabhängigkeiten: UML-Darstellung

Paket1 enthält die Klassen Klasse1 und Klasse2, Paket2 die Klassen Klasse1, Klasse2 und Klasse3. Die Unterscheidung der Klassen gleichen Namens ist hier durch die Zugehörigkeit zum jeweiligen Paket gewährleistet, so dass sich die vollständig qualifizierten Namen voneinander unterscheiden.

Der obere Teil der Abbildung zeigt, dass Klasse2 aus Paket1 (Paket1::Klasse2) von Klasse3 aus Paket2 (Paket2::Klasse3) ab-

hängig ist. Daraus leitet sich automatisch auch eine Abhängigkeitsbeziehung zwischen `Paket1` und `Paket2` ab. Diese Abhängigkeit stellt der untere Teil der Abbildung dar.

Auf Paketebene besagen die Abhängigkeiten, dass in einem Paket unabhängige Elemente existieren, deren Änderung sich auf Elemente eines anderen Paketes auswirken könnten. So sind strukturelle Abhängigkeiten bereits in der Paketübersicht erkennbar. Konkret muss der Entwickler im abgebildeten Fall immer die Elemente im Paket überprüfen. Falls `Paket2::Klasse3` geändert werden sollte, muss er die Auswirkungen auf `Paket1::Klasse2` untersuchen.



Abhängigkeitsbeziehungen zwischen Elementen verschiedener Pakete erschweren den modularen Aufbau eines Modells. Auch für Pakete gilt das aus der Softwaretechnik bekannte Konzept der schmalen Kopplung zwischen Modulen, das möglichst wenige Abhängigkeiten zwischen Modulen - hier Paketen - fordert. Sind Pakete stark durch Abhängigkeitsbeziehungen miteinander verwoben. Die Wiederverwendung eines Paketes ist erschwert und Änderungen in einem Paket führen zu einem Rattenschwanz von Änderungen in weiteren Paketen. Die Pakete verlieren dadurch Ihre Eigenständigkeit. Bilden Sie daher Ihre Pakete so, dass möglichst wenige Abhängigkeiten zwischen den Paketen entstehen.

Statik: Implementierungsdiagramme

Klassen- und Objektdiagramme repräsentieren eine logische Sicht auf die Zusammenhänge einer Anwendung. Sie enthalten keinerlei Informationen über die Verteilung der modellierten Strukturen auf Software-Komponenten. Auch eine Zuordnung von Software-Komponenten auf Arbeitsrechner, Server bleibt dort unbestimmt.

In diese Lücke springen die Implementierungsdiagramme und legen die physische Struktur einer Anwendung aus der Sicht von Hardware- und Software-Komponenten fest. Sie zeigen insbeson-

dere deren Verteilung und Abhängigkeiten. Im Vergleich zu der logischen Sicht, sind die Implementierungsdiagramme eher von untergeordneter Bedeutung in der UML.

Komponentendiagramm – realisierte Einheiten

Die UML stellt zwei verschiedene Diagrammtypen zur Verfügung, die jedoch sinnvoll kombiniert werden können.

- ✓ Das Komponentendiagramm beschreibt eigenständige Software-Einheiten und deren Zusammenspiel. Eine Komponente bezeichnet man als wieder verwendbaren Programmcode, der eine abgegrenzte Aufgabe erfüllt. Häufig werden die auf der logischen Ebene definierten Pakete in ähnlicher Zusammensetzung in Komponenten überführt.
- ✓ Die häufig anzutreffende Kongruenz von Paketen und Komponenten ergibt sich fast zwangsläufig, wenn Pakete nach Kriterien der fachlichen Zusammengehörigkeit zur Lösung von (Teil-) Aufgaben gebildet werden.
- ✓ Komponenten betonen im Gegensatz zu Paketen die softwaretechnischen Abhängigkeiten, die jedoch wie bereits gesehen häufig mit den fachlichen Abhängigkeiten korrespondieren.
- ✓ Komponenten sind von außen betrachtet Einheiten, die zur Anbindung anderer Komponenten Schnittstellen besitzen.
- ✓ Komponentendiagramme stellen Komponenten mit ihren Schnittstellen grafisch dar und zeigen darüber hinaus auch deren Abhängigkeiten.

Die folgende Abbildung zeigt das Symbol für eine Komponente (siehe Abbildung 4.53).

Komponentendiagramme werden auch dazu genutzt, statische Abhängigkeiten, wie beispielsweise Compiler-Abhängigkeiten zwischen Programmen, darzustellen.

Abhängigkeiten zwischen Komponenten stellen die bereits erläuterten gestrichelten Pfeile dar. Auch hier zeigt die abhängige Komponente auf die unabhängige Komponente.



Abbildung 4.53: Komponente: UML-Darstellung

Die folgende Abbildung zeigt die UML-Notation der Abhängigkeit von Komponenten.

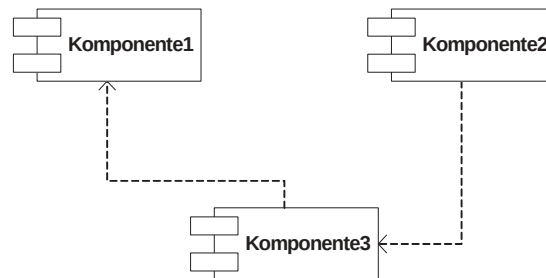


Abbildung 4.54: Abhängigkeiten von Komponenten: UML-Darstellung

- ✓ Dabei hängt die Komponente2 von Komponente3 ab und Komponente3 wiederum ist abhängig von Komponente1.
- ✓ Komponente1 ist eine unabhängige Komponente, da sie von keiner weiteren Komponente abhängt.
- ✓ Komponente2 ist eine abhängige Komponente, die bei Änderungen in Komponente3 überprüft werden muss.
- ✓ Komponente3 ist abhängig in Bezug auf Komponente1 und unabhängig in Bezug auf Komponente1.
- ✓ Eine Änderung von Komponente1 kann sich folglich über Komponente3 bis auf Komponente2 durchschlagen.

Komponenten können zusätzlich weitere Elemente wie Objekte, andere Komponenten und Knoten enthalten. Aus inhaltlicher

Sicht kann eine Komponente konkretisiert und in ihren Details genauer beschrieben werden. Dazu können die im weiteren Verlauf vorgestellten Anwendungsfall-Diagramme oder auch Klassendiagramme eingesetzt werden.

Die technischen Aspekte der Implementierung werden hingegen in Verteilungsdiagrammen beschrieben.

Verteilungsdiagramme – Träger von Komponenten

Verteilungsdiagramme enthalten die Konfiguration einer Hard- und Software-Umgebung. Sie zeigen, welche Komponenten auf welchen Knoten verarbeitet werden und was für Kommunikationsbeziehungen zwischen den Knoten bestehen. Knoten können in diesem Kontext alle physischen Objekte wie Computer, Speicher oder Prozessoren sein.

Darüber hinaus können auch bei der Geschäftsprozessmodellierung Mitarbeiter und Organisationseinheiten als Aufgabenträger einer Aufgabe in den Kontext einbezogen werden. Als Komponenten spielen dann neben Software-Komponenten auch Vorgehensweisen und Dokumente, die zur Bewältigung von Aufgaben herangezogen werden, eine Rolle.

Die Abbildung 4.55 zeigt ein Verteilungsdiagramm, das auch Software-Komponenten mit einbezieht.

Die abgebildete Systemlandschaft zeigt die physische Verteilung der folgenden Komponenten:

- ✓ Einen dedizierten Datenbankserver (DBServer1), der eine Kundendatenbank enthält;
- ✓ Einen zweiten Datenbankserver (DBServer2) zur Verwaltung der Lagerhaltung;
- ✓ einen Anwendungsserver (AppServer), der Software-Komponenten für den Vertrieb bereitstellt;
- ✓ einen Arbeitsplatzrechner (Client1), der mit dem Betriebssystem Windows 2000 ausgestattet ist;

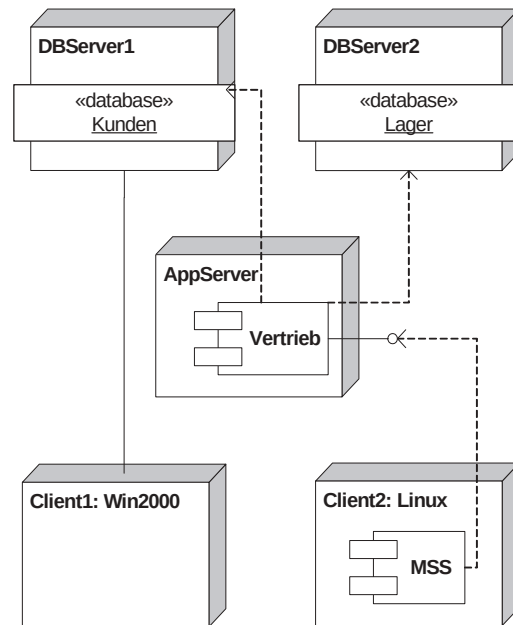


Abbildung 4.55: Verteilungsdiagramm mit Komponenten

- ✓ sowie einen Arbeitsplatzrechner (Client2) mit Linux-Betriebssystem, das als Software-Komponente ein Management Support System (MSS) bereitstellt.

Zwischen den Knoten und Komponenten bestehen die folgenden Abhängigkeiten und Verbindungen:

- ✓ AppServer ist abhängig von den beiden Datenbankservern DBServer1 und DBServer2, die ihm Daten über die Kunden und das Warenlager zur Verfügung stellen.
- ✓ Das Management Support System greift als Komponente von von Client2 auf die Dienste des Anwendungsservers (AppServer) zu.

- ✓ Sollte die Schnittstelle der Vertriebskomponente geändert werden, so muss auch eine Anpassung des MSS geprüft werden.
- ✓ Zwischen Client1 und DBServer1 besteht eine einfache Kommunikationsbeziehung. Der Windows2000-Rechner dient als Zugangsterminal, um Kundendaten zu bearbeiten und auszuwerten.

Abschließend sei noch angemerkt, dass die Implementierungsdiagramme in der UML ein Schattendasein führen. Sie werden im Vergleich zu den Diagrammen der logischen Innensicht eher selten eingesetzt.

Dynamik: Verhaltensdiagramme

Dynamische Modelle repräsentieren das Verhalten der Objekte in einem System. Aus diesem Grund werden sie auch als *Verhaltensmodelle* bezeichnet. Aufgabe der dynamischen Modelle ist es, die Methoden, Zustände (bzw. Zustandsänderungen), Interaktionen und die Kooperation von Objekten abzubilden.



Dynamische Sachverhalte innerhalb eines objektorientierten Modells werden durch die *Verhaltensdiagramme* dargestellt.

Die UML kennt vier verschiedene Arten von Verhaltensdiagrammen, deren unterschiedliche Betrachtungsweisen sich zu einem objektorientierten Gesamtbild des analysierten Problembereichs ergänzen:

- ✓ *Sequenzdiagramme* veranschaulichen die Beziehungen zwischen Objekten sowie deren Austausch von Botschaften im Zeitablauf.
- ✓ *Kollaborationsdiagramme* beschreiben Objekte mit ihren Verbindungen untereinander.
- ✓ *Zustandsdiagramme* zeigen verschiedene Zustände von Modell-elementen und die Übergänge zwischen den Zuständen.

- ✓ *Aktivitätsdiagramme* sind spezielle Zustandsdiagramme. Sie zeigen Aktivitäten und Zustände von Objekten sowie die Zustandsübergänge.



Sequenz- und Kollaborationsdiagramme dienen gemeinsam der Verdeutlichung von Interaktionen zwischen Objekten. Sie werden daher auch unter dem Oberbegriff *Interaktionsdiagramme* zusammengefasst.

Sequenz- und Kollaborationsdiagramme basieren auf den gleichen Informationen über Elemente der Problemdomäne. Sie betonen jedoch unterschiedliche Aspekte. Es handelt sich jeweils um die Darstellung von Interaktionen – jedoch aus unterschiedlichen Perspektiven.

Sequenzdiagramme

Die *Sequenzdiagramme* bilden neben den Kollaborationsdiagrammen eine Form der Interaktionsdiagramme.



Sequenzdiagramme veranschaulichen Interaktionen zwischen Instanzen im Zeitverlauf.

Die grafische Darstellung besteht in der Vertikalen aus einer zeitlichen Dimension. In der Horizontalen werden die an der Interaktion beteiligten Objekte aufgeführt. Die Dimensionen können bei Bedarf jedoch auch vertauscht werden.

Objekte werden in Sequenzdiagrammen durch ein Rechteck mit ihrem Namen dargestellt. Sie erhalten zusätzlich eine senkrechte gestrichelte Linie, die den Zeitablauf verdeutlicht. Die Linie symbolisiert die Lebensdauer eines Objekts von seiner Entstehung bis zu seinem Entfernen (*Objektlebenszyklus*).



Sequenzdiagramme zeigen in der Regel nicht die Kommunikation zwischen konkreten Objekten. Vielmehr handelt es sich um Repräsentanten beliebiger Objekte einer bestimmten Klasse bzw. um deren Rollen.

Senkrechte Balken, die eine der gestrichelten Linien überlagern, geben an, wann ein Objekt während des betrachteten Zeitraums aktiv ist. Sie werden auch als *Steuerungsfokus* bezeichnet.

Die Aktivitäten der Objekte äußern sich in der Ausführung von Methoden, die durch Nachrichten (Stimuli) eines Objekts an ein anderes aktiviert werden. Hat ein Objekt eine Methode beendet, dann sendet es eine Nachricht als Antwort an das aufrufende Objekt. Ein Stimulus wird dabei durch einen durchgezogenen Pfeil dargestellt, die Antwort-Nachricht durch einen gestrichelten Pfeil.



Die Reihenfolge der Objekte in einem Sequenzdiagramm spielt keine Rolle. Sie ist beliebig und kann so gewählt werden, dass sich eine möglichst übersichtliche Darstellung ergibt.

Die folgende Abbildung 4.56 zeigt die genannten Bestandteile eines Sequenzdiagramms in einer schematischen Darstellung.

Anhand des Objekt1 lässt sich der Lebenszyklus eines Objekts erkennen. In der UML wird das Erzeugen eines Objekts durch einen Nachrichtenpfeil auf den Objektkörper (das Rechteck mit seinem Namen) symbolisiert. Objekt1 wird in der obigen Abbildung erzeugt (*Objektkonstruktion*), indem ein Akteur die Nachricht1 sendet.

Die beiden übrigen Objekte (bzw. Rollen), mit denen Objekt1 in der obigen Abbildung kommuniziert, existieren bereits vor der Entstehung von Objekt1. Sie sind in der senkrechten Zeitachse über (zeitlich also vor) Objekt1 angeordnet.

Objekt2 und Objekt3 existieren nicht nur bereits vor Objekt1, sie bleiben auch nach dessen Ausscheiden aus dem Modell weiter bestehen (erkennbar an der weitergehenden Lebenslinie von Objekt2 und Objekt3).

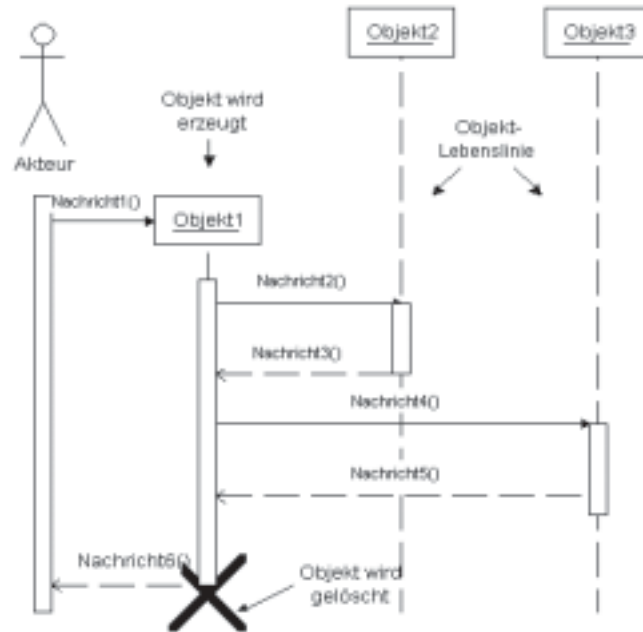


Abbildung 4.56: Sequenzdiagramm

In der Abbildung sendet Objekt1 nach seiner Entstehung zunächst eine Nachricht (Nachricht2) an Objekt2, das diese verarbeitet und eine Antwort (Nachricht3 mit gestrichelter Linie) an Objekt1 zurücksendet. Dieser Vorgang bedeutet nichts anderes als den Aufruf einer Methode von Objekt2 durch Objekt1.

Eine weitere Kommunikationsbeziehung besteht zwischen Objekt1 und Objekt3, bevor Objekt1 schließlich gelöscht wird. Das große »X« am Ende der Lebenslinie von Objekt1 symbolisiert dessen Entfernung (*Objektdestruktion*).



Endet ein Nachrichtenpfeil in dem »X«, das die Objektdestruktion symbolisiert, dann löst die entsprechende Nachricht die Zerstörung des Objekts aus.

Die Ausführung von Operationen und das Senden von Nachrichten durch Objekte können an Bedingungen geknüpft sein. Auch für die Wiederholung von Nachrichten und Operationen sind in der UML Sprachelemente vorgesehen.

Bedingungen werden in eckigen Klammern vor den Namen der Nachricht geschrieben. Die entsprechende Nachricht wird nur dann gesendet, wenn die angegebene Bedingung erfüllt ist. Das mehrfache Senden einer Nachricht wird als *Iteration* bezeichnet. Sie wird durch ein »*« vor dem Namen der Nachricht bzw. vor der Bedingung symbolisiert.

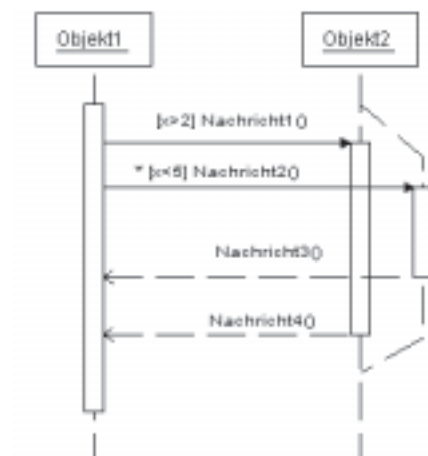


Abbildung 4.57: Wiederholungen und Bedingungen in Sequenzdiagrammen

Im Beispiel der obigen Abbildung wird Nachricht1 nur dann gesendet, wenn die Bedingung $x > 2$ erfüllt ist. Nachricht2 wird so lange wiederholt, wie die Bedingung $x < 5$ gilt.

Die Abbildung zeigt außerdem die Notation für die gleichzeitige Ausführung zweier Methoden durch ein Objekt. Die Lebenslinie von Objekt2 ist geteilt. Vorausgesetzt, die genannten Bedingungen sind erfüllt, beginnt Objekt2 nach Empfang der Nachricht1 zunächst mit der Ausführung der entsprechenden Methode. Während diese noch bearbeitet wird, beginnt die Verarbeitung der zweiten Methode, ausgelöst durch Nachricht2. Das Ergebnis der zweiten Methode wird als Antwort an Objekt1 gesendet (Nachricht3). Schließlich ist auch die erste Methode beendet und die letzte Nachricht (Nachricht4) geht an Objekt1.

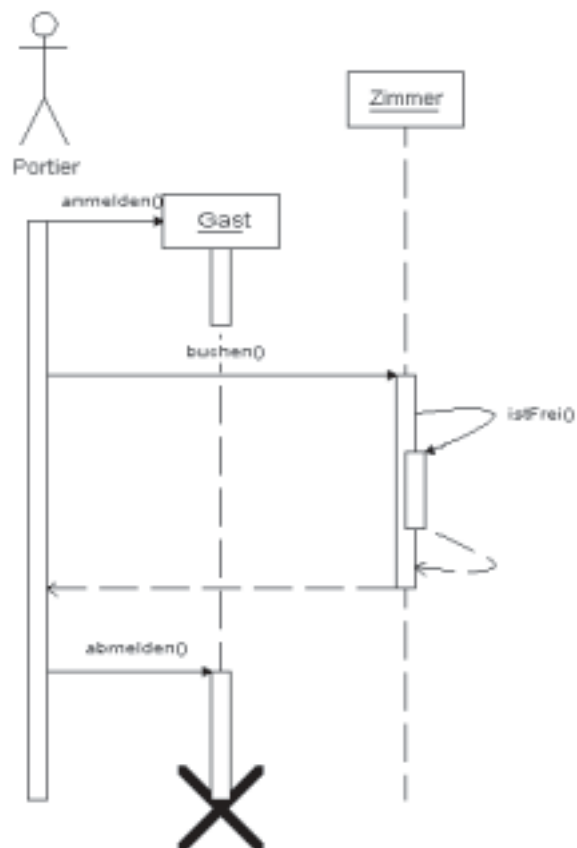


Sequenzdiagramm und Klassendiagramm eines Objektmodells müssen stets übereinstimmen. Ein Sequenzdiagramm darf daher nur solche Nachrichten enthalten, die als Methoden in den Klassenbeschreibungen des Klassendiagramms enthalten sind.

Ein weiteres Beispiel für ein Sequenzdiagramm zeigt Abbildung 4.58.

Der Akteur Portier stößt die Buchung eines Zimmers durch einen Gast an. Dazu muss sich der Gast zunächst anmelden. Anschließend wird ein Zimmer gebucht. Dazu führt das Objekt Zimmer seine eigene Methode `istFrei` aus, um die Verfügbarkeit für den gewünschten Zeitraum zu überprüfen. Wenn der Gast wieder auszieht, wird er vom Portier abgemeldet und das entsprechende Objekt gelöscht.

Dem Sequenzdiagramm sind die unterschiedlichen Lebenszyklen der Objekte (bzw. Rollen) Gast und Zimmer zu entnehmen. Das Objekt Zimmer existiert bereits vor dem hier betrachteten Zeitraum. Ein neues Objekt Gast wird erst innerhalb des hier betrachteten Zeitraums mit der Anmeldung eines neuen Gastes erzeugt.

**Abbildung 4.58:** Sequenzdiagramm Hotelbuchung

Nachdem der Gast abgereist ist, wird in dem hier betrachteten Kontext (Sichtweise auf die Problemdomäne »Hotel«) das Objekt Gast nicht länger benötigt. Es kann also gelöscht werden. Das Objekt Zimmer besteht jedoch weiterhin, auch über den im Sequenzdiagramm abgebildeten Zeitraum hinaus.

Da ein Sequenzdiagramm nur solche Nachrichten enthalten kann, die als Methoden in den Klassenbeschreibungen der Klassendiagramme enthalten sind, müssen die beteiligten Klassen zumindest die Methoden anmelden, buchen und istFrei enthalten, wie in der folgenden Abbildung dargestellt.



Abbildung 4.59: Klassendiagramm Hotelbuchung

Die Klasse **Gast** enthält die Methoden anmelden und abmelden. Zu der Klasse **Zimmer** gehören die Methoden buchen und istFrei. Zwischen den beiden Klassen besteht eine binäre Assoziation. In dem Beispiel wird angenommen, dass jeder Gast genau ein Zimmer bucht und umgekehrt jedes Zimmer zu einem Gast gehört. Sollen für ein Zimmer auch Buchungen verschiedener Gäste im Voraus angenommen werden oder soll ein Gast mehrere Zimmer buchen können, dann müssen die Kardinalitäten entsprechend angepasst werden.

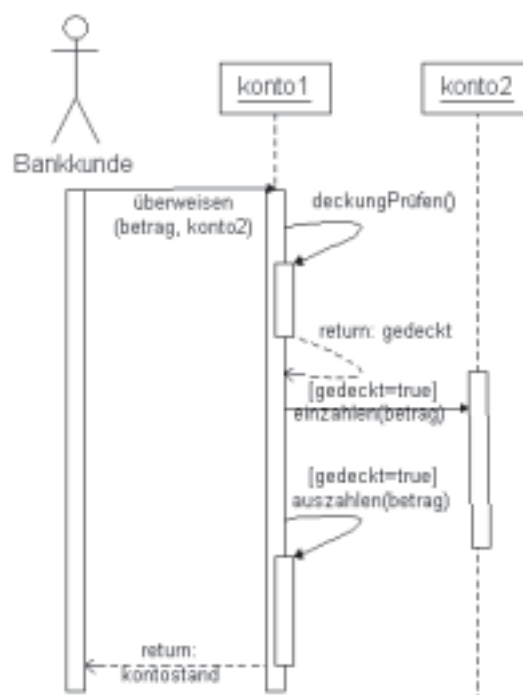
Zur Verdeutlichung zeigt das folgende Beispiel die Elemente von Sequenzdiagrammen noch einmal anhand der Überweisung von Girokonten.

Eine vereinfachte Klasse **Konto** ist in der folgenden Abbildung dargestellt.

Die Klasse **Konto** beschreibt ein Muster für Konten-Objekte, die eine Variable `kontostand` als Attribut sowie die Operationen `einzahlen`, `auszahlen`, `überweisen` und `prüfeDeckung` besitzen. Für eine Überweisung wird der zu überweisende Betrag sowie das Zielkonto, auf das überwiesen werden soll, benötigt. Einzahlungen und Auszahlungen erfordern jeweils nur eine Angabe über den gewünschten Betrag. Diese Werte werden den Operationen als Parameter übergeben.

**Abbildung 4.60:** Vereinfachte Klasse Konto

Das folgende Sequenzdiagramm zeigt die Vorgänge zwischen den beteiligten Konten-Objekten, wenn ein Bankkunde eine Überweisung vornimmt.

**Abbildung 4.61:** Sequenzdiagramm Überweisung

Der Kunde möchte einen bestimmten Betrag von `konto1` an `konto2` überweisen. Er ruft dazu die Operation `überweisen` auf. Als Parameter werden der Operation der zu überweisende Betrag und `konto2` als Zielkonto übergeben.

Das Objekt `konto1` überprüft nun mit seiner Operation `deckungPrüfen()`, ob es für den gewünschten Betrag überhaupt eine ausreichende Deckung besitzt. Das Ergebnis dieser Operation (`gedeckt`) ist ein Wert vom Datentyp Boolean (wahr oder falsch bzw. `true` oder `false`). Anhand des Balkens für den Steuerungsfokus im Sequenzdiagramm können Sie erkennen, dass die Operation `deckungPrüfen()` aus der Operation `überweisen` heraus aufgerufen wird.

Je nach Ergebnis der Deckungsprüfung werden weitere Operationen ausgeführt. Hat `gedeckt` den Wert `true`, so ruft `konto1` die Operation `einzahlen` des Objekts `konto2` auf und übergibt ihr als Parameter den zu überweisenden Betrag. Anschließend wird, wiederum abhängig vom Wahrheitswert der Variablen `gedeckt`, aus der Operation `überweisen` heraus die Operation `auszahlen` von `konto1` aktiviert. Die Operation `überweisen` wird abgeschlossen durch die Rückgabe des aktualisierten Kontostands von `konto1` an den Kunden.

Die Überweisung eines bestimmten Betrags von einem Konto auf ein anderes wird in dem Beispiel in zwei Schritte aufgeteilt. Vorausgesetzt, das überweisende Konto besitzt eine ausreichende Deckung, wird der zu überweisende Betrag dem einen Konto in einer Einzahlung gutgeschrieben und dem anderen Konto in einer Auszahlung belastet.

Die Nachrichten des Sequenzdiagramms sind die Operationen der beteiligten Objekte. Dementsprechend finden Sie alle Nachrichten in der Klasse `Konto` als Operationen wieder.

Wir wollen nun die Konzepte des Sequenzdiagramms auf unser Wecker-Beispiel übertragen. Die Sichtweise beschränkt sich dabei auf die reine Weckfunktion. Mit Hilfe eines Sequenzdiagramms soll verdeutlicht werden, welche Abläufe im Rahmen der Weckfunktion im Objektmodell zu beobachten sind.

Für die Weckfunktion sind die beiden Attribute `weckzeit` und `weckstatus` sowie die Operationen `weckstatusUmschalten`, `weckzeitEinstellen` und `wecken` relevant. Die übrigen Eigenschaften des Weckers sind an der Weckfunktion nicht direkt beteiligt.

Das entsprechende Sequenzdiagramm könnte dann etwa wie folgt aussehen.

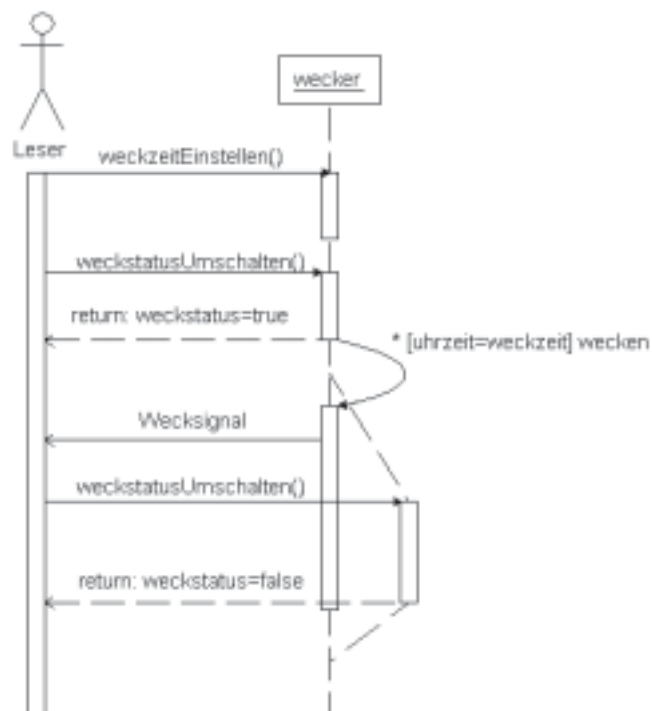


Abbildung 4.62: Sequenzdiagramm Weckfunktion

Der Leser (Akteur) stellt zunächst die Weckzeit ein, indem er die Operation `weckzeitEinstellen` des Objekts `wecker` aufruft. An-

schließlich muss er noch die Weckfunktion aktivieren. Dazu benutzt er die Operation `weckstatusUmschalten`, die das Attribut `weckstatus` mit dem Wert `true` belegt. Zur Bestätigung erhält der Akteur vom Wecker eine Rückmeldung, die beispielsweise in einer optischen Anzeige besteht.

Während der Akteur nun sanft vor sich hin schlummert, prüft die Operation `weckstatusUmschalten` ständig, ob die eingestellte Weckzeit mit der aktuellen Uhrzeit übereinstimmt (Bedingung `[uhrzeit=weckzeit]`). Im Sequenzdiagramm ist die ständige Wiederholung dieser Bedingungsprüfung erkennbar an dem »*« vor der Bedingung der Operation `wecken`. Sobald die Weckzeit erreicht ist, wird die Operation `wecken` aktiviert, die dafür sorgt, dass der Akteur ein Wecksignal (z.B. einen Alarmton) erhält.

Durch erneuten Aufruf der Operation `weckstatusUmschalten` kann der Akteur das Wecksignal beenden und die Weckfunktion ausschalten. Er erhält wiederum eine Bestätigung durch die Rückmeldung über den aktuellen Weckstatus (jetzt: `weckstatus=false`).

Kollaborationsdiagramme

Eine weitere Möglichkeit, Interaktionen in der UML darzustellen, bilden die Kollaborationsdiagramme.

Sie definieren die Verbindungen zwischen einer Menge von Objekten in einer bestimmten Situation. Mit Hilfe einer hierarchischen Nummerierung können die Reihenfolge und die Schachtelung der Verbindungen durch Operationen angegeben werden. Somit sind auch Kollaborationsdiagramme in der Lage, zeitliche Abläufe zu veranschaulichen. Sie sind nicht so sehr zur präzisen Abbildung von Vorgängen geeignet wie Sequenzdiagramme, bieten jedoch eine Alternative, um Interaktionen zwischen Objekten komprimiert zu verdeutlichen.

Die zeitliche Reihenfolge der Nachrichten bzw. Operationen wird durch Ziffern angegeben. Diese so genannten *Sequenznummern* können entfallen, wenn die Abfolge für die betrachtete Situation nicht relevant ist. Ausgangspunkt der Aktionen ist stets eine Nachricht

von außerhalb der betrachteten Situation. Sie wird als *Start-Nachricht* bezeichnet und häufig durch einen Akteur ausgelöst. Die schematische Darstellung eines typischen Kollaborationsdiagramms zeigt die folgende Abbildung.

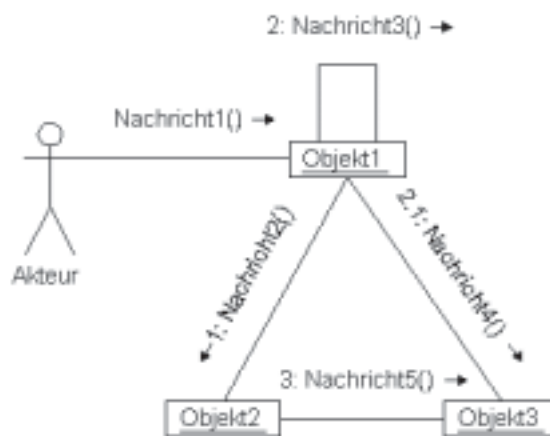


Abbildung 4.63: Kollaborationsdiagramm

Der Akteur startet den Ablauf, indem er die Nachricht1 (Start-Nachricht) sendet. Dadurch wird Objekt1 erzeugt. Objekt1 aktiviert über Nachricht2 eine Methode von Objekt2 und führt anschließend mit Nachricht3 eine eigene Methode aus. Diese wiederum sendet Nachricht4 an Objekt3. Nachricht5 an Objekt3 beendet den Vorgang.

Die Reihenfolge der zu sendenden Nachrichten und der dadurch ausgelösten Methoden wird mit den Sequenznummern vor den Nachrichten festgelegt. Eine hierarchische Beziehung findet sich in der obigen Abbildung zwischen den Methoden, die von Nachricht3 und Nachricht4 ausgelöst werden, da Letztere aus Ersterer aufgerufen wird.

Auch Kollaborationsdiagramme können Nachrichten mit Bedingungen verknüpfen.

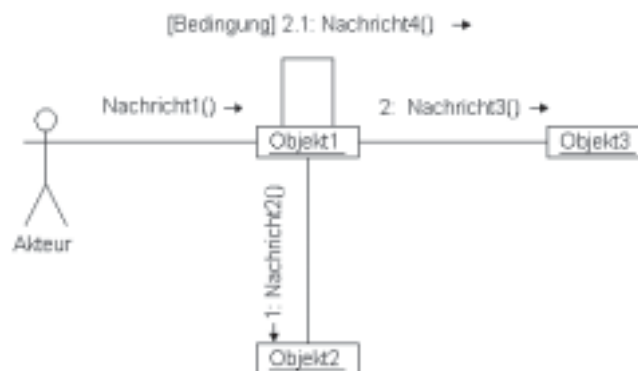


Abbildung 4.64: Bedingungen in Kollaborationsdiagrammen

Die Abbildung zeigt das Kollaborationsdiagramm aus der vorhergehenden Abbildung in leicht veränderter Form. Die Elemente 2 und 2.1 (Nachrichten 3 und 4) haben die Hierarchiestufen getauscht. Schritt 2.1 ist zudem mit einer Bedingung verknüpft. Nur wenn diese Bedingung erfüllt ist, wird die Nachricht gesendet bzw. die entsprechende Methode aktiviert.

Verschiedene Arten von Pfeilen symbolisieren unterschiedliche Kommunikationsbeziehungen in einem Kollaborationsdiagramm. Sie dienen insbesondere der Darstellung spezieller *Synchronisationsbedingungen*. Abbildung 4.65 enthält eine Auswahl dieser Pfeilarten.

- ✓ Der *Prozeduraufruf* wird durch einen durchgehenden Pfeil mit einer ausgefüllten Spitze symbolisiert. Ein Objekt ruft in dieser Kommunikationsbeziehung eine Methode eines anderen Objekts auf.
- ✓ Der Sender einer *sequentiellen Nachricht* fährt mit seinen eigenen Operationen erst dann fort, wenn der Empfänger die erhaltene Nachricht vollständig verarbeitet hat.

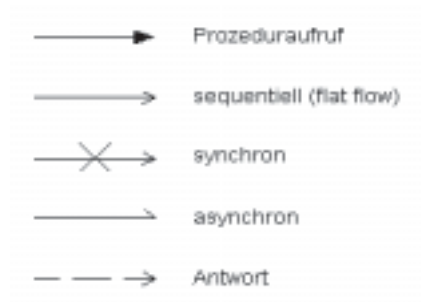


Abbildung 4.65: Synchronisationsbedingungen

- ✓ Bei der *synchronen Nachricht* wartet der Sender mit seinen eigenen Operationen, bis der Empfänger die Nachricht angenommen hat.
- ✓ Die *asynchrone Nachricht* führt nicht zu einer Unterbrechung der Operationen des Senders. Wann sie vom Empfänger angenommen wird, spielt keine Rolle.
- ✓ Der gestrichelte Pfeil symbolisiert die *Antwort auf einen Prozeduraufruf*.

Neben den genannten Synchronisationsbedingungen können auch erweiterte Variationen eingesetzt werden. So gibt es beispielsweise *eingeschränkte Nachrichten* (Abbruch, falls Nachricht nicht sofort angenommen wird) und *zeitabhängige Nachrichten* (Abbruch, falls Nachricht nicht innerhalb eines bestimmten Zeitraums angenommen wird).

Die UML enthält einige Standard-Zusicherungen, die anzeigen, ob ein Element oder eine Verbindung während des Ablaufs einer Interaktion erzeugt oder gelöscht wird:

- ✓ Objekte und Verbindungen, die während einer Interaktion erzeugt werden, erhalten die Kennzeichnung *{new}*.
- ✓ Objekte und Verbindungen, die während einer Interaktion gelöscht werden, sind mit der Zusicherung *{destroyed}* markiert.

- ✓ Objekte und Verbindungen, die während einer Interaktion zunächst erzeugt und später wieder gelöscht werden, haben die Bezeichnung *{transient}*.

Diese Zusicherungen werden als Text an das entsprechende Element geschrieben.

Außerdem kann ein Kollaborationsdiagramm Informationen über die Ursache für eine Beziehung zwischen Objekten enthalten. Auf der Seite des Objekts, das eine Nachricht erhält, wird dann einer der folgenden Stereotypen notiert, um die Ursache zu kennzeichnen:

- ✓ Den Standardfall bildet der Stereotyp *«association»*. Die Objektbeziehung ist eine Assoziation, eine Aggregation oder eine Komposition. Diese Beziehung ist implizit anzunehmen, wenn keine nähere Bezeichnung notiert wird.
- ✓ *«global»*: Das Empfänger-Objekt einer Nachricht ist in der beschriebenen Situation global.
- ✓ *«local»*: Der Empfänger ist ein lokales Objekt der sendenden Methode.
- ✓ *«parameter»*: Der Empfänger ist ein Parameter der sendenden Methode.
- ✓ *«self»*: Das Objekt sendet eine Nachricht an sich selbst, ist also gleichzeitig Sender und Empfänger der Nachricht.

Wie bei den Sequenzdiagrammen gilt auch hier, dass es sich bei den aufgeführten Elementen nicht unbedingt um konkrete Objekte handelt. Sie stehen lediglich stellvertretend für eine Rolle bzw. eine Menge von Objekten, die in diese Rolle schlüpfen können. Eine Interaktion in einem Kollaborationsdiagramm ist somit eine Spezifikation der Kommunikationsmuster zwischen Rollen. Es wird definiert, welche Nachrichten gesendet werden, welche Methoden aufgerufen werden und welche Rollen Sender- und Empfängerobjekt im Rahmen der Interaktion spielen.

Anhand des Konten-Beispiels haben wir Ihnen bereits einige Konzepte der UML verdeutlicht. Die folgende Abbildung zeigt die Vorgänge bei der Überweisung von einem Konto auf ein anderes, dargestellt mit Hilfe eines Kollaborationsdiagramms.

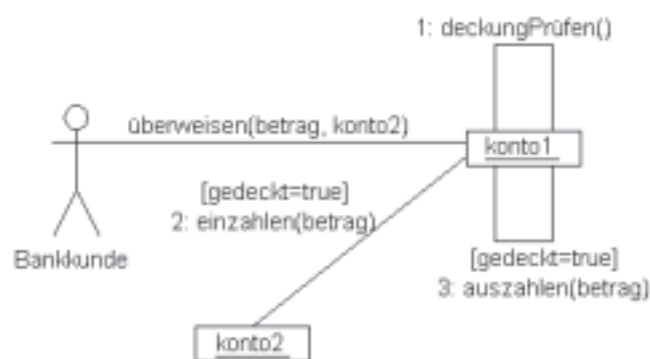


Abbildung 4.66: Kollaborationsdiagramm einer Überweisung

Die Abbildung enthält den Akteur Bankkunde, der die Überweisung durch den Aufruf der Operation `überweisen` mit den Parametern `betrag` und `konto2` in Gang setzt. `überweisen` ist eine Operation des Objekts `konto1`. Die weitere Vorgehensweise der beteiligten Objekte ist der Nummerierung der übrigen Operationen zu entnehmen. Das Objekt `konto1` führt die Operation `deckungPrüfen` aus. Je nach Ergebnis dieser Operation (Wert der booleschen Variable `gedeckt` ist `true` oder `false`) wird die Operation `einzahlen` des Objekts `konto2` mit dem Parameter `betrag` aufgerufen und die eigene Operation `auszahlen` ausgeführt – wiederum mit dem Parameter `betrag`.

Kollaborationsdiagramme zeigen die Zusammenarbeit von Objekten. Daher ist ein Kollaborationsdiagramm für unser Wecker-Beispiel wenig sinnvoll. Es würde allein das Objekt `wecker` enthalten. Eine Zusammenarbeit zwischen verschiedenen Objekten findet in diesem Anwendungsfall nicht statt.

Zustandsdiagramme

Den Begriff »Zustand« haben Sie bereits kennen gelernt. Er drückt die statischen Eigenschaften eines Objekts aus und wird mit Hilfe von Attributen beschrieben. Zustandsdiagramme ermöglichen es Ihnen, die verschiedenen Zustände, die ein Objekt in einem Programm oder in einem Geschäftsprozess einnehmen kann, zu verfolgen.



Ein Zustandsdiagramm dient der Darstellung der Zustände eines Objekts in seinem Lebenszyklus sowie der Abbildung von Zustandsänderungen und auslösenden Ereignissen.

Von Interesse sind hier also zum einen die verschiedenen Zustände selbst und zum anderen die Ereignisse, die zu einer Zustandsänderung führen.

Zur Abbildung von Zuständen und ihren Änderungen betrachtet die UML einen so genannten *Zustandsautomaten*, der aus Zuständen und Zustandsübergängen (Transitionen) besteht.



Da der Zustand eines Objekts durch seine Attribute beschrieben wird, bestimmen die *Attributwerte* seinen jeweils aktuellen Zustand.

Der Zustand bleibt gleich, solange kein Ereignis auf das Objekt einwirkt, das eine Zustandsänderung hervorruft.

Ein Zustand wird immer durch ein Ereignis ausgelöst und bleibt so lange bestehen, bis er durch ein neues Ereignis verändert wird. So durchläuft ein Objekt in seinem Lebenszyklus nacheinander verschiedene Zustände. Jeder dieser Zustände ist an einen bestimmten Zeitpunkt gebunden. Das Objekt besitzt also zu jedem Zeitpunkt genau einen Zustand.

Elemente von Zustandsdiagrammen

Die grafischen Elemente der UML zur Darstellung von Zustandsdiagrammen sind relativ einfach. Die Zustände selbst werden durch abgerundete Rechtecke, Zustandsübergänge durch Pfeile symbolisiert. Ein Pfeil (Zustandsübergang) verbindet jeweils zwei Rechtecke (Zustände).



Abbildung 4.67: Zustände und Zustandsübergang



Es sollte nicht jede einzelne Änderung eines Attributwertes gleich als eigener Zustand in ein Zustandsdiagramm aufgenommen werden. Nur solche Zustände bzw. Zustandsübergänge, die eine für die betrachtete Problemstellung wirklich wichtige Bedeutung haben, sollten im Modell abgebildet werden.

Jedes Zustandsdiagramm enthält einen *Start-* (oder Anfangs-) und einen *Endzustand*, um auch in diesem Verhaltensdiagramm Beginn und Ende einer Dynamik erkennen zu können. Die Symbole für Start- und Endzustände zeigt die folgende Abbildung.



Abbildung 4.68: Symbole für Start- und Endzustände



Start- und Endzustände werden als *Pseudozustände* bezeichnet, da Objekte diese Zustände nicht wirklich einnehmen können.

Neu erzeugte Objekte befinden sich zunächst in dem auf den Startzustand folgenden Zustand. Der Endzustand kennzeichnet das Ende des Lebenszyklus eines Objekts. Ein Zustandsdiagramm enthält immer einen Startzustand und kann einen Endzustand enthalten.

Zustände bestehen ähnlich wie Objekte und Klassen aus den drei (optionalen) Elementen:

- ✓ Name,
- ✓ Attribute (Zustandsvariable) und
- ✓ Ereignisse, Bedingungen und Operationen.

Dementsprechend wird das Zustandssymbol bei Bedarf in diese drei Segmente unterteilt, wie die folgende Abbildung zeigt.



Abbildung 4.69: Elemente des Zustandssymbols



Ein Zustand muss nicht zwangsläufig einen Namen besitzen. Es ist auch möglich, namenlose Zustände in ein Diagramm aufzunehmen. Diese heißen dann *anonyme Zustände*. Mehrere anonyme Zustände in einem Zustandsdiagramm sind alle voneinander verschieden. Andererseits kann ein und derselbe Zustand mehrfach in ein Zustandsdiagramm aufgenommen werden (mehrere Zustandssymbole mit demselben Namen); etwa um die Lesbarkeit des Diagramms zu verbessern. Diese Zustände sind dann alle identisch.



Für die Bezeichnung von Zuständen sollten möglichst keine Verben verwendet werden, um zu verdeutlichen, dass es sich bei dem Zustand selbst nicht um eine dynamische Komponente handelt. Gleichwohl kann ein Zustand mit Aktionen oder Aktivitäten verbunden werden.

Zustandsübergänge

Jeder Zustand kann mit Aktionen verbunden sein, die das Objekt, während es sich in diesem Zustand befindet, ausführt. Dabei werden drei Arten von Aktionen unterschieden:

- ✓ *Entry-Aktionen* werden automatisch beim Eintritt in einen Zustand ausgelöst. Dabei spielt es keine Rolle, durch welchen Zustandsübergang der Eintritt in den Zustand erfolgt. Die Entry-Aktion wird stets unabhängig vom vorhergehenden Zustand ausgeführt und selbständig beendet.
- ✓ *Exit-Aktionen* werden beim Verlassen eines Zustands automatisch ausgeführt.
- ✓ *Do-Aktionen* werden ständig wiederholt, solange ein Zustand aktiv ist.
- ✓ *Include-Aktionen* rufen externe Subroutinen auf.

Ein typisches Zustandsdiagramm, das die gerade beschriebenen grafischen Elemente enthält, zeigt Abbildung 4.70.

Das schematische Zustandsdiagramm der Abbildung lässt sich wie folgt interpretieren:

Solange der erste Zustand (Zustand1) Bestand hat, führt das Objekt eine Aktivität durch. Ereignis1 löst dann Zustand2 aus, bei dessen Eintritt die Aktion1 (Entry-Aktion) ausgeführt wird. Das Verlassen von Zustand2 führt zu Aktion2 (Exit-Aktion). Zustand3, der mit keinerlei Aktivitäten oder Aktionen verbunden ist, wird durch ein implizites Ereignis erreicht. Ereignis2 überführt das betrachtete Objekt in Zustand4. Ereignis3 zieht Aktion3 nach sich, führt jedoch zu keiner Zustandsänderung.

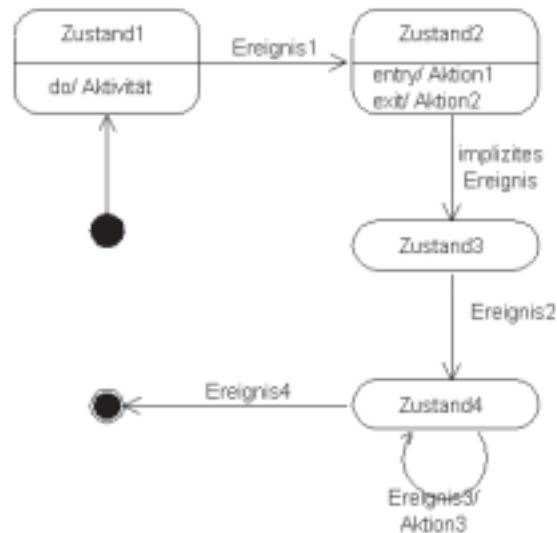


Abbildung 4.70: Zustandsdiagramm

Das Objekt verharrt in Zustand4. Erst mit Eintreten von Ereignis4 erreicht das Objekt seinen Endzustand und wird gelöscht – der Lebenszyklus des Objekts ist beendet.

Ein Zustandsübergang verbindet zwei Zustände miteinander. Er wird stets durch ein *Ereignis* ausgelöst. Für einen Zustandsübergang können verschiedene Arten von Ereignissen verantwortlich sein:

- ✓ eine Nachricht (Methodenaufruf);
- ✓ eine Bedingung (Ausdruck vom Typ boolean); jedes Mal wenn der Wert der Bedingung von falsch nach wahr wechselt, wird ein Zustandsübergang ausgelöst;
- ✓ das Erreichen eines bestimmten Zeitpunkts;
- ✓ der Ablauf einer bestimmten Zeitspanne (so genannter *elapsed time event*).

Bedingungen werden mit dem Schlüsselwort *when*, gefolgt von einem Boolean-Ausdruck formuliert (z.B. *when (x=5)*). Für elapsed time events gibt es das Schlüsselwort *after*. Ein gültiger Ausdruck zur Definition einer Zeitspanne lautet etwa *after (10 Sekunden)* oder *after (2 Minuten nach Beendigung von Zustand1)*.



Ein Zustandsübergang ist auch ohne explizite Angabe eines Ereignisses möglich. Er wird dann ausgelöst, wenn die mit einem Zustand verbundenen Aktivitäten abgeschlossen sind. Man spricht in diesem Fall von einem *impliziten Ereignis*.

Auch mit dem Zustandsübergang selbst kann eine Aktion verbunden sein, die ausgeführt wird, sobald der Übergang eingeleitet wird. Insofern gleicht sie einer Exit-Aktion.

Die beiden folgenden Abbildungen veranschaulichen die beschriebenen Komponenten von Zustandsdiagrammen anhand des Beispiels des Objekts *telefon*.

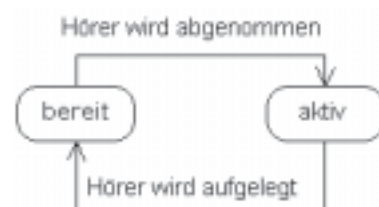


Abbildung 4.71: Zustandsdiagramm des Objekts *telefon*.

Das Telefon besitzt die beiden Zustände *bereit* und *aktiv*. Die Übergänge zwischen diesen beiden Zuständen werden durch zwei Ereignisse ausgelöst. Vom Zustand *bereit* wechselt das Telefon in den Zustand *aktiv*, wenn der Hörer abgenommen wird. Umgekehrt ändert sich der Zustand von *aktiv* in *bereit*, wenn der Hörer wieder aufgelegt wird.

Während der Benutzung des Telefons treten weitere Zustände auf, die sich durch den Vorgang des Telefonierens ergeben. Es handelt sich um Unterzustände des Zustands *aktiv*. Die folgende Abbildung ist in abgewandelter Form der UML-Spezifikation der OMG entnommen (vgl. die Quellenhinweise im Anhang). Sie zeigt das Verhalten des Objekts *telefon*.

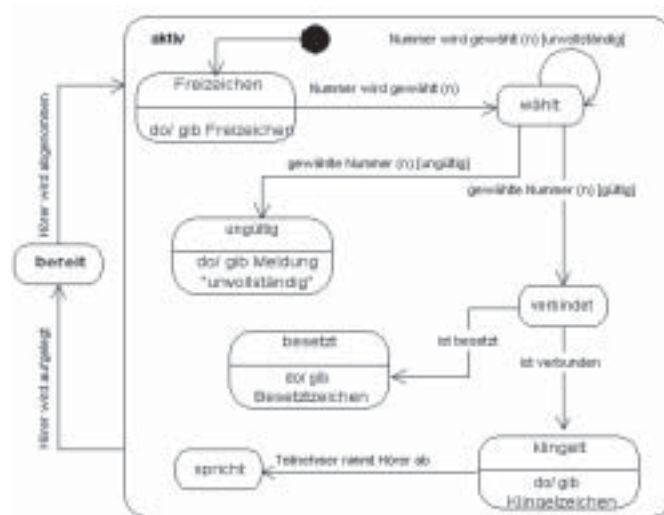


Abbildung 4.72: Detailliertes Zustandsdiagramm des Objekts *telefon*

Der Zustand *bereit* ist gegenüber der vorhergehenden Abbildung unverändert geblieben. Der große Kasten auf der rechten Seite symbolisiert den Zustand *aktiv*, der nun in seine Unterzustände zerlegt ist.

Der Übergang vom Zustand *bereit* zum Zustand *aktiv* wird ausgelöst durch das Ereignis *Hörer wird abgehoben*. Innerhalb von *aktiv* heißt der erste Zustand *Freizeichen*. Hier wird die Aktivität *gib*

Freizeichen ausgeführt. Nach Abheben des Hörers ertönt somit ein Freizeichen. Das Telefon bleibt in diesem Zustand, bis eine Nummer gewählt wird. Sobald eine Ziffer gedrückt wurde, wechselt das Telefon in den Zustand wählt. Nach jeder weiteren Ziffer wird geprüft, ob die Nummer bereits vollständig ist. Wenn nicht, verharret das Objekt in diesem Zustand.

Erst bei vollständiger Nummer erfolgt in Abhängigkeit von einer Bedingung ein Zustandsübergang. Ist die gewählte Nummer ungültig, dann nimmt das Telefon den Zustand ungültig an und gibt eine entsprechende Meldung aus. Nur bei vollständiger und gültiger Nummer wechselt das Objekt in den Zustand verbindet. Der Anwender (Akteur) wird mit dem gewünschten Teilnehmer verbunden.

Von diesem Zustand ausgehend gibt es nun wiederum zwei Ereignisse, die zu einer Zustandsänderung führen können. Entweder es ist besetzt, dann wird eine Meldung ausgegeben, oder die Verbindung ist möglich und das Telefon klingelt (Aktion: Ausgabe eines Klingelzeichens). Sobald der Angerufene seinerseits den Hörer abhebt, ist die Verbindung hergestellt und es kann gesprochen werden. Durch Auflegen des Hörers wird schließlich wieder der Zustand bereit erreicht.



Beachten Sie, dass alle in dem Zustandsdiagramm aufgeführten Aktionen und Aktivitäten als Methoden in dem Objekt `telefon` hinterlegt sein müssen. Das Zustandsdiagramm findet seine Entsprechung in dem zugehörigen Klassendiagramm.

Untenzustände

In dem Beispiel des Telefons wurden die Unterzustände bereits eingeführt. Dazu nun noch einige nähere Betrachtungen.

Untenzustände dienen der Verfeinerung eines zu beschreibenden Zustands in einem Verhaltensdiagramm. Ein Objekt kann dabei stets nur genau einen der Unterzustände annehmen – sie schließen

sich gegenseitig aus. Die folgende Abbildung zeigt eine grafische Darstellung des Zusammenhangs.

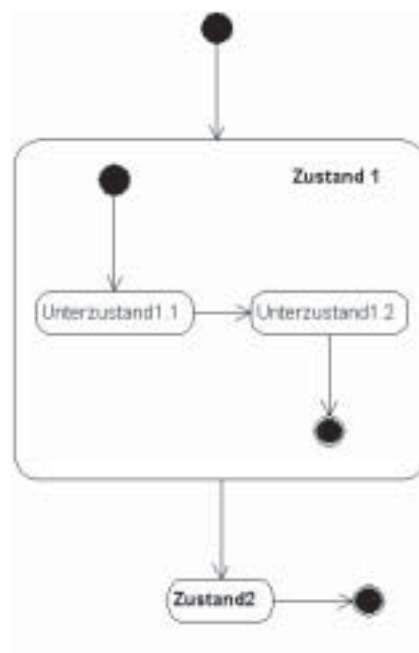


Abbildung 4.73: Verfeinerung durch Unterzustände (sequentiell)

Zustand1 wird in der obigen Abbildung verfeinert durch die Zustände Unterzustand1.1 und Unterzustand1.2. Sowohl das gesamte Zustandsdiagramm mit den Zuständen Zustand1 und Zustand2 als auch das verfeinerte Teildiagramm von Zustand1 besitzen einen Anfangs- und einen Endzustand.

Neben dieser sequentiellen Verschachtelung von Unterzuständen bietet die UML auch die Möglichkeit, eine parallele Verschachte-

lung von Unterzuständen zu modellieren, wie in der folgenden Abbildung zu sehen ist.

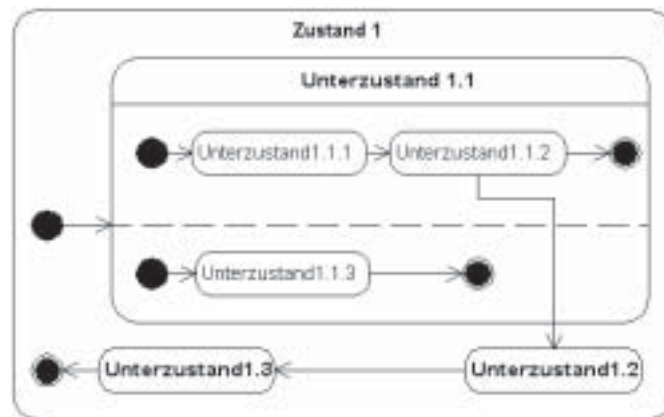


Abbildung 4.74: Verfeinerung durch Unterzustände (parallel)

Zustand1 besitzt in der Abbildung die drei Unterzustände 1.1, 1.2 und 1.3. Unterzustand1.1 kann wiederum die Formen Unterzustand1.1.1, Unterzustand1.1.2 und Unterzustand1.1.3 annehmen. Dabei sind die Unterzustände 1.1.1 und 1.1.2 einerseits und der Unterzustand 1.1.3 andererseits parallele Unterzustände. Es ist also entweder das eine oder das andere Verhalten zu beobachten. Die Ereignisse, die zu einem Zustandsübergang führen, sind in der Abbildung nicht aufgeführt. Unterzustand1.1.2 führt je nach Ereignis zu einem Endzustand oder über Unterzustand1.2 zu Unterzustand1.3.

Mit Hilfe von Zustandsdiagrammen lassen sich auch wiederholte Vorgänge abbilden. Dazu wird ein Rücksprung zu einem vorherigen Zustand modelliert. Das folgende Beispiel enthält einen solchen Rücksprung.

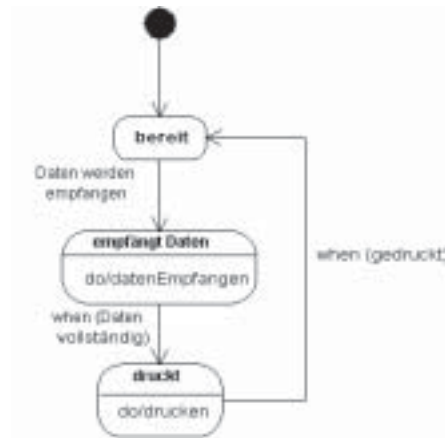


Abbildung 4.75: Wiederholung von Zuständen durch Rücksprünge

In der Abbildung werden die Zustände eines Druckers vereinfacht dargestellt. Zunächst befindet sich der Drucker im Zustand *bereit*. Sobald der Eingang von Daten registriert wird, wechselt das Objekt in den Zustand *empfängt Daten*. In diesem Zustand entfaltet das Gerät die Aktion *datenEmpfangen*. Sind die Daten vollständig übertragen (Bedingung *when (Daten vollständig)*), beginnt das Gerät zu drucken (Aktion *do/drucken*). Nach Beendigung des Druckauftrags (Bedingung *when (gedruckt)*) wechselt das Objekt wieder in seinen ursprünglichen Zustand *bereit*.

Anhand des Zustandsdiagramms lassen sich die Methoden ableiten, die das Objekt *drucker* mindestens besitzen muss.

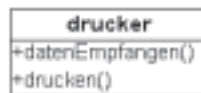


Abbildung 4.76: Objekt *drucker*

Es handelt sich dabei um die Methoden `datenEmpfangen` und `drucken`. Welche zusätzlichen Eigenschaften (Attribute und Operationen) das Objekt `drucker` besitzt, hängt von potenziellen weiteren Verhaltensweisen bzw. der erzeugenden Klasse ab.

Für unser durchgängiges Wecker-Beispiel ergibt sich folgendes Zustandsdiagramm.



Abbildung 4.77: Zustandsdiagramm Wecker

Der erste Zustand des Weckers ist `weckfunktion aus`. In diesem Zustand verharrt der Wecker, bis das Ereignis `Weckfunktion wird eingeschaltet` den Übergang in den Zustand `weckfunktion an` auslöst. Dieses Ereignis repräsentiert den Aufruf der Operation `weckstatusumschalten`.

Im Zustand `weckfunktion an` führt das Objekt die Aktion `weckbereitschaft anzeigen` aus. Der Wecker signalisiert also z.B. durch eine Einblendung im Display, dass die Weckfunktion eingeschaltet wurde.

Ein erneuter Zustandsübergang ist abhängig von der Bedingung `weckzeit = uhrzeit`. Bei Erreichen dieses Zeitpunkts nimmt der Wecker den Zustand `wecken an`, der mit der Aktion `Wecksignal` ver-

bunden ist. Das Signal ertönt so lange, bis jemand die Weckfunktion wieder ausschaltet, wodurch der Wecker erneut seinen ursprünglichen Zustand (`weckfunktion aus`) annimmt.



Die beiden Aktionen in der obigen Abbildung werden in den jeweiligen Zuständen ständig wiederholt – erkennbar an dem Zusatz `do`. Entry- oder Exit-Aktionen würden nur ein einziges Mal, beim Eintritt in einen Zustand bzw. beim Verlassen eines Zustands, ausgeführt.

Aktivitätsdiagramme

Aktivitätsdiagramme stellen eine besondere Form der Zustandsdiagramme dar. Die Zustände sind hier mit einer Verarbeitung verknüpft. In der Regel erfolgen Zustandsübergänge im Unterschied zum Zustandsdiagramm nicht durch Ereignisse, sondern durch die Beendigung eines mit dem Zustand verbundenen Verarbeitungsvorgangs.



Beim Aktivitätsdiagramm sollten weder in einem Zustand noch in einem Zustandsübergang Ereignisse vorkommen.

Aktivitäten und Übergänge

Eine *Aktivität* ist ein Zustand mit einer internen Aktion, dem ein oder mehrere, durch Bedingungen verknüpfte Zustandsübergänge folgen. Einer der Zustandsübergänge wird nach Abschluss der internen Aktion automatisch ausgelöst. Eine Aktivität ist somit ein einzelner Verarbeitungsschritt im Rahmen eines Ablaufs.



Das Auslösen eines Zustandsübergangs wird auch als *Feuern* einer *Transition* bezeichnet.

Die UML bietet damit die Möglichkeit, Aktivitäten sowohl in Zustandsdiagrammen zu verwenden als auch in eigenen Aktivitätsdiagrammen zu konkretisieren.



Auch für die Aktivitäten gilt, dass sie stets ihre Entsprechung in einem statischen Diagramm finden. Sie sind entweder einer Klasse, einer Methode oder einem Anwendungsfall zugeordnet.

Die Notation von Aktivitäten ist leicht zu verwechseln mit den Zuständen, wie die folgende Abbildung zeigt.



Abbildung 4.78: Notation von Aktivitäten und Zuständen im Vergleich

Ein Zustand wird durch ein Rechteck mit abgerundeten Ecken dargestellt. Im Unterschied dazu besteht das Symbol einer Aktivität aus zwei parallelen Geraden (oben und unten), die durch zwei konvexe Linien miteinander verbunden werden. In der Mitte des Symbols steht die Bezeichnung der jeweiligen Aktivität. Diese UML-Konvention findet in der Praxis zahlreiche Abwandlungen. Insbesondere bei handschriftlicher Notation von Zustands- und Aktivitätsdiagrammen sind diese häufig kaum voneinander zu unterscheiden.



Aktivitätsdiagramme dienen vor allem der Beschreibung interner Prozesse. Sie werden verwendet, um den *internen* Kontrollfluss in Prozessen abzubilden. Zustandsdiagramme sind einzusetzen, wenn es besonders auf die Verdeutlichung externer Ereignisse, die ein System beeinflussen, ankommt.

Entscheidungen

Eine Aktivität kann mehrere ausgehende Transitionen besitzen, die an Bedingungen geknüpft sind. Je nachdem welche der Bedingungen nach Beendigung des entsprechenden Verarbeitungsschritts erfüllt ist, wird ein anderer Übergang ausgelöst. In Aktivitätsdiagrammen wird zur Modellierung solcher *Entscheidungen* eine Raute verwendet.



Abbildung 4.79: Entscheidungen in Aktivitätsdiagrammen

Welche Aktivität nach Abschluss von Aktivität1 als Nächste folgt, hängt in dem Beispiel der Abbildung vom Wert x ab. Für $x < 5$ folgt Aktivität2, für $x = 5$ folgt Aktivität3 und falls die Bedingung $x > 5$ erfüllt ist, folgt Aktivität4.

Als Beispiel soll erneut die Überweisung von einem Girokonto dienen (siehe Abbildung 4.80).

Wie die Abbildung zeigt, wird zunächst geprüft, ob das Konto über eine ausreichende Deckung verfügt. Falls nicht, wird die Zahlung abgelehnt, ansonsten erfolgt die Auszahlung des zu überweisenden Betrags.



Abbildung 4.80: Aktivitätsdiagramm Überweisung



Das Beispiel verdeutlicht, dass ein Aktivitätsdiagramm – ebenso wie ein Zustandsdiagramm – in der Regel nur die Aktivitäten im Zusammenhang mit einem einzigen Objekt abbildet. Abbildung 4.80 enthält dementsprechend nur die Aktivitäten des überweisenden Kontos. Zur Betrachtung des zweiten Kontos ist ein eigenes Aktivitätsdiagramm zu erstellen oder ein einziges Aktivitätsdiagramm muss in verschiedene Verantwortlichkeitsbereiche aufgeteilt werden, die wir weiter unten einführen.

Hier werden lediglich die Aktivitäten rund um das bisher als `konto1` bezeichnete Objekt analysiert. Dieses Konto führt bei ausreichender Deckung seine Operation `auszahlen` aus. Die Gutschrift auf dem Gegenkonto (`konto2`) erfolgt über dessen Operation `ein zahlen`.

Splitting und Synchronisation

Auch die Teilung (*Splitting*) und die *Synchronisation* von Aktivitäten ist in der UML vorgesehen. Die entsprechenden Symbole zeigt die folgende Abbildung.

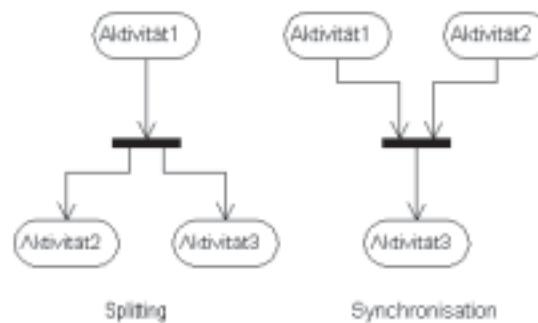


Abbildung 4.81: Splitting und Synchronisation von Aktivitäten

Splitting und Synchronisation werden jeweils durch einen waagrechten schwarzen Balken dargestellt. Beim Splitting führt ein Übergang zu mehreren verschiedenen Aktivitäten, die Synchronisation führt mehrere Aktivitäten zu einer Folgeaktivität zusammen.

Neben Regeln lassen sich mit Aktivitätsdiagrammen ganze Prozesse modellieren. Abbildung 4.82 demonstriert dies an einem Beispiel, das aus der UML-Spezifikation der Object Management Group stammt (vgl. die Hinweise auf Web-Quellen im Anhang).

Abbildung 4.82 zeigt die Bereitung eines Erfrischungsgetränks als Aktivitätsdiagramm. Die erste Aktivität besteht darin, ein Getränk zu finden. In dem Beispiel werden die verfügbaren Getränke auf Kaffee und Mineralwasser beschränkt. Bevorzugtes Getränk sei Kaffee. Wird weder Mineralwasser noch Kaffee gefunden, dann werden die Aktivitäten sofort beendet. Ist nur Mineralwasser zu finden, wird dieses geöffnet und anschließend getrunken (Aktivitäten Mineralwasser öffnen und Trinken).

Für den Fall, dass Kaffee gefunden wird, zeigt das Aktivitätsdiagramm den gesamten Prozess des Kaffeekochens. Dabei sind zunächst drei parallele Teilprozesse zu erledigen: Kaffeepulver in Filter füllen, Wasser in Tank füllen und Tassen aus Schrank nehmen.



Abbildung 4.82: Aktivitätsdiagramm »Erfrischungsgetränk bereiten«

Die ersten beiden Teilprozesse werden wieder zusammengeführt und gehen in die Aktivität Kaffeemaschine einschalten über. Letztere kann allerdings erst begonnen werden, wenn Wasser in Tank füllen und Filter in Kaffeemaschine legen beendet sind.

Nachdem das Wasser komplett durch die Maschine gelaufen ist und die Tassen aus dem Schrank geholt wurden, kann der Kaffee einge-

gossen werden. Als letzte Aktivität folgt schließlich Trinken, womit das Ende des Prozesses erreicht ist.

In dem Beispiel wurden die verschiedenen, bereits eingeführten Elemente von Aktivitätsdiagrammen verwendet. Es lässt zudem erkennen, dass der Abstraktionsgrad für die Darstellung von Prozessen in Aktivitätsdiagrammen weitgehend frei wählbar ist.



Ob ein Prozess bis in das kleinste Detail modelliert und abgebildet wird oder nur ein einziges Aktivitätssymbol für den gesamten (Teil-) Prozess steht, bleibt dem Modellierer überlassen. So können besonders komplizierte oder wichtige Teilprozesse im Einzelnen visualisiert und diskutiert werden. Routineaufgaben und weniger wichtige Aspekte lassen sich auf ein angebrachtes Maß reduzieren oder sogar ganz ausblenden.

Verantwortlichkeitsbereiche

Aktivitätsdiagramme können in *Verantwortlichkeitsbereiche* (so genannte *swimlanes*) aufgeteilt werden. Die einzelnen Verantwortlichkeitsbereiche werden im Diagramm durch senkrechte Linien optisch voneinander getrennt. Jede Aktivität gehört zu genau einem Verantwortlichkeitsbereich, der seinerseits von einem oder mehreren Objekten implementiert werden kann.

Wenn Sie Ihr Aktivitätsdiagramm in Verantwortlichkeitsbereiche aufteilen, können Sie auch die Aktivitäten mehrerer Objekte in einem einzigen Diagramm darstellen.

Abbildung 4.83 demonstriert die Aufteilung eines Aktivitätsdiagramms in Verantwortlichkeitsbereiche am Beispiel der Überweisung.

Wesentliche Aspekte dieser Abbildung sind bereits in dem Aktivitätsdiagramm zu `konto1` enthalten. Neu hinzugekommen ist hier das zweite Konto.

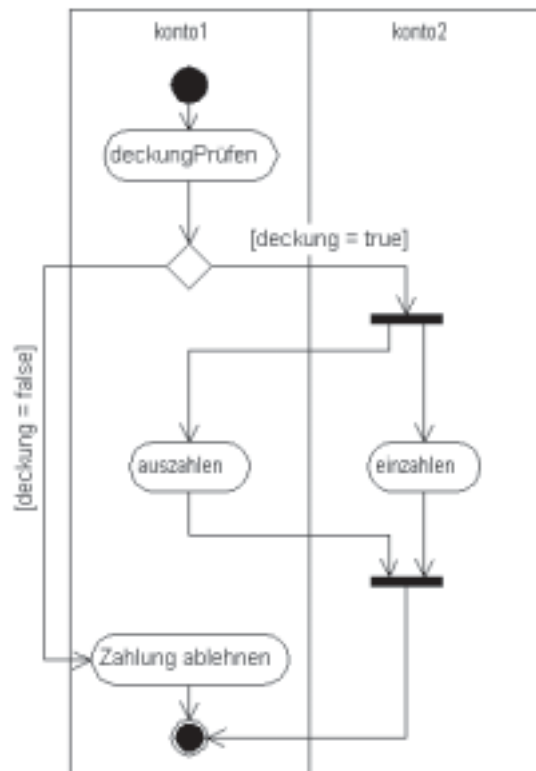


Abbildung 4.83: Verantwortlichkeitsbereiche bei einer Überweisung

Durch diese Erweiterung der Betrachtungsweise auf zwei Konten ist die Einführung von Verantwortlichkeitsbereichen erforderlich. Außerdem benötigen wir nun ein Splitting der Aktivitäten.

Falls die Deckungsprüfung von **konto1** negativ ausfällt, bleibt alles wie gehabt. Die Zahlung wird abgelehnt und die Aktivitäten im Zusammenhang mit der Überweisung sind beendet.

Ist jedoch eine ausreichende Deckung vorhanden, dann wird der Aktivitätsfluss aufgeteilt. Das Objekt *konto1* führt eine Auszahlung und das Objekt *konto2* eine Einzahlung aus (Operationen *auszahlen* und *einzahlen*). Diese beiden Aktivitäten können durchaus parallel ablaufen.

Anschließend wird der Aktivitätsfluss wieder zusammengeführt und der betrachtete Prozess ist zu Ende.



In welcher Reihenfolge Sie die Verantwortungsbereiche in einem Aktivitätsdiagramm anordnen, spielt keine Rolle. Bei der gleichzeitigen Betrachtung mehrerer Objekte können allerdings durch eine geschickte Darstellung die Beziehungen zwischen den einzelnen Bereichen hervorgehoben werden.

Objektzustände in Aktivitätsdiagrammen

Aktivitäten können Objektzustände verändern. Es besteht die Möglichkeit, diese Zustandsänderungen in Aktivitätsdiagramme zu übernehmen und so die Zusammenhänge zu verdeutlichen.

Eine Aktivität, die einen Objektzustand verändert, wird mit einem gestrichelten Pfeil mit diesem verbunden. Das Symbol für den Objektzustand besteht aus einem Rechteck mit dem unterstrichenen Namen des Objekts und darunter der Bezeichnung des Objektzustands in eckigen Klammern.



Abbildung 4.84: Objektzustände in Aktivitätsdiagrammen

Zeigt der Pfeil von der Aktivität auf das Objekt, so bewirkt die Aktivität die dargestellte Änderung des Objektzustands. Zeigt der Pfeil umgekehrt von dem Objektsymbol auf die Aktivität, dann muss der entsprechende Objektzustand erreicht sein, bevor die Aktivität entfaltet werden kann.

Signale

Um zusätzliche Informationen in ein Aktivitätsdiagramm aufzunehmen, können *Signale* als eine Art Kontrollsymbol eingefügt werden. Dabei handelt es sich um Signale, die während eines Prozesses an ein externes Objekt gesendet oder von einem solchen Objekt ausgehend empfangen werden.

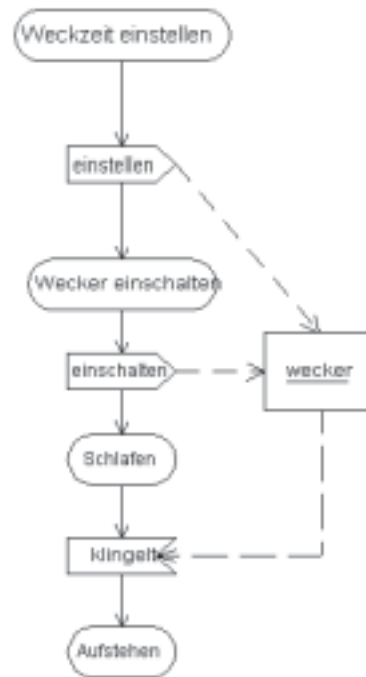
Die folgende Abbildung erläutert die Symbole für eingehende und ausgehende Signale anhand des Wecker-Beispiels. Der hier betrachtete Prozess bildet die Aktivitäten im Rahmen des Schlafens ab (siehe Abbildung 4.85).

Die erste Aktivität, die in dem Realitätsausschnitt berücksichtigt wird, ist das Einstellen der Weckzeit. Diese Aktivität bindet das Objekt *wecker* in den betrachteten Prozess ein. Die Weckzeit wird als Signal an den Wecker gesendet.

Anschließend muss die Weckfunktion noch eingeschaltet werden. Wiederum erhält das Objekt *wecker* ein entsprechendes Signal aus dem Prozess. Daraufhin wird die Aktivität Schlafen ausgelöst. Es folgt das Klingeln des Weckers, das als externes Signal (*klingelt*) vom Wecker in den Prozess eingeht, und schließlich die Aktivität Aufstehen.



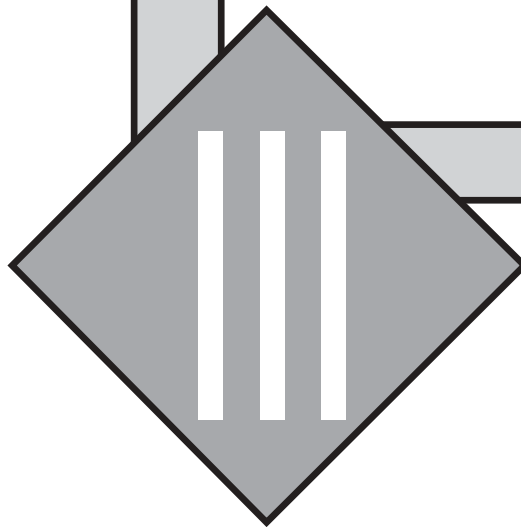
Für das Modell, das in der Abbildung den Prozess des Schlafens repräsentiert, spielt der Wecker nur als externer Signalgeber und -empfänger eine Rolle. Welche Aktivitäten das Objekt in der Zwischenzeit durchführt, ist an dieser Stelle nicht von Bedeutung. Sie können gleichwohl in einem weiteren Aktivitätsdiagramm näher spezifiziert werden.

**Abbildung 4.85:** Eingehende und ausgehende Signale

TEIL

Know-How – UML im Software-Entwicklungsprozess

Ausgehend von dem Fallbeispiel »Mein liebes Geld«, in dem es um die Verwaltung von privatem Vermögen geht, werden konkrete Modellierungstechniken praktisch durchgeführt und insbesondere die Implementierung der erstellten UML-Modelle mit der Programmiersprache Java gezeigt.





KAPITEL

Der Entwicklungsprozess – wie Software entsteht

Dieses Kapitel stellt wesentliche Aspekte eines Software-Entwicklungsprozesses dar und zeigt, welche Beteiligten bei welchen Aktivitäten mit der UML welche Ergebnisse erarbeiten.

5



Der Entwicklungsprozess – wie Software entsteht

Der Prozess der Entwicklung von Software wird traditionell in verschiedene Phasen gegliedert, in denen unterschiedliche Aktivitäten ausgeführt werden. Wie der Prozess der Software-Entwicklung abläuft, hängt entscheidend davon ab, welche grundlegende Sicht Initiatoren und Entwickler einnehmen.

Sichtweisen von Software-Entwicklung

Grundsätzlich lassen sich drei voneinander verschiedene Sichtweisen unterscheiden.

Die Sicht des kreativen Künstlers

Software-Entwicklung besteht aus der Sicht des kreativen Künstlers darin, Ideen völlig frei von äußeren Zwängen umzusetzen. Jede Vorgabe, sei es eine Vorgehensleitlinie, seien es Termine, Meilensteine oder Reviews, lehnt der Programmierer, der sich als Künstler sieht, kategorisch ab. Er unterwirft sich keiner Methodik zur Darstellung (zum Beispiel UML) und zum Vorgehen (beispielsweise RUP oder OEP) und dokumentiert seine Arbeitsergebnisse selten.

Dieser in der Geschichte der Softwaretechnik praktizierte Ansatz ist seit Ende der 1960er Jahre nur noch sehr selten zu finden. Die Einsicht, dass zur Umsetzung hoch komplexer Aufgabenstellungen, die das Arbeiten im Team unter Zeitdruck und ökonomischen Zwängen erfordert, Koordination, Kommunikation und damit auch Vorgaben notwendig sind, ist mittlerweile in der Fachwelt unbestritten.

Die Sicht des verfahrensorientierten Ingenieurs

Ende der 1960er Jahre steckte die Software-Entwickler-Community in einer Krise. Die entwickelte Software war oft mit Fehlern behaftet, entsprach nicht den Vorstellungen des Anwenders und musste mit erheblichem ökonomischen Aufwand nachgebessert werden.

Diese Situation führte die Community zu einer konsequenten Ausrichtung des Software-Entwicklungsprozesses am Beispiel der großindustriellen Fertigung von Konsumgütern. Dort zeigten tüchtige Ingenieure, dass man den Produktionsprozess – zum Beispiel zur Herstellung eines Automobils – nur exakt und formal durchstrukturieren muss, um eine definierte Qualität mit minimalem Aufwand herstellen zu können.

Diese Ausrichtung am industriellen Fertigungsprozess als Leitbild für die Software-Entwicklung brachte auch für die Software-Entwicklung formale Verfahren, die Sie heute in vielen Lehrbüchern der Softwaretechnik finden. Ausgangspunkt waren strukturierte höhere Programmiersprachen und Methoden wie die Strukturierte Analyse (SA) und das Strukturierte Design (SD), lineare Phasenkonzepte und Ähnliches.

Im Ergebnis brachte jedoch auch dieser Ansatz für Aufgaben, zu deren Lösung Anwender mit dem Rechner und Anwender mit dem Computer stark interagieren müssen, häufig nur unzureichende Ergebnisse. Das lag zum einen daran, dass anders als bei einem Konsumgut die Anforderungen für komplexe interaktive Anwendungen, wie beispielsweise ein intelligentes Workflow- und Groupware-System, von vorneherein oft nicht genau festgelegt werden können.

Die Sicht des teamorientierten Designers

Dass Software-Entwicklung mehr ist als das geordnete Zusammenwürfeln von Bits und Bytes, weiß der teamorientierte Designer. Er geht weniger nach einem starren vorgegebenen Phasenschema vor. Er weiß, dass Anforderungen häufig erst im Dialog zwischen Anwendern und Entwicklern Schritt für Schritt im Laufe eines sich selbst steuernden, also evolutionären Prozesses sinnvoll festgelegt

werden können. Für ihn besteht das Ziel seiner Arbeit nicht darin, der Software bestimmte Eigenschaften mit auf den Weg zu geben. Er misst den Erfolg seines Produktes in seiner Einsatzumgebung.

Dazu werden die zu lösenden Aufgaben mit den Menschen und der Technik in Organisationen im Zusammenhang betrachtet. Demnach ist eine Software nur dann erfolgreich, wenn Anwender mit Hilfe der Software ihre Aufgaben zur eigenen Zufriedenheit und im Sinne der Aufgabenstellung ökonomisch effizient erledigen können.

In den letzten Jahren zeigt sich immer mehr, dass gerade für die Entwicklung komplexer, hoch interaktiver Software neue Wege beschritten werden müssen. Der Software-Designer versucht, die individuell unterschiedlichen Sichtweisen der beteiligten Entwickler und Anwender durch die Vorgabe von Leitbildern zu harmonisieren. Dazu schafft er freie und ungebundene Kommunikationsstrukturen, in denen alle Beteiligten in einem gemeinsamen Lehr- und Lernprozess, wie Christiane Floyd ihn bezeichnet, Systemstrukturen konstruieren. Geeignete Konstruktionsansätze wie die UML in Kombination mit einem evolutionären Vorgehen, das keine starren linearen Abläufe vorgibt, unterstützen dann die Umsetzung gemeinsam akzeptierter Systemeigenschaften.

In diesem Ansatz werden selbststeuernde Prozesse initiiert und gestärkt, in der Hoffnung, dass daraus ein System, eingebettet in den sozialen und ökonomischen Kontext, entsteht, das später eher dem entspricht, was der Anwender erwartet.



Betrachten Sie wie der ingenieurmäßige Ansatz die Software-Entwicklung durch die Brille eines Ingenieurs, so steht die exakte Spezifikation von Produkteigenschaften im Vordergrund, die mittels formaler Produktionsverfahren umgesetzt werden soll. Der Designer stellt den Prozess in seiner Entstehung und Entfaltung in den Mittelpunkt, um auf diese Weise ebenfalls ein qualitativ hochwertiges System zu erstellen, das er jedoch immer eingebettet in den Anwendungskontext sieht. Prozess und Produkt sind immer zwei Seiten derselben Medaille.

Welchen Ansatz Sie in einem Software-Entwicklungsprojekt verfolgen, hängt stark von der Problemstellung, aber insbesondere auch von Werten und Überzeugungen der Projektverantwortlichen ab.



Abschließend sei angemerkt, dass wir – die Autoren – die teamorientierte Design-Sicht mit unserer eigenen wissenschaftlichen und praktischen Arbeit fördern und zutiefst davon überzeugt sind, dass dieser Ansatz gerade auch in Zeiten, in denen sich Theorie und Praxis ein wenig auseinander gelebt haben, die Praxis der Software-Entwicklung nachhaltig verbessern kann.

Die Beteiligten – wer entwickelt?

Als Beteiligte eines Systementwicklungsvorhabens bezeichnet man alle Menschen, die mittelbar oder unmittelbar von der Entwicklung und/oder dem Einsatz des Systems betroffen sind. Um der Vielfalt der Einzelpersonen im Entwicklungsprozess Herr zu werden, definiert man einige Kernrollen, die Beteiligte einnehmen. Im Allgemeinen sind dies die Folgenden.

- ✓ Der *Projektinitiator* oder bei Realisierung durch eine Softwarefirma der Auftraggeber, entscheidet über den Start des Projektes bzw. unterzeichnet den Vertrag mit einem Softwarehaus.
- ✓ Ein *Projektmanager* steuert und koordiniert alle Projektaktivitäten und ist verantwortlich für Termine, Meilensteine, Reviews.
- ✓ Der *Anwender* kooperiert mit einer Software in einem Arbeitssystem. Dazu muss er nicht zwingend mit der Software interagieren. Es reicht aus, wenn er Arbeitsergebnisse nutzt.
- ✓ *Benutzer* interagieren direkt mit einer Software. Er ist direkt mit der Benutzungsschnittstelle des Systems in Kontakt.
- ✓ *Fachexperten* kennen die fachlichen Inhalte, die in ein Software-system umgesetzt werden. Sie müssen nicht direkt als Anwender oder Benutzer auftreten.

- ✓ *Analytiker* legen ihren Arbeitsschwerpunkt auf die Modellierung und Darstellung fachlicher Inhalte.
- ✓ *Designer* modellieren auf der Grundlage des fachlichen Analysemodells einen detaillierten Entwurf, der technische Elemente wie eine Benutzungsoberfläche, Datenbankschnittstelle, Formalisierung und Verfeinerung des Analysemodells unter konstruktiven Gesichtspunkten der Implementierung hervorbringt.
- ✓ *Architekten* sehen ihren Schwerpunkt in der Konstruktion von Anwendungssystemen. Konstruktion wird dabei schwerpunktmäßig als das Erstellen, Anpassen und Zusammenfügen von Komponenten verstanden. Arbeitsschwerpunkt ist die Entwurfsphase. Analyse wird zur Vorbereitung betrieben und Implementierung zu Testzwecken.
- ✓ *Entwickler* definieren ihre Tätigkeit von der Programmierung her. Sie sehen Analyse und Entwurf als Mittel zur Vorbereitung ihres Kerngeschäfts, das in der Programmierung liegt.
- ✓ *Programmierer* sind spezialisiert auf die Implementierung vorgegebener Entwürfe. Sie übernehmen die Entwürfe und sind selten an anderen Aktivitäten des Entwicklungsprozesses beteiligt.

Die hier aufgeführten Rollennamen werden häufig gebraucht, wenn Beteiligte ihre Funktion definieren sollen. Deutlich werden sollte durch die Auflistung, dass hinter jeder Bezeichnung unterschiedliche Grundüberzeugungen und Werte stehen, die zu einer bestimmten Perspektive führen. Da die Rollen Perspektiven enthalten, die sich kreuzen, und folglich nicht überschneidungsfrei sind, betrachten wir im Folgenden exemplarisch die Rollen des *Anwenders* stellvertretend für fachlich orientierte Beteiligte und des *Entwicklers* als Repräsentanten der Modellierungs- und Implementierungsseite.

Die Aktivitäten – wie geht man vor?

Im professionellen Umfeld entwickeln Teams von Experten Software nach modernen Methoden des Projektmanagement und der Softwaretechnik. Dabei unterstützen Software-Werkzeuge den gesamten Zyklus der Entwicklung von der Idee bis hin zur Realisierung und darüber hinaus auch während des Einsatzes zur Wartung und Pflege.

Ist die Entscheidung zur Entwicklung eines Softwareprodukts gefallen, so erstellen die Systemanalytiker zunächst einen fachlichen Entwurf, der noch keine softwarebezogenen Details enthält. Im nächsten Schritt verfeinern Softwaredesigner diesen Entwurf und beschreiben ihn in einer formalen Sprache, damit die Programmierer, die diesen Entwurf anschließend mit Hilfe einer Programmiersprache umsetzen, möglichst wenig Aufwand treiben müssen. In jeder Phase des Zyklus arbeiten Entwickler mit Anwendern zusammen.

Auch hier lassen sich grundlegende Sichtweisen unterscheiden, die wir nach zwei Gesichtspunkten darstellen. Zunächst betrachten wir das Verhältnis von Plan und Ausführung.

- ✓ Das *starre Vorgehen* richtet die Aktivitäten an einem Vorgehensmuster aus. Ein Vorgehensmuster legt die logische und zeitliche Abfolge des Entwicklungsprozesses in einem Plan fest, an dem sich der Projektverlauf verbindlich orientiert. Das Phasenmodell determiniert den Prozessverlauf
- ✓ Beim *evolutionären Vorgehen*, also einem aus sich selbst heraus entwickelnden Entwicklungsprozesses, bestimmen die Erfordernisse, die sich aus einer Aktivität ergeben, die Folgeaktivitäten. Hier stellt sich das Verhältnis von tatsächlichem Ablauf und zugrunde liegendem Phasenmodell genau anders herum dar. Der tatsächliche Ablauf determiniert das Phasenmodell.



Beim starren Vorgehen bestimmt das Phasenmodell den tatsächlichen Ablauf und besitzt somit Steuerungscharakter für den Entwicklungsprozess. Beim evolutionären Vorgehen spielt das

Phasenkonzept eine untergeordnete Rolle, da es erst ex post als Ergebnis vorliegt und damit keinen direkten gestaltenden Einfluss besitzt.

Im Ablauf können für beide Grundperspektiven im Ergebnis zwei idealtypische Projektverläufe resultieren.

- ✓ Der *lineare Ablauf* gibt es eine Abfolge von der Projektplanung über die Analyse, den Entwurf bis hin zur Realisierung vor, die mit der Einführung endet und nachfolgend von Pflege- und Wartungsaktivitäten ergänzt wird. Entscheidend ist, dass jede Phase im Grundsatz nur einmal durchlaufen wird und nach Abschluss die nächste Phase initiiert. Rücksprünge sind immer Sonderfälle, die bei mangelnden Ergebnissen der vorhergehenden Phasen möglich sind, jedoch nur einen im Kern unerwünschten Schritt darstellen.
- ✓ Beim *zyklischen Ablauf* ist ein wiederholter Durchlauf von Planung, Analyse, Entwurf und Implementierung ausdrücklich erwünscht. Man geht im Prinzip davon aus, dass Erkenntnisgewinne, die in weiteren Konstruktionszyklen umgesetzt werden, die Systemqualität entscheidend verbessern können.



Beim linearen Ablauf geht man davon aus, dass die Anforderungen an ein System von vornherein bekannt sind und im Projektverlauf konstant bleiben. Sind sie einmal fixiert, gelten sie als konstant und jedes weitere Zwischenergebnis in Analyse, Entwurf und Implementierung wird daran gemessen. Abweichungen von den Vorgaben können dann als unerwünschte aber notwendige Nachbesserungsschritte eingeleitet werden. Beim zyklischen Vorgehen wird der Wissenszuwachs, der sich bei allen Beteiligten im Projektverlauf ergibt, dadurch systematisch genutzt, dass planmäßig mehrere Zyklen von der Planung über die Analyse und den Entwurf bis hin zur Realisierung durchlaufen werden.

Insbesondere durch die Zusammenarbeit von Anwendern und Entwicklern findet ein solcher Wissenszuwachs statt.

In der folgenden Abbildung sind Fachgebietsexperten (häufig Anwender) hell und EDV-Experten (Entwickler) dunkel dargestellt. Die Anzahl der Personen einer Gruppe symbolisiert dabei das Gewicht in der einzelnen Phase.

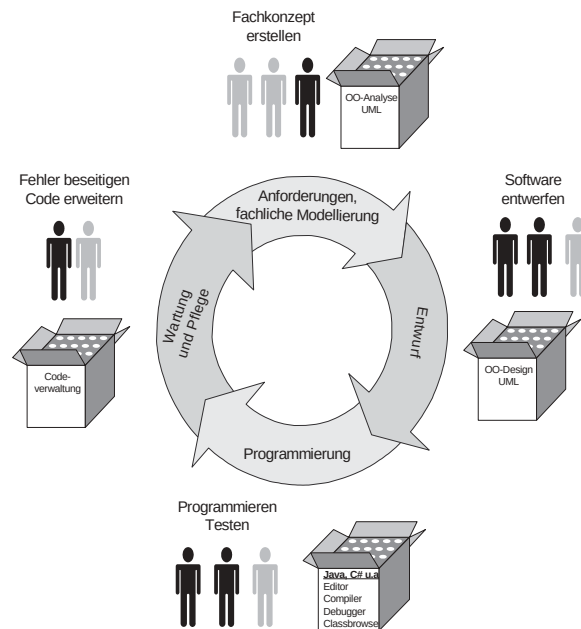


Abbildung 5.1: Aktivitäten und Werkzeuge des Software-Entwicklungszyklus

Neben den bereits erwähnten Phasen sind die eingesetzten Werkzeuge und Methoden dargestellt, die typischerweise in objektorientierten Projekten eingesetzt werden.

Dabei kommt der UML als Modellierungssprache eine große Bedeutung insbesondere in den Phasen Analyse und Entwurf zu. Während die UML bei der Analyse als Strukturierungshilfe für kognitive Prozesse und als Kommunikationsmittel zwischen den Beteiligten eingesetzt wird, dient sie beim Entwurf als technische Konstruktionssprache.

Dabei ist der Einsatz eines Modellierungswerkzeuges, wenn Sie die UML einsetzen, absolut zu empfehlen. Im nächsten Teil stellen wir Ihnen bekannte und aus unserer Sicht zu empfehlende Werkzeuge unter dem Aspekt der UML-Modellierung vor, so dass Sie in die Lage versetzt werden, nach eigenen Kriterien das zu Ihren Anforderungen passende Werkzeug zu finden.



Setzen Sie zur Modellierung immer ein UML-Werkzeug ein, da die UML so komplex ist, dass Sie ohne Werkzeug die Vorteile, die Ihnen die UML bietet, nicht ausschöpfen können. Außerdem sind Ihre Modelle kaum erweiter- und änderbar. Darüber hinaus bieten die meisten UML-Tools Konsistenzchecks Ihrer Modelle und eine automatische Code-Generierung (UML-Modelle in Java, C++ oder C#-Code) sowie Reverse-Engineering (Programm-Code in ein UML-Modell) an.

Damit ist der Übergang vom objektorientierten Entwurf in eine objektorientierte Programmiersprache sehr erleichtert. Mehr als ein grobes Codegerüst dürfen Sie aber nicht erwarten.



Alle UML-Tools, die sich auf der CD zum Buch befinden, unterstützen die Code-Generierung und z.T. auch das Reverse-Engineering.

Der Entwurf oder das automatisch generierte Codegerüst wird dann in der Programmierung von *menschlichen* Programmierern mit den Standardwerkzeugen Editor (Erstellen des Quellcodes), Compiler (Übersetzen des Quellcodes), Debugger (Testen des Programms)

und für objektorientierte Sprachen dem Klassenbrowser (*Classbrowser*, Navigieren in vorgegebenen oder selbst erstellten Klassenbibliotheken) in eine lauffähige Anwendung übersetzt.

Die Ergebnisse – was wird entwickelt?

Betrachten wir noch die Ergebnisse, die auf der Basis einer bestimmten Vorgehensweise zum Ende von Aktivitäten als Dokumente oder Programme entstehen.

Die Abfolge der Aktivitäten Analyse, Entwurf und Programmierung, die logisch aufeinander aufbauen, kann zeitlich auch parallel vorangetrieben werden. So kann es sein, dass als erste Aktion im Projekt ein Prototyp der Benutzungsoberfläche für den Kernbereich entsteht, damit Anwender und Entwickler sich über die Inhalte besser unterhalten können.

Parallel dazu testet ein Programmierer die Performance einer Datenbankkopplung. Wichtig ist, dass Aktivitäten zwar eine Affinität zu einer bestimmten Phase haben, aber eben nicht ausschließlich dort ausgeführt werden. So gibt es Prototypen (Programmierung) in allen Phasen. Auch der Check von Entwurfsmustern (Entwurf) kann in der Analysephase sinnvoll sein (siehe Abbildung 5.2).

Wie gezeigt, ist eine starre Zuordnung von Aktivitäten zu den Phasen ebenso wenig möglich wie die feste und exklusive Zuordnung von Beteiligten zu den Phasen. Ja nicht einmal der zeitliche Ablauf muss der in den Abbildungen gezeigten logischen Abfolge entsprechen.

Die Kunst, ein Projekt erfolgreich durchzuführen, liegt darin, eine für jedes Projekt individuelle Kombination von Beteiligten, Aktivitäten und Ergebnisdokumenten zu finden, die auf der Basis der jeweiligen Grundperspektive beruht.

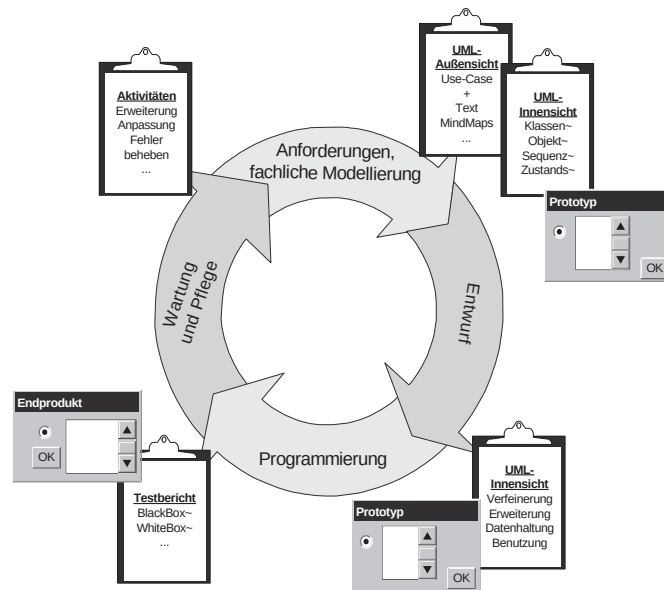


Abbildung 5.2: Ergebnisdokumente im Entwicklungsprozess

Stellt man die Menschen, die an einem Projekt beteiligt sind, in den Mittelpunkt seiner Betrachtung, so treten damit automatisch soziale und kognitive Aspekte als zentrale Erfolgskriterien an die Seite der bekannten ökonomischen und technischen Kriterien. Zu Beginn haben wir die drei K der Systementwicklung *Kognition*, *Kommunikation* und *Konstruktion* als die zentralen Elemente identifiziert.

Kognition, also das menschliche Denken, die Abstimmung der Denkprozesse durch Kommunikation und die ökonomische technische Umsetzung in Konstruktionsschritten spiegeln Ergebnisse dieser Grundhaltung wider.



KAPITEL

Fallbeispiel: »Mein liebes Geld«

In diesem Kapitel werden die Erkenntnisse des fünften Kapitels genutzt, um anhand des Fallbeispiels einer privaten Vermögensverwaltung konkrete Modellierungstechniken praktisch anzuwenden.

6



Fallbeispiel: »Mein liebes Geld«

Stellen Sie sich vor, Sie sind für die Finanzen und das Vermögen eines mittelständischen Unternehmens zuständig. Um effizient handeln zu können, möchten Sie das Vermögen und die Finanztransaktionen der Unternehmung mit dem PC verwalten, um schnell auf Aktienkurse oder Wertpapierangebote reagieren zu können.

Ihnen schwebt vor, die wichtigsten Anlageformen mit den zentralen Daten im Computer zu verwalten, um flexibel und schnell reagieren zu können. Dazu gehören neben Auswertungen und Übersichten der Vermögensanlagen genauso die Ermittlung der zur Verfügung stehenden finanziellen Mittel. Sie als Finanzchef haben zwei Mitarbeiter, die sich um Finanztransaktionen und die Vermögensanlage im operativen Geschäft kümmern.

Ausgangssituation

Bisher haben Sie Ihre Transaktionen über die Bank abgewickelt und telefonisch oder vor Ort bei Ihrem Bankberater Auskunft über Ihre finanziellen Mittel und sonstigen Vermögenswerte eingeholt, die Sie in unterschiedlichen Depots halten. Da die Planung dadurch unnötig verzögert wird, soll ein System erstellt werden, das Ihnen als Anwender einen Überblick über Finanzmittel und Vermögen bietet.

Nachdem Sie geprüft haben, ob eine fertige Standardlösung am Markt an Ihre individuellen Bedürfnisse angepasst werden kann, entscheiden Sie sich für eine individuell auf Sie zugeschnittene Lösung. Sie beauftragen ein Softwarehaus mit der Umsetzung des Projektes.

Analyse – Anforderungen

Anwender einigen sich mit dem Entwickler auf die folgenden groben Anforderungen, die zwingend erfüllt sein müssen:

- ✓ Verwaltung von Aktien, Wertpapieren und Fonds in Depots
- ✓ Verwaltung von Giro- und Sparkonten
- ✓ Jederzeitige Ermittlungsmöglichkeit der Liquidität
- ✓ Auswertungen über die Zinsen aller Anlagen der Unternehmung

Darüber hinaus ist es wünschenswert, Aktienkäufe und Finanztransaktionen aus dem System heraus online durchzuführen:

- ✓ Unterstützung von Online-Banking für alle Girokonten der Unternehmung
- ✓ Unterstützung von Online-Brokerage aus dem System heraus

Das System soll nicht beinhalten:

- ✓ Buchführung
- ✓ Kennzahlenberechnung (cash flow o.Ä.)

Analyse – fachliche Beschreibung

Die Modellierung der fachlichen Inhalte wird mit der UML dargestellt. Hilfreiche Zusatztechniken wie das Erstellen von Mind-Maps zur Visualisierung von Ideen und ein Fachlexikon zur Herausbildung einer gemeinsamen Sprache vernachlässigen wir an dieser Stelle.

Anwendungsfälle

Um einen ersten Überblick über die Aktivitäten und Zuständigkeiten des Anwendungsbereiches zu erhalten, modelliert ein Entwick-

ler zusammen mit dem Finanzchef und einem seiner Mitarbeiter ein Anwendungsfalldiagramm.

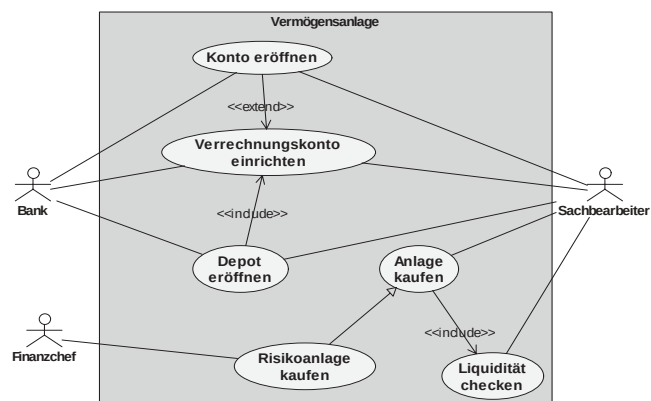


Abbildung 6.1: Anwendungsfall: Vermögensverwaltung

Die Abbildung zeigt einen Ausschnitt, der die Aktivitäten der Einrichtung und des Kaufs von Anlagen zeigt. Um den betrachteten Ausschnitt übersichtlich zu halten, sehen wir von der Darstellung weiterer Aktivitäten wie Verkauf und Strukturüberwachung der Anlagen ab.

- ✓ Einer der Akteure des Anwendungssystems ist die Hausbank (Bank), bei der die betrachtete Unternehmung ihre Konten und Depots einrichtet. Der Finanzchef und die Sachbearbeiter tragen die Verantwortung für die Aktivitäten aus der Sicht der Unternehmung.
- ✓ Die Bank und der Sachbearbeiter der Unternehmung sind als Akteure verantwortlich für die Eröffnung eines Kontos (Konto eröffnen).
- ✓ Bank und Sachbearbeiter sind ebenfalls beteiligt an der Eröffnung eines Depots (Depot eröffnen).

- ✓ Da jedes Depot mit einem Liquiditätskonto zur Verrechnung ausgestattet wird (Verrechnungskonto einrichten), sind auch dafür Sachbearbeiter und Bank gemeinsam zuständig.
- ✓ Der Sachbearbeiter erhält allein die Verantwortung für den Kauf von Anlagen, Aktien und Wertpapieren.
- ✓ Der Finanzchef der Unternehmung kümmert sich ausschließlich um den Kauf von Anlagen, die ein hohes Risiko besitzen (Risikoanlage kaufen).

Die Anwendungsfälle weisen untereinander Verbindungen auf, die im Folgenden kurz erläutert werden:

- ✓ Eröffnet der Sachbearbeiter in Kooperation mit der Bank ein Depot (Depot eröffnen), so können Anlagen in Form von Aktien und Wertpapieren erst dann gekauft werden, wenn ein Konto zur Abbuchung des Kaufpreises und zur Verrechnung der Zinsen eingerichtet ist. Folglich verwendet der Anwendungsfall Depot eröffnen den Anwendungsfall Verrechnungskonto einrichten (<<uses>> - Beziehung).
- ✓ Falls noch kein Konto besteht, kann optional der Anwendungsfall Konto eröffnen den Anwendungsfall Verrechnungskonto einrichten erweitern (<<extends>> - Beziehung). Im Unterschied zur <<uses>> - Beziehung stellt extends eine optionale, nicht obligatorische Erweiterung an einem definierten Erweiterungspunkt (extension point) dar.
- ✓ Kauft der Sachbearbeiter eine Anlage (Anlage kaufen), so überprüft er in diesem Zusammenhang immer die Liquidität (Liquidität prüfen). Da die Liquiditätsprüfung auch in anderen Kontexten eine Rolle spielt, gilt sie als eigenständiger Anwendungsfall, den der Anwendungsfall Anlage kaufen über die uses-Beziehung einbindet.
- ✓ Der Kauf einer Anlage, die mit Risiko behaftet ist (Risikoanlage kaufen), bedarf einer gesonderten Prüfung, die sehr viel strenger und genauer als beim Kauf einer gewöhnlichen Anlage ist. Diese Prüfung initiiert immer der Finanzchef selbst. Der

Anwendungsfall Risikoanlage kaufen ist somit ein Spezialfall des Anwendungsfalles Anlage kaufen. Diese Spezialisierung stellt die Vererbungsbeziehung (Pfeil mit durchgezogener Linie und dreieckiger Pfeilspitze) dar.

Klassenstruktur

Der Sachbearbeiter erarbeitet zusammen mit dem Finanzchef und dem Software-Entwickler diese Grobstruktur der Anwendung, die zunächst nur zentrale Klassen, deren Attribute und Operationen sowie deren Beziehungen untereinander enthält.

Das Ziel dieser fachlichen Darstellung ist dabei, eine möglichst leicht erfassbare Gesamtstruktur darzustellen, die nur für das Verständnis notwendige Details enthält. Darüber hinaus sollte das Modell dem kognitiven Verständnis der Beteiligten entsprechen, die sich in Kommunikationsprozessen auf eine Struktur einigen.

Entscheidend für diese Phase ist, dass technische Aspekte weder direkt noch indirekt in die Betrachtung mit einbezogen werden sollen. Um das zu gewährleisten, ist es sinnvoll, auch eine personelle Trennung bei den Entwicklern vorzunehmen. Das heißt, dass es sinnvoll sein kann, zur Ermittlung des fachlichen Modells in der Analysephase von Seiten des Software-Hauses andere Mitarbeiter einzusetzen als in der Entwurfs- und Realisierungsphase.

Sollte eine Trennung aus personellen Gründen nicht möglich oder aus anderen Gründen unerwünscht sein, besteht die Gefahr, dass der Entwickler die technische Machbarkeit im Hinterkopf hat und damit unbewusst in der Analyse die zum Teil im Konflikt stehenden Ziele von Analyse und Entwurf miteinander vermischt.



Verhindern Sie die zu starke Berücksichtigung von Entwurfszielen in der Analysephase durch eine personelle Trennung von Analytikern und Entwicklern, die das fachliche Modell in der Entwurfsphase übernehmen und weiterentwickeln. Sollte eine

personelle Trennung nicht möglich sein, muss der Entwickler in der Lage sein, zwischen Analyse- und Entwurfszielen sauber zu trennen.

Die folgende Abbildung zeigt die wichtigsten fachlichen Zusammenhänge, ohne Details und technische Informationen zu enthalten.

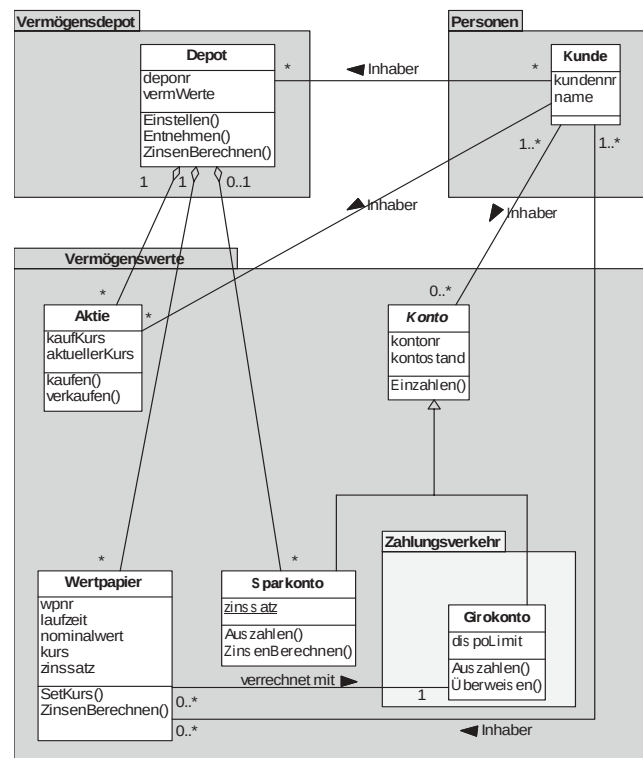


Abbildung 6.2: Analyse: Klassenstruktur

Die gesamte Anwendung umfasst drei Organisationsbereiche, die jedoch miteinander in Beziehung stehen.

- ✓ Das Paket Vermögensdepot enthält Depots, in denen alle Vermögenswerte, die die betrachtete Unternehmung besitzt, verwaltet werden.
- ✓ Das Paket Personen enthält Informationen über Mitarbeiter, die berechtigt sind, Transaktionen auf den jeweiligen Konten vorzunehmen. Sie werden der Einfachheit halber als Inhaber geführt, obwohl rein rechtlich die Unternehmung immer Inhaber der Anlagen ist.
- ✓ Das Paket Vermögenswerte definiert alle Anlageformen, die für unsere Unternehmung in Frage kommen.
- ✓ Innerhalb des Paketes Vermögenswerte ist ein Unterpaket Zahlungsverkehr vereinbart, das Informationen über die Abwicklung von Transaktionen enthält.

Das Paket Vermögensdepot enthält eine Klasse Depot, die als Kartei für alle Anlagen aufgefasst werden kann.

- ✓ Jedes Depot erhält eine eindeutige Nummer (deponr).
- ✓ Wesentliche Aufgaben des Depots bestehen in der Aufnahme neuer Vermögenswerte (Einstellen()), dem Verkauf oder dem Depotwechsel von Vermögenswerten (Entnehmen()). Um den Gesamtertrag eines Depots zu bestimmen, summiert es die Zinsen bzw. Erträge über alle Vermögenswerte (ZinsenBerechnen()).

Jedes Depot definiert verfassungsberechtigte Mitarbeiter als Inhaber, die für ein bestimmtes Depot zuständig sind. Das Paket Personen vereinbart aus diesem Grund die Klasse Kunde.

- ✓ Ein Kunde ist aus der Sicht der Vermögensverwaltung nur mit den Attributen kundennr und name belegt, die ihn zu diesem Zeitpunkt ausreichend charakterisieren. Hier ist die Terminologie aus dem Bankbereich übernommen, die von Kunden und Inhabern sprechen und nicht von Mitarbeitern, die vertretungsberechtigt sind.

- ✓ Der Kunde tritt als Inhaber eines Depots auf. Damit hat er die Möglichkeit, Vermögenswerte einzustellen, zu entnehmen und einen Überblick über die Erträge zu ermitteln.
- ✓ Jedes Depot kann mehrere Kunden als Inhaber besitzen und jeder Kunde mehrere Depots verwalten.

Ein Kunde besitzt eine weitere Funktion in Bezug auf die Vermögenswerte, die in ein Depot aufgenommen werden können. Für jeden Vermögenswert gibt es Inhaber, die unabhängig von der Depotzugehörigkeit festgelegt werden.

- ✓ So besitzen Aktie, Wertpapier und die abstrakte Klasse Konto, als Stellvertreter für Sparkonto und Girokonto, eine Beziehung zur Klasse Kunde, die die Inhaberschaft ausdrückt.
- ✓ Für jeden Vermögenswert muss mindestens ein Inhaber festgelegt werden. Es kann jedoch auch mehr als ein Inhaber bestimmt werden.
- ✓ Ein Wertpapier ist durch `wpnr`, `laufzeit`, `nominalwert`, `kurs` und `zinssatz` beschrieben. `SetKurs()` dient dazu, aktuelle Kursänderungen zu verbuchen und `ZinsenBerechnen()` ermittelt die Erträge. Bei der Implementierung gehen wir detaillierter auf die Attribute und Operationen ein.
- ✓ Jedes Konto, egal ob Sparkonto oder Girokonto, kann durch seine `kontonr` und den `kontostand` näher beschrieben werden. Das `Einzahlen()` ist ebenfalls für alle Konten identisch.
- ✓ Ein Sparkonto, das als spezialisierte Variante über eine Vererbungsbeziehung mit der generalisierten Klasse Konto verbunden ist, definiert zusätzlich den `zinssatz` als Klassenattribut sowie die Operationen `Auszahlen()` und `ZinsenBerechnen()`, um den Ertrag zu bestimmen.
- ✓ Das Girokonto, das ebenfalls von der Klasse Konto erbt, vereinbart zusätzlich das Attribut `dispoLimit` sowie die Operationen `Auszahlen()` und `Überweisen()`.

- ✓ Die Aktie wird mit den Attributen `kaufKurs` und dem aktuellen `Kurs` beschrieben und kann gekauft (`kaufen()`) und verkauft (`verkaufen()`) werden. Da die Projektbeteiligten nur wenig Kenntnisse über Aktien besitzen wird, die weitere Modellierung zurückgestellt und soll später, nachdem die Kernfunktionalität für Wertpapiere und Sparkonten umgesetzt ist, in das System integriert werden.
- ✓ Aktien und Wertpapiere müssen als Teile genau einem Depot (Aggregat) zugeordnet werden, während jedes Depot beliebig viele Aktien und Wertpapiere enthalten kann.
- ✓ Sparkonten können einem Depot zugeordnet werden, können aber auch unabhängig von einem Depot verwaltet werden, je nachdem, ob die Erträge mit erfasst werden sollen oder nicht.
- ✓ Girokonten können keinem Depot zugeordnet werden, da sie ausschließlich finanzielle Transaktionen wie beispielsweise zum Kauf und Verkauf von Aktien und Wertpapieren realisieren.
- ✓ Zusätzlich dient das Girokonto dazu, die Zinsen von Wertpapieren aufzunehmen. Das besagt die Assoziation verrechnet mit, in der festgelegt ist, dass jedes Wertpapier mit einem festgelegten Girokonto seine Erträge verrechnet und jedes Girokonto als Gegenpart beliebig vieler Wertpapiere fungieren kann.

Typische Sequenz der Depotverwaltung

Den typischen Ablauf der Depotverwaltung stellt das folgende Sequenzdiagramm dar. Dabei sind drei verschiedene Szenarien hier in einer Abbildung zusammengefasst.

Die Einrichtung eines Depots, die Zinsermittlung aller Werte eines Depots und die Entnahme eines Wertpapiers aus einem Depot zwecks Verkauf zeigen die grundlegende Zusammenarbeit der Objekte, ausgehend vom Sachbearbeiter.

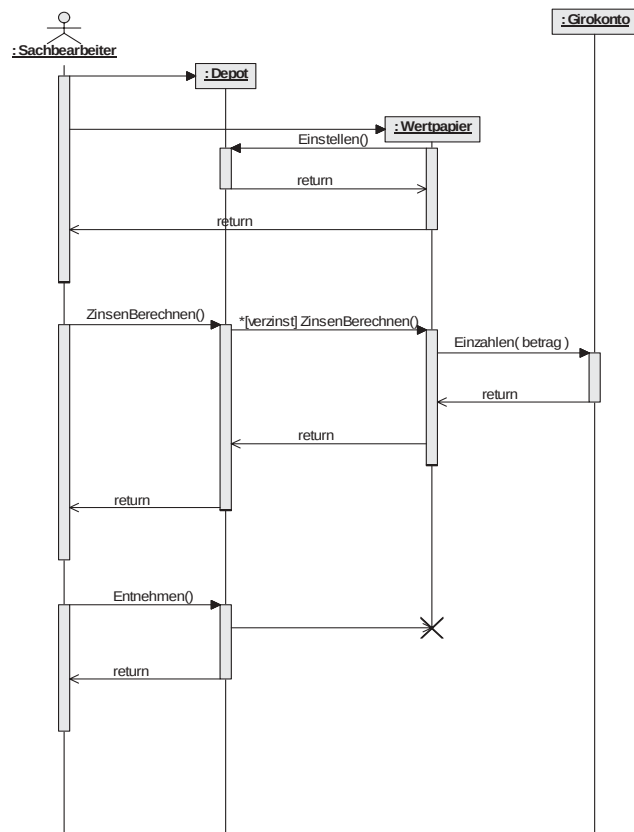


Abbildung 6.3: Sequenz: Wertpapier einstellen, verzinsen, verkaufen

Die erste Sequenz zeigt das Zusammenspiel von Sachbearbeiter, Depot und Wertpapier bei der Eröffnung eines Depots und dem Zukauf eines Wertpapiers.

- ✓ Der Sachbearbeiter eröffnet zunächst ein :Depot, um anschließend ein :Wertpapier in das System aufzunehmen. Da ein Wertpapier immer einem Depot angehören muss, wird es über die Operation Einstellen() in das Depot aufgenommen.
- ✓ Die zweite Sequenz zeigt, wie ein Sachbearbeiter die Zinsberechnung für ein Depot anstößt (ZinsenBerechnen()) und diese Operation wiederum die Zinsberechnung aller verzinsten Wertpapiere durchführt, um die Ergebnisse aufzusummieren und an den Sachbearbeiter zurückzugeben. Die Operation ZinsenBerechnen() eines jeden Wertpapiers sorgt dafür, dass der Ertrag dem als Verrechnungskonto vereinbarten :Girokonto durch Aufruf der Operation Einzahlen() gutgeschrieben wird.
- ✓ In der dritten Sequenz entnimmt der :Sachbearbeiter ein :Wertpapier dem :Depot. Die Operation Entnehmen() sorgt gleichsam dafür, dass das entnommene :Wertpapier aus dem System entfernt wird.

Auf die Modellierung weiterer Diagramme verzichten wir, da sowohl die Statische Perspektive mit dem Klassendiagramm als auch die Dynamische Perspektive ausreichend detailliert und verständlich erscheinen.

Entwurf – technische Konstruktion

Im Entwurf geht es um die technischen Aspekte der Konstruktion des Softwaresystems.

Verfeinerung und Erweiterung

Aus der Sicht des Entwurfs wird die Struktur erweitert und verfeinert, zwecks Vorbereitung der Umsetzung mit einer objektorientierten Programmiersprache.

Für unser Modell legen wir zusätzliche Schnittstellen fest, um verschiedene Vermögenswerte in einem Depot zusammenfassen und verzinsbare Vermögensgegenstände gruppieren zu können.

Inhaltlich ergibt sich eine Änderung, indem auch Girokonten Teile eines Depots werden können, damit eine Gesamtübersicht auch die finanziellen Mittel darstellen kann.

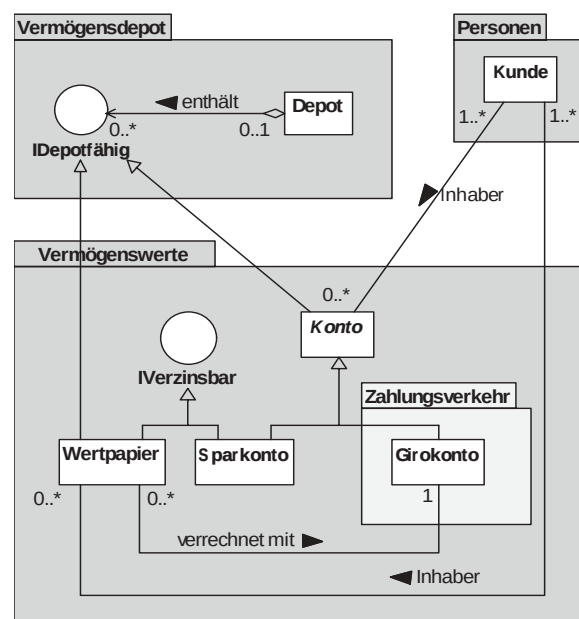


Abbildung 6.4: Entwurf: Strukturelle Veränderung

- ✓ Die Schnittstelle `IDepotfähig` vereinbart Operationen, die notwendig sind, damit ein Vermögenswert in einem Depot verwaltet werden kann.
- ✓ Die Schnittstelle `IVerzinsbar` definiert Aufrufkonventionen von Operationen, die für alle verzinsbaren Vermögenswerte garantiert anbieten müssen.

- ✓ Da jedes Konto die Schnittstelle `IDepotfähig` implementiert und alle Objekte, die diese Schnittstelle implementieren, in ein Depot eingestellt werden können, kann ein Depot auch ein Girokonto als Unterklasse der Klasse `Konto` aufnehmen.
- ✓ Auch müssen nicht mehr alle depotfähigen Vermögensgegenstände in einem Depot gehalten werden. Damit ist die Anwendung flexibler als in der Analyse vorgesehen.

Des Weiteren werden technisch erforderliche zusätzliche Klassen definiert, und den bestehenden Klassen Attribute und Operationen hinzugefügt, die für die internen Abläufe notwendig sind.

Operationen erhalten Parameter und Rückgabewerte, deren Typ ebenfalls vereinbart werden muss.

Die gerade genannten zusätzlichen Vereinbarungen stellen wir im folgenden Kapitel, das sich mit der Realisierung dieses Modells mit der Sprache C# beschäftigt, vollständig dar, da viele Aspekte erst mit der Umsetzung verständlich werden.

Datenhaltung

Viele Anwendungen müssen Daten, die über eine Sitzung hinaus im System erhalten bleiben sollen, dauerhaft speichern. Oft werden Anwendungssysteme mit Datenbanken gekoppelt. Um eine Unabhängigkeit von einem Datenbanksystem zu erreichen, ist es sinnvoll, die Datenhaltungskomponente systematisch vom erstellten fachlichen Modell zu entkoppeln. Vielleicht planen Sie ja, in regelmäßigen Abständen auf neuere Versionen Ihres Datenbanksystems umzusteigen oder gar das Datenbanksystem zu wechseln. Bleibt das fachliche Modell dagegen über mehrere Wechsel des Datenbanksystems hinaus konstant, verringert sich der Aufwand bei einer weitgehenden Entkopplung erheblich.

Benutzungsoberfläche

Es hat sich als Vorteil herausgestellt, das fachliche Modell, das Gegenstand der Analyse war, zu trennen von Datenhaltung und Be-

nutzungskomponente. Die Erfinder der Programmiersprache Smalltalk in den 1970er Jahren haben das so genannte Model-View-Controller-Paradigma geprägt, um das fachliche Modell (Model) von seiner visuellen Darstellung (View) und der Interaktion (Controller) mit dem Benutzer zu entkoppeln.

Häufig ist es so, dass verschiedene Benutzergruppen ein Programm nutzen werden. Da hat es Sinn, dasselbe fachliche Modell für unterschiedliche Benutzungstypen auch unterschiedlich zu präsentieren.

- ✓ Fachkundige Benutzer beherrschen die Terminologie des Anwendungsbereichs und kennen elementare Verfahren zur Lösung von Standardaufgaben. Der Computer kann ihnen Routinenarbeiten auf Anforderung abnehmen.
- ✓ Fachunkundige Benutzer müssen ohne Fachtermini in der Lösung einer Aufgabe mit Fachwissen und Lösungsverfahren unterstützt werden.

In Bezug auf die Erfahrungheit der Benutzer mit EDV-Systemen unterscheiden wir grob die folgenden beiden Benutzertypen:

- ✓ Computer-erfahrene Benutzer beherrschen die Bedienung moderner grafischer Benutzungsoberflächen und sind mit Standard-Dialogen zum Öffnen und Speichern von Arbeitsergebnissen, sowie der grundlegenden Steuerung eines Rechners über die Tastatur und die Maus geübt.
- ✓ Computer-unerfahrene Benutzer wissen wenig über das Handling eines Computers und sollten intuitiv an die Benutzung eines Anwendungsprogramms herangeführt werden.

Egal ob ein Benutzer Erfahrung im Umgang mit Computern besitzt und unabhängig von seinen einschlägigen fachlichen Kenntnissen und Fertigkeiten lassen sie sich in verschiedene Typen unterteilen.

- ✓ Der visuelle Typ findet in grafischen Benutzungsoberflächen mit Piktogrammen und Maussteuerung, die auch im Smalltalk-Projekt in den 1970er Jahren erforscht und entwickelt wurden, eine ihm angemessene Interaktionsumgebung. Sie wurde nach

der Schreibtisch-Metapher entworfen und bietet dem Benutzer ein bekanntes Szenario, den Schreibtisch mit Dokumenten (Fenstern), Karteikästen (Dateiordner), Papierkorb (Löschroutine) und anderen Werkzeugen (Programme). Die Umgebung ist intuitiv nutzbar, ähnlich einem klassischen Schreibtisch und ohne viel Vorerfahrung.

- ✓ Der textliche Typ liebt eher spartanisch anmutende, aus der DOS-Zeit bekannte zeichenorientierte Benutzungsoberflächen. Er zieht Tastaturkürzel der Maussteuerung vor und mag keine poppigen bunten Benutzungsoberflächen. Analytische, formal denkende Menschen ziehen häufig textliche Interaktionsformen vor.

Benutzungsstrategien legen fest, in welchem Verhältnis Benutzer und System stehen.

- ✓ Der Benutzer steuert die Dialogfolge möglichst frei, wenn er qualifiziert ist, um Arbeitsinhalte und Arbeitsabläufe kompetent zu steuern. Der Computer ist sein Assistent, der per Einzelauftrag durch den Benutzer einzelne Arbeitsschritte übernimmt. Der Computer dient als Werkzeug des Menschen.
- ✓ Der systemgeführte Dialog, den Sie von Assistenten oder Wizards kennen, beherrscht eine Methode zur Lösung von konkreten Aufgaben. Der Mensch gilt als Bediener des Computers.

Im Ergebnis müssen oft verschiedene Benutzungsstrategien für unterschiedliche Benutzertypen in ein System integriert werden. Folglich sollte das fachliche Modell mit möglichst wenig Aufwand mit mehreren Benutzungsschnittstellen ausgestattet werden können.

Ein anderer eher technischer Aspekt ist die Notwendigkeit, eine Anwendung an neue grafische Oberflächen oder andere Betriebssysteme anzupassen. Auch da hilft eine klare Trennung zwischen fachlichem Modell und der Benutzungsschnittstelle.

Die präsentierten Elemente können in GUI-Klassen unabhängig vom fachlichen Modell im UML-Klassendiagramm abgebildet wer-

den. Dialogabläufe dagegen lassen sich übersichtlich mit Zustandsdiagrammen darstellen.

KAPITEL

Implementierung von UML-Modellen

Dieses Kapitel zeigt, wie Klassen, Schnittstellen und deren definierte Beziehungen mit der Programmiersprache C# umgesetzt werden können. Wir greifen den Entwurf der Fallstudie »Mein liebes Geld« auf und setzen diesen Schritt für Schritt um.

7



Implementierung von UML-Modellen

Im Gegensatz zum Vorgehen eines erfahrenen Programmierers entwickeln wir das Modell des Fallbeispiels »Mein liebes Geld« in diesem Kapitel von der Pike auf, als müssten wir viele Erkenntnisse noch erarbeiten. Das hat den Vorteil, dass viele Konzepte und insbesondere auch eine Begründung ihrer Auswahl viel deutlicher wird.

In den vorangegangenen Kapiteln haben Sie erfahren, wie Sie Klassen und Objekte mit der UML in Klassendiagrammen definieren, wie Sie Aktivitäten von Operationen in Sequenzdiagrammen, Zustandsdiagrammen und Aktivitätsdiagrammen festlegen und wie Sie beschreibende Eigenschaften einem Objekt zuordnen.

In den folgenden Abschnitten erarbeiten wir zunächst systematisch den Aufbau, die Nutzung und das Beenden von Objektbeziehungen aus der Perspektive des Programmierers. Derartige Beziehungen sind aus der Sicht eines objektorientierten Programms als Attributwerte in Attributen verankert und können im Lauf der Existenz eines Objekts durch eine einfache Wertzuweisung verändert oder beendet werden. Daher bezeichnen wir sie auch als *dynamische Beziehungstypen*.

Anschließend betrachten wir Beziehungen, die in der Klasse fest verankert sind und sich somit während der Existenz eines Objekts nicht ändern können. Diese elementaren *statischen Beziehungstypen* beziehen sich auf den Aufbau von Klassen und beinhalten die Übernahme von fremden Klassendefinitionen und Schnittstellen in Form der *Vererbung*.

Damit Sie in Systemen, die viele Klassen und Schnittstellen beinhalten, den Überblick behalten, können Sie ausgewählte Klassen und Schnittstellen zu den bereits zu Beginn kurz erläuterten Paketen oder, wie sie in C# genannt werden, *Namensräumen* (Namespaces)

zusammenfassen. Diese Organisationseinheiten können auch zu übergeordneten Namensräumen zusammengefasst werden, so dass eine hierarchische Struktur von Namensräumen aufgebaut wird.

Namensräume sind Organisationseinheiten, die ähnlich aufgebaut sind wie die Organisation von Unternehmungen. Wie Sie es aus Organigrammen kennen, die Mitarbeiter Abteilungen zuordnen und Abteilungen zu Oberabteilungen zusammenfassen, lassen sich Klassen und Interfaces in Namensräumen zusammenfassen, die ihrerseits wieder eine hierarchische Struktur besitzen können. Stellen Sie sich Namensräume wie ein Organigramm für Klassen und Interfaces vor, das diese Elemente bestimmten Namensräumen zuordnet, die ihrerseits zu übergeordneten Namensräumen zusammengefasst werden können.

»Mein liebes Geld« – Sparkonto in C#

Kern unserer Überlegungen für die Geldanlage ist ein Sparkonto, das wir bei einer Bank eröffnen. Uns interessiert zunächst nur die Kontonummer, der Name des Kontoinhabers sowie der aktuelle Kontostand. Darüber hinaus sollen Einzahlungen und Auszahlungen beliebiger Beträge vorgenommen werden können.

Sparkonto
-kontonr -inhaber -kontostand
+Einzahlen(betrag) +Auszahlen(betrag) +GetKontostand()

Abbildung 7.1: Klasse Sparkonto – Grundform

Zusätzlich zu den hier abgebildeten Strukturinformationen enthält die Klasse Sparkonto in C# einen Individual-Konstruktor (Zeilen

9-12), der den Namen des Inhabers (`inhaber`) und die Kontonummer (`kontonr`), die als Parameter übergeben werden, den entsprechenden Attributen zuweist.

```
1: // Listing 7.1
2: using System;
3: public class Sparkonto {
4:     string inhaber;
5:     int kontonr;
6:     double kontostand;
```

In der Klasse `Sparkonto` vereinbaren Sie den Kontoinhaber `inhaber` als Zeichenkette, die Kontonummer `kontonr` als Ganzzahl und den Kontostand `kontostand` als Fließkommazahl.

Der Individual-Konstruktor, den Sie zum Einrichten eines Objekts aufrufen, erhält als Übergabewert den Namen des Inhabers sowie die Kontonummer.

```
7:     public Sparkonto(string inhaber, int kontonr){
8:         this.inhaber = inhaber;
9:         this.kontonr = kontonr;
10:    }
```

Die übergebenen Werte legt der Konstruktor in den Zeilen 9-10 als Attributwerte fest. Damit sind die Angaben dauerhaft, über die Laufzeit des Konstruktors hinaus, im `Sparkonto`-Objekt verfügbar.

Um eine Einzahlung vorzunehmen, definieren Sie die Operation `Einzahlen(...)`, die bei ihrem Aufruf immer die Angabe eines Betrags benötigt, der dem `Sparkonto`-Objekt gutgeschrieben werden soll.

```
11:     public void Einzahlen(double betrag){
12:         kontostand += betrag;
13:     }
```

Jede Einzahlung erhöht den Wert des Attributs `kontostand` um den angegebenen Betrag.

Den entgegengesetzten Weg beschreitet die Operation `Auszahlen(...)`. Sie benötigt ebenfalls die Angabe eines Betrags, der jedoch in diesem Fall ausgezahlt werden soll.

```
14: public bool Auszahlen(double betrag){
15:     bool gedeckt;
16:     if (gedeckt = kontostand >= betrag)
17:         kontostand -= betrag;
18:     return gedeckt;
19: }
```

Im Unterschied zu einer Einzahlung darf aber nicht jeder beliebige Betrag ausgezahlt werden. Nur wenn der Kontostand höher als der Zahlungsbetrag ist, also eine Auszahlung nicht zu einem negativen Kontostand führen würde, darf der Betrag vom Konto abgeboben werden. Die Operation `Auszahlen(...)` zahlt den geforderten Betrag nur dann aus, wenn dadurch der Kontostand nicht negativ wird – daher die bedingte Anweisung in Zeile 16.

Die Operation `GetKontoStand()` ermöglicht die Abfrage des aktuellen Kontostands.

```
20: public double GetKontoStand(){
21:     return kontostand;
22: }
23: }
```

Die folgende Startklasse erzeugt ein Sparkonto, führt einige Kontobewegungen durch und gibt anschließend den Kontostand aus.

```
1: // Listing 7.2
2: using System;
3: class Start {
4:     public static void Main(string[] args) {
5:         Sparkonto meinKonto =
6:             new Sparkonto("Micha", 4711);
7:         meinKonto.Einzahlen(100);
8:         meinKonto.Einzahlen(300);
9:         meinKonto.Auszahlen(1150);
```

```
10:         meinKonto.Auszahlen(150);
11:         Console.WriteLine("Kontostand: {0} EURO",
12:                             meinKonto.GetKontostand());
13:     }
14: }
```

Führen Sie die Startklasse aus, so erhalten Sie die folgende Bildschirmausgabe, die nach allen Kontobewegungen einen Kontostand von 250 Euro anzeigt.

Kontostand: 250 EURO

Nachdem das Sparkonto-Objekt `meinKonto` erzeugt wurde (Zeilen 6-7), zahlen Sie die Beträge 100 € und 300 € zunächst ein (Zeilen 8-9), bevor die erste Auszahlung von 1150 € (Zeile 10) abgelehnt wird, da aus der Auszahlung ein negativer Kontostand resultieren würde. Die letzte Auszahlung von 150 € wird hingegen vorgenommen (Zeile 11) und der resultierende Kontostand von 250 € ausgegeben (Zeilen 12-13).

Ausgehend von diesem Sparkonto-Objekt demonstrieren die nächsten Abschnitte die Handhabung von Objektbeziehungen in C#.

Assoziation – »LAG« von Objekten

Nachdem Sie Beziehungen zwischen Objekten bereits zwischen Karaffen und Regalen sowie zwischen Sparschwein-Objekten und den in ihnen enthaltenen Münzfächern aufgebaut haben, werden Sie in diesem Abschnitt am Beispiel des Sparkontos lernen, wie Objekte Beziehungen aufbauen, nutzen und wieder beenden können.

Phasen einer Objektbeziehung

Wenn wir Objekte in einer Analogie mit Menschen gleichsetzen, indem wir ihnen ein Gedächtnis (Zustand bzw. Gesamtheit der Attribute) und eine Reaktionsmöglichkeit auf Umweltereignisse (Ver-

halten bzw. Gesamtheit der Operationen) zuschreiben, erscheint es nur konsequent, wenn wir die Beziehungen von Objekten ebenfalls mit menschlichen Beziehungen vergleichen.

Betrachten wir den Lebenszyklus eines Objekts von seiner Erzeugung bis zu seiner Vernichtung, so sind die Beziehungen, die es im Laufe seiner Existenz eingeht, mit LAGs (Lebensabschnittsgemeinschaft) von Menschen vergleichbar.

Bevor wir von einer Beziehung profitieren können, muss diese zunächst aufgebaut werden. Das bedeutet, dass die Beteiligten sich als Beziehungspartner zunächst anerkennen und die Möglichkeiten und Fähigkeiten des anderen verstehen müssen, um Aufgaben möglichst effizient gemeinsam bewältigen zu können. Bietet eine Beziehung den Beteiligten keine Vorteile mehr, so kann die Bindung wieder gelöst werden.

Beziehung aufbauen

Der erste Schritt einer Beziehung ist immer die bewusste Anerkennung eines Beziehungspartners. Genau wie »im richtigen Leben« gibt es auch zwischen Objekten einseitige und zweiseitige Beziehungen oder, wenn mehr als zwei Objekte beteiligt sind, auch mehrseitige Beziehungen.

Eine derartige einseitige Beziehung geht im folgenden Beispiel auch ein Sparkonto-Objekt mit einem Kunden-Objekt ein.

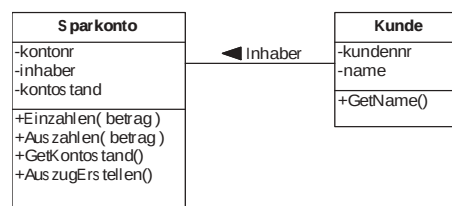


Abbildung 7.2: Sparkonto – Beziehung aufbauen

Um den Aufbau einer Beziehung zu demonstrieren, erstellen wir eine neue Klasse `Kunde`. `Konto`-Objekte können dann Beziehungen zu `Kunden`-Objekten aufbauen, die den oder die Inhaber eines Kontos festlegen.

Vereinbaren Sie zunächst die Klasse `Kunde` mit den Attributen `kundennr` und `name`.

```
1: // Listing 7.3
2: public class Kunde {
3:     int kundennr;
4:     string name;
```

Dem folgenden Individual-Konstruktor übergeben Sie zur Erzeugung eines `Kunden`-Objekts den Namen und die Kundennummer. Der Konstruktor weist die übergebenen Werte den entsprechenden Attributen zu.

```
5:     public Kunde(string name, int kundennr) {
6:         this.kundennr = kundennr;
7:         this.name = name;
8:     }
```

Zu Dokumentationszwecken stellt jedes `Kunden`-Objekt auf Anfrage eine Kennung, die sich aus dem Kundennamen und der Kundennummer zusammensetzt, zur Verfügung.

```
9:     public string GetKennung(){
10:         return name + " (" + kundennr + ")";
11:     }
12: }
```

Damit das `Sparkonto`-Objekt eine Beziehung zu einem `Kunden`-Objekt aufbauen kann, müssen wir in der Klasse `Sparkonto` das bisher als Zeichenkette behandelte Attribut `inhaber` nun als Verweis auf ein eigenständiges `Kunden`-Objekt definieren. Das Attribut `Inhaber` ist somit in der Lage, eine Beziehung zu einem `Kunden`-Objekt herzustellen.


```
1: // Listing 7.4
2: using System;
3: public class Sparkonto {
4:     Kunde inhaber;
5:     /*...*/
```

Da jedes Sparkonto zwingend bereits zu Beginn seiner Existenz einen Inhaber besitzen muss, ist der Konstruktor die geeignete Operation, um eine solche Beziehung aufzubauen. Dazu wird der Übergabeparameter `inhaber` mit einem Verweis auf den Kontoinhaber gefüttert.

```
6:     public Sparkonto(Kunde inhaber, int kontonr){
7:         this.inhaber = inhaber;
8:         this.kontonr = kontonr;
9:     }
10:    /*...*/
```

Der Konstruktor trägt dann diesen Verweis im gleichnamigen Attribut des Objekts ein (Zeile 7). Damit ist die Beziehung vom Sparkonto zu seinem Inhaber über die Zeit, in der der Konstruktor aktiv ist, hinaus im Konto-Objekt verfügbar.

Die Operation `AuszugErstellen()` zeigt, wie das Sparkonto-Objekt seine Beziehung zum Kunden-Objekt nutzt, um die Kennung eines Kunden in einem Kontoauszug auszugeben.

```
11: public void AuszugErstellen(){
12:     Console.WriteLine("Konto: {0}, Inhaber: "+
13:         "{1}, Kontostand: {2} EURO",
14:         kontonr,
15:         inhaber.GetKennung(),
16:         kontostand);
17: }
18: }
```

Dazu ruft die Operation in der Zeile 15 über das Beziehungsattribut `inhaber` die Operation `GetKennung()` auf, die dann als Argument der Ausgabeanweisung zusammen mit der Kontonummer und dem Kontostand auf dem Bildschirm ausgegeben wird.

Mit der alleinigen Verankerung der Beziehung im Attribut `inhaber` der Klasse `Sparkonto` liegt eine einseitige oder einseitig navigierbare Beziehung vor. Ein `Sparkonto`-Objekt kennt das als `Inhaber` zugeordnete `Kunden`-Objekt und kann seine Operation `GetKennung()` (Zeile 15) aufrufen. Das `Kunden`-Objekt hat dagegen keinen Zugriff auf die ihm zugeordneten `Sparkonto`-Objekte.

Die folgende Startklasse erzeugt ein `Sparkonto`-Objekt, führt einige Ein- und Auszahlungen aus und gibt anschließend den Kontoauszug auf dem Bildschirm aus.

```
1: // Listing 7.5
2: using System;
3: class Start {
4:     static void Main(string[] args) {
5:         Kunde einKunde = new Kunde("Tom", 102);
6:
7:         Sparkonto einKonto =
8:             new Sparkonto(einKunde, 4711);
9:         einKonto.Einzahlen(100);
10:        einKonto.Einzahlen(300);
11:        einKonto.Auszahlen(150);
12:
13:        einKonto.AuszugErstellen();
14:    }
15: }
```

Das in Zeile 5 erzeugte `Kunden`-Objekt, das Sie über die Variable `einKunde` ansprechen können, soll `Kontoinhaber` des `Sparkonto`-Objekts `einKonto` sein. Durch die Übergabe des Verweises `einKunde` bei der Erzeugung des `Konten`-Objekts an dessen Konstruktor ist `einKunde` als `Kontoinhaber` festgelegt.

Da der Konstruktor den Verweis auf das `Kunden`objekt dauerhaft im Attribut `inhaber` des `Sparkonto`-Objekts verankert, ist diese Beziehung über die Laufzeit des Konstruktors hinaus verfügbar. Die Beziehung ist erst dann beendet, wenn entweder das `Sparkonto`-Objekt aufgelöst wird oder eine andere Operation explizit den Verweis entfernt.

Beziehung leben

Nachdem das Sparkonto eingerichtet und damit auch die Beziehung zu seinem Inhaber festgelegt ist, nutzt das Sparkonto-Objekt nach der Durchführung einiger Kontobewegungen (Zeilen 9-11) in der Operation `AuszugErstellen()` diese Beziehung, um den Namen des Kontoinhabers und seine Kundennummer auf dem Kontoauszug anzugeben.

Die Anweisung in Zeile 13 ruft dazu lediglich die Operation auf. Der Zugriff auf den Inhaber ist, wie Sie im vorhergehenden Listing bereits gesehen haben, intern im Körper der Operation geregelt. Der Aufruf der Operation gibt den folgenden einfachen Kontoauszug, der bisher lediglich das Konto, den Kontoinhaber und den Kontostand enthält, auf dem Bildschirm aus.

Konto: 4711, Inhaber: Tom (102), Kontostand: 250 EURO

Um die speziellen Fähigkeiten des Kunden-Objekts zu nutzen, muss das Sparkonto-Objekt neben der Zugriffsmöglichkeit, die über das Attribut `inhaber` festgelegt ist, auch deren Operationen kennen.

Während Menschen in der Lage sind zu lernen, welche Fähigkeiten andere Menschen besitzen und bereit sind, zur Verfügung zu stellen, ist in einem Software-Objekt der Programmierer dafür verantwortlich, diese Kenntnisse, die nur er besitzt, umzusetzen. So inspiziert der Programmierer während der Erstellung der Klasse `Sparkonto` die Klasse `Kunde` und implementiert dann den Aufruf von `GetKennung()` des Kunden-Objekts innerhalb der Operation `AuszugErstellen()` des Sparkonto-Objekts. Der Programmierer nutzt also sein Wissen über den Aufbau und die Verwendungsmöglichkeit des Kunden-Objekts, um diese Kenntnisse als interne Operationsaufrufe in Operationen eines Sparkonto-Objekts zu verwenden.



Während Menschen auch Beziehungen zu fremden, bisher unbekannten Menschen aufbauen können, ist das für Software-Objekte nur sehr begrenzt möglich. Zwar existieren im .NET-

Framework Ansätze zur Ermittlung von Informationen über unbekannte fremde Objekte (`Namespace System.Reflection`), sie vermitteln jedoch ausschließlich die formalen Aspekte einer Kopplung wie die Aufrufkonvention von Operationen, nicht dagegen ihre inhaltliche Bedeutung.

Beziehung beenden

Im Laufe seines Daseins kann es vorkommen, dass der Kontoinhaber wechselt. So ist z.B. denkbar, dass der Firmeninhaber seinem Sohn, der als Assistenten in der Firma arbeitet, zunächst auf seinen eigenen Namen ein neues Sparkonto eröffnet hat, das er ihm nach einer Probezeit zur alleinigen Verantwortung überlässt.

Für ein Konto ist es entscheidend, dass immer ein Kontoinhaber existiert. Somit ist die Beendigung einer Beziehung zwischen einem Konto und seinem Inhaber nur möglich, wenn im gleichen Atemzug ein neuer Inhaber festgelegt wird.

Diesen Wechsel wollen wir nun durch die Ergänzung der Operation `InhaberWechseln()` in der Klasse `Sparkonto` vornehmen.

```
1: // Listing 7.6
2: using System;
3: public class Sparkonto {
4:     /*...*/
5:     public void InhaberWechseln(Kunde neuInhaber){
6:         this.inhaber = neuInhaber;
7:     }
8: }
```

Das folgende Programm führt einige Transaktionen durch und wechselt den Inhaber.

```
1: // Listing 7.7
2: using System;
3: class Start {
4:     static void Main(string[] args) {
```

```
5:      Kunde einKunde = new Kunde("Micha", 101);
6:
7:      Sparkonto einKonto = new
8:          Sparkonto(einKunde, 4711);
9:
10:     einKonto.Einzahlen(100);
11:     einKonto.Einzahlen(300);
12:     einKonto.Auszahlen(150);
13:     einKonto.AuszugErstellen();
14:
15:     Kunde einSohn = new Kunde ("Chiron", 103);
16:     einKonto.InhaberWechseln(einSohn);
17:     einKonto.AuszugErstellen();
18: }
19: }
```

Der Aufruf der Operation `InhaberWechseln(...)` beendet in Zeile 16 die Beziehung zu dem alten Kontoinhaber mit dem Namen Micha und stellt gleichzeitig eine Beziehung zum neuen Inhaber Chiron her. Chiron ist nun Inhaber desselben Kontos mit der Kontonummer 4711, wie die folgenden Bildschirmausgaben der Startklasse zeigen.

Konto: 4711, Inhaber: Micha (101), Kontostand: 250 EURO

Konto: 4711, Inhaber: Chiron (103), Kontostand: 250 EURO

Beziehungsmengen – Kardinalität

Bisher erlaubt die Klasse `Sparkonto` ausschließlich die Zuordnung genau eines Kontoinhabers. Denkbar ist aber auch, dass ein Vater für sich und seinen Sohn ein Gemeinschaftskonto einrichtet, damit er auch nach dem zehnten Geburtstag seines Sohnes Zugriff auf das Konto besitzt.

Jetzt ist es also denkbar, dass ein neuer Kontoinhaber hinzugefügt wird, ohne dass der oder die bisherigen Kontoinhaber ausscheiden.

Darüber hinaus kann ein Inhaber ausscheiden, ohne dass ein neuer Inhaber an seine Stelle rücken muss, wenn nach dem Ausscheiden noch mindestens ein Inhaber verbleibt.

Die hier im Beispiel beschriebenen zulässigen Zuordnungsmengen einer Objekt-Beziehung, die so genannten *Kardinalitäten*, können auch im Klassendiagramm dargestellt werden. Die Bereichsangabe *1..** (vgl. die folgende Abbildung) am Kunden-Ende der Objekt-Beziehung besagt, dass ein Sparkonto mindestens einen Kontoinhaber besitzen muss. Das ***-Symbol weist darauf hin, dass keine Begrenzung für die Anzahl der Kontoinhaber besteht. Der *** auf der Seite des Sparkontos bedeutet, dass ein Kunde nicht Inhaber eines Sparkontos sein muss, aber Inhaber eines Sparkontos oder mehrerer Sparkontos sein kann.

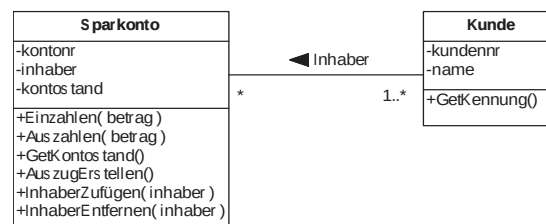


Abbildung 7.3: Sparkonto-Klasse – Beziehung mit Kardinalitäten

Die Operation `InhaberWechseln(...)` wird ab sofort durch die beiden Operationen `InhaberZufügen(...)` und `InhaberEntfernen(...)` abgelöst.

Weitere Anpassungen müssen Sie am Attribut `Inhaber` und an all den Operationen vornehmen, die auf `Inhaber` zugreifen.

```

1: // Listing 7.8
2: using System;
3: using System.Collections;
4: public class Sparkonto {
  
```

```
5:   ArrayList inhaber;  
6:   int kontonr;  
7:   double kontostand;
```

Das Attribut `Inhaber` vereinbaren Sie vom Typ `ArrayList` (Zeile 5). Diese Klasse bietet ähnlich wie der Typ `Array` die Möglichkeit, mehrere Objekte aufzunehmen. Im Gegensatz zu einem `Array` ist die Anzahl der Elemente nicht beschränkt und das Einfügen und Entfernen von Elementen wird durch geeignete Operationen vereinfacht. Da die Klasse `ArrayList` dem Namensraum `System.Collections` zugeordnet ist, müssen Sie diesen in Zeile 3 zunächst einbinden.

```
8:   public Sparkonto(Kunde inhaber, int kontonr){  
9:       this.inhaber = new ArrayList();  
10:      this.inhaber.Add(inhaber);  
11:      this.kontonr = kontonr;  
12:   }  
13:   /*...*/
```

Da der `Inhaber` jetzt in einem eigenen `Collection`-Objekt hinterlegt wird, muss auch dieses vor seiner ersten Verwendung mit dem `new`-Operator (Zeile 9) erzeugt werden. Die Operation `Add(...)` fügt jetzt dem als Attribut vereinbarten `Collection`-Objekt den Verweis auf das `Inhaber`-Objekt `inhaber` zu.

Um dem neuen Attribut `inhaber` weitere `Inhaber` zuzuordnen, vereinbaren Sie die Operation `InhaberZufügen(...)`. Ebenso wie beim Konstruktor übergeben Sie einen Verweis auf den Kunden, der Kontoinhaber werden soll.

```
14:  public void InhaberZufügen(Kunde inhaber){  
15:      this.inhaber.Add(inhaber);  
16:  }
```

Da das `Collection`-Objekt `inhaber` bereits durch den Aufruf des `Individual`-Konstruktors erzeugt ist, weist die Operation an dieser Stelle den neuen `Inhaber` lediglich zu.

Die Operation `InhaberEntfernen(...)` erhält als Übergabewert einen Verweis auf das Inhaber-Objekt, das als Inhaber ausscheidet.

```
17:  public void InhaberEntfernen(Kunde inhaber){
18:      if (this.inhaber.Count > 1) {
19:          this.inhaber.Remove(inhaber);
20:      }
21:  }
```

Das Attribut `Count` des `ArrayList`-Objekts `inhaber` enthält die Anzahl der dort verzeichneten Inhaber-Objekte. Nur wenn mindestens zwei Inhaber dort verzeichnet sind, darf der Verweis auf den gewünschten Inhaber entfernt werden (Aufruf der Operation `Remove`).

Die Operation `AuszugErstellen()` berücksichtigt in ihrer neuen Version alle Inhaber.

```
22:  public void AuszugErstellen(){
23:      string kennungen = null;
24:      foreach(Kunde kunde in inhaber) {
25:          kennungen += kunde.GetKennung() + " ";
26:      }
27:      Console.WriteLine("Konto      : {0}" +
28:                        "\nInhaber   : {1}" +
29:                        "\nKontostand: {2} EURO\n",
30:                        kontonr,
31:                        kennungen,
32:                        kontostand);
33:  }
34: }
```

Zunächst wird in dem obigen Listing die Variable `kennungen` vereinbart (Zeile 23), die von der `foreach`-Anweisung (Zeilen 24-26) dazu verwendet wird, der Reihe nach die Kennungen aller Inhaber zu einer Zeichenfolge zusammenzufügen.

Diese Zeichenfolge wird anschließend in der Ausgabeanweisung (Zeilen 27-33) eingesetzt.

Erstellen Sie auch hier wieder eine Startklasse, um die neue Funktionalität unseres Kontos zu überprüfen.

```

1: // Listing 7.9
2: using System;
3: class Start {
4:     static void Main(string[] args) {
5:         Kunde einKunde = new Kunde("Micha", 101);
6:
7:         Sparkonto einKonto = new
8:             Sparkonto(einKunde, 4711);
9:         einKonto.Einzahlen(345);
10:        einKonto.AuszugErstellen();

```

Nachdem ein Kunden-Objekt erzeugt wurde (Zeile 5), dient es bei der Einrichtung des Sparkontos (Zeilen 7-8) zunächst als alleiniger Kontoinhaber.



Abbildung 7.4: Konto-Objekt – Einfach-Beziehung

Somit besitzt das Sparkonto-Objekt einKonto nach seiner Einrichtung eine Beziehung zu genau einem Inhaber einKunde.

Die folgenden Anweisungen nutzen jetzt die hinzugewonnenen Möglichkeiten. Der Sohn Chiron wird jetzt Mitinhaber des Sparkontos.

```

11: Kunde einSohn = new Kunde ("Chiron", 103);
12: einKonto.InhaberZufügen(einSohn);
13: einKonto.AuszugErstellen();

```

Das neu erstellte Kunden-Objekt einSohn wird zusätzlich zu dem bisherigen Inhaber durch Aufruf der Operation InhaberZufügen(einSohn) im Attribut inhaber des Sparkonto-Objekts festgelegt.

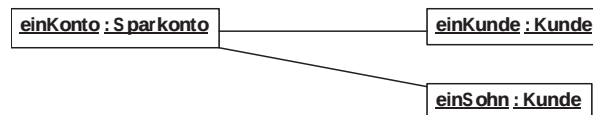


Abbildung 7.5: Sparkonto-Objekt – Mehrfach-Beziehung

Das Sparkonto-Objekt wird nun von den beiden Inhabern einKunde und einSohn gemeinschaftlich geführt.

Nachdem der Vater sich überzeugt hat, dass sein Sohn das Sparkonto in seinem Sinne führt, scheidet er als Kontoinhaber aus und überlässt seinem Sohn die alleinige Verantwortung.

```
14:      einKonto.InhaberEntfernen(einKunde);  
15:      einKonto.AuszugErstellen();
```

Um das Objekt einKunde als Kontoinhaber zu entfernen, rufen Sie die Operation `InhaberEntfernen(einKunde)` auf.



Abbildung 7.6: Sparkonto-Objekt – Ersetzen eines Inhabers I

Damit ist der Sohn einSohn alleiniger Kontoinhaber.

Der Sohn entschließt sich, seine Schwester ebenfalls an dem Sparkonto zu beteiligen.

```
16:      Kunde  
20: }
```

Nachdem das zugehörige Objekt `eineTochter` angelegt ist und anschließend dem Konto hinzugefügt wurde, existieren wieder zwei Kontoinhaber.

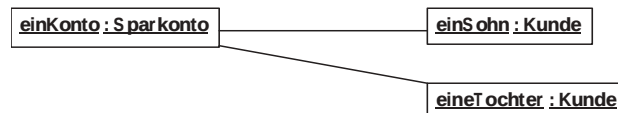


Abbildung 7.7: Sparkonto-Objekt – Ersetzen eines Inhabers II

Wenn Sie die Startklasse ausführen, sehen Sie in den vier Kontoauszügen die unterschiedlichen Inhaberkonstellationen.

```
Konto      : 4711,
Inhaber    : Micha (101)
Kontostand: 345 EURO
```

```
Konto      : 4711,
Inhaber    : Micha (101) Chiron (103)
Kontostand: 345 EURO
```

```
Konto      : 4711,
Inhaber    : Chiron (103)
Kontostand: 345 EURO
```

```
Konto      : 4711,
Inhaber    : Chiron (103) Gina (104)
Kontostand: 345 EURO
```

Die Kontoinhaber werden jeweils in der zweiten Zeile eines Kontoauszugs aufgelistet.



Um einen Inhaber eines Kontos zu entfernen, ist es ebenfalls denkbar, als Argument alternativ die Kundennummer des zu entfernenden Kunden an die Operation `InhaberEntfernen(...)` zu übergeben. Dann müsste allerdings zusätzlich eine Suchoperation in der Klasse `Sparkonto` implementiert werden. Andere Möglichkeiten bestehen in der Verwendung der Klasse `Hashtable` anstelle der Klasse `ArrayList` oder in der Verwendung der Schnittstelle `Comparable`.

Besteht bei der Definition der Kardinalität eine Obergrenze, so kann diese durch Überprüfung in der Operation, die eine Beziehung hinzufügt, berücksichtigt werden, so wie die Untergrenze beim Entfernen überprüft wurde.

Die dargestellte Beziehung zwischen einem Sparkonto und seinen Inhabern war in unserem Beispiel nur einseitig navigierbar. Das bedeutet, dass das Sparkonto Zugriff auf seine Inhaber besitzt. Umgekehrt hat in unserem Modell ein Kunde keinen Zugriff auf seine Konten. Wollen Sie diesen Zugriff ermöglichen, ergänzen Sie in der Klasse Kunde ein Attribut vom Typ `ArrayList`, das einen Verweis auf seine Konten besitzt. Beachten Sie dabei, dass jede Beziehung dann sowohl bei einem Sparkonto-Objekt als auch bei einem Kunden-Objekt im Rahmen ihres Aufbaus registriert und bei ihrer Beendigung auch wieder entfernt werden muss.

Beziehungsarten

Sie haben gesehen, wie Objekte Verbindungen über eine bestimmte Dauer eingehen, diese nutzen und auch wieder beenden können. Die in der UML oder anderen Ansätzen zur objektorientierten Modellierung unterschiedenen Arten von Objektbeziehungen werden in einer Programmiersprache mit den Mitteln umgesetzt, die sie im vorhergehenden Abschnitt kennen gelernt haben.

Dennoch sollten Sie bei der Umsetzung einige Details beachten, die im Folgenden als Überblick durch konzeptionelle Beispiele dargestellt werden.

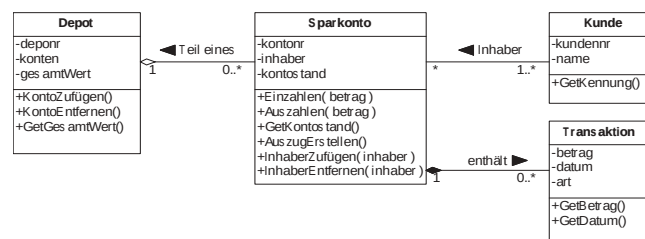


Abbildung 7.8: Sparkonto-Klasse – Beziehungsarten

Einfache Assoziation – fachliche Bindung

Eine einfache Assoziation zwischen selbstständigen Objekten, die eine fachliche Verbindung eingehen, haben Sie bereits im vorhergehenden Beispiel zwischen einem Sparkonto und den Kunden kennen gelernt.

- ✓ Die Beziehung wird als Attribut in einem oder allen beteiligten Objekten gespeichert. Für komplexe Beziehungen können auch eigene Beziehungsobjekte, die aus Assoziationsklassen erzeugt werden, gebildet werden.
- ✓ Der Aufbau und die Beendigung der Beziehung erfolgen durch Operationen.

Aggregation – Zusammenfassung unabhängiger Objekte

Eine *Aggregation* fasst unabhängige Objekte als Teile zu einem umfassenden Objekt, dem Ganzen, zusammen. Stellen Sie sich vor, dass ein Sparkonto Teil eines Depots ist.

- ✓ Die Beziehung wird zwingend in einem Attribut des Ganzen (in unserem Beispiel das Depot-Objekt) registriert.
- ✓ Der Aufbau und die Beendigung einer Beziehung erfolgen durch Operationen des Ganzen (im Beispiel das Depot-Objekt).

Komposition – Zusammenfassung abhängiger Objekte

Eine *Komposition* fasst ebenfalls Objekte in einem anderen Objekt als Ganzes zusammen. Dabei sind die Teile jedoch abhängig vom Ganzen. Betrachten Sie ein Sparkonto, das nicht nur den Kontostand, sondern wie üblich alle Transaktionen, die ebenfalls Objekte darstellen, speichert.

- ✓ Diese Beziehung registriert ebenfalls ein Attribut des Ganzen (im Beispiel das Sparkonto-Objekt).

- ✓ Die abhängigen Objekte (im Beispiel die Transaktions-Objekte) werden ausschließlich von Operationen des umfassenden Objekts (im Beispiel das Sparkonto-Objekt) erzeugt.
- ✓ Wird das umfassende Objekt vernichtet, gibt es auch für die zugeordneten Kompositions-Objekte keine Existenzberechtigung mehr (keine Transaktionen ohne zugehöriges Konto).

Mit diesen Hinweisen wenden wir uns von den im Zeitablauf änderbaren Strukturen ab und denjenigen Strukturen zu, die zeitunabhängig die Struktur einer Klasse, und damit den Aufbau der daraus erzeugten Objekte, festlegen.

Vererbung – Weitergabe von Strukturen

Im vorhergehenden Abschnitt haben wir einleitend die Beziehungen zwischen Objekten mit den Beziehungen von Menschen verglichen. Eine ähnliche Analogie legt die metaphorische Bedeutung des Begriffs *Vererbung* nahe.

Die Eigenschaften, die einem Menschen als genetische Bausteine mitgegeben wurden, sind im Grundsatz unveränderbar. Das Gleiche gilt auch für Klassen eines objektorientierten Systems. Mit der Vererbung übernimmt eine Klasse vollständig die Struktur einer anderen Klasse. Diese übernommene Struktur ist dann die Basis für Erweiterungen und Spezialisierungen und nur im Ausnahmefall Grundlage von Einschränkungen.

Identifizieren von Vererbungsstrukturen

Um die Weitergabe des Aufbaus einer Klasse zu demonstrieren, erweitern wir zunächst unser Ausgangsbeispiel, das aus der Klasse Sparkonto und der dazu in Beziehung stehenden Klasse Kunde besteht, um eine weitere Klasse. Um auch Transaktionen wie Überweisungen nachvollziehen zu können und Daueraufträge wie unsere monatliche Miete und den Eingang des Gehalts festzuhalten, ergänzen wir nun die Klasse Girokonto.

Die Operationen zum Aufbau und zum Beenden der Inhaber-Beziehung zu Kunden-Objekten wollen wir der Übersicht halber im Folgenden vernachlässigen. Der erste Aufbau erfolgt über den Konstruktor, der im Klassendiagramm hier nicht dargestellt ist.

Zusätzlich wollen wir in der Klasse Sparkonto den Zinssatz `zins s atz` als Attribut zur Berechnung der Zinsen in der Operation `ZinsenBerechnen()` verwenden.

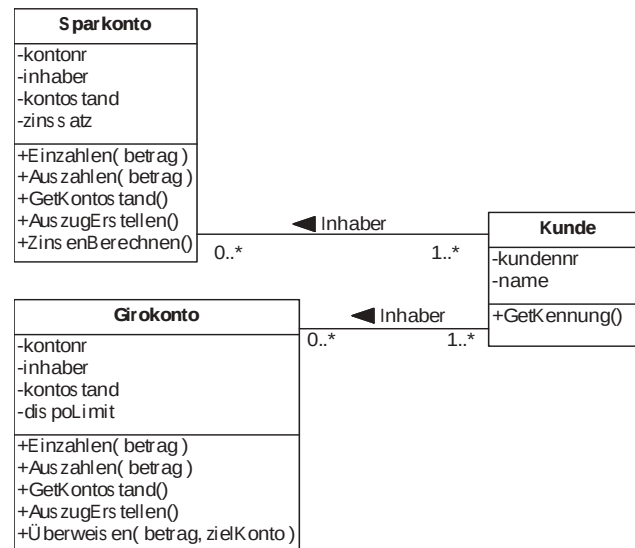


Abbildung 7.9: Konten – ohne Vererbung

Die neu aufgenommene Klasse Girokonto fällt dadurch auf, dass alle Attribute bis auf das Attribut `dispoLimit` mit den Attributen der Klasse Sparkonto übereinstimmen. Das Dispositionslimit legt fest, bis zu welchem Betrag ein Girokonto überzogen werden darf.

Weiterhin finden sich bis auf die Operation `überweisen()` alle Operationen ebenfalls im Sparkonto wieder. Die Operation zum Überweisen rufen Sie auf, um einen Betrag von einem Konto auf ein anderes Konto zu transferieren.

Richten Sie Ihr Augenmerk auf die Assoziation zwischen den Objekten Sparkonto und Kunde sowie Girokonto und Kunde, dann finden Sie auch dort eine übereinstimmende Definition. Das Attribut `inhaber`, das alle Beziehungen registriert, hat damit in beiden Klassen dieselbe Aufgabe.

Somit sind in beiden Klassen drei von vier Attributen, die auch die übereinstimmende Inhaberbeziehung beinhalten, sowie vier von fünf Operationen identisch.



Beachten Sie, dass übereinstimmende Operationsnamen noch nicht zwingend eine vollständige Übereinstimmung bedeuten. Auch wenn die Aufrufkonventionen einer Operation identisch sind, können sie sich dennoch in ihrem Verhalten, das durch die Anweisungen im Operationskörper festgelegt ist, unterscheiden.

Ebenso wie wir die Struktur von Objekten mit gleichen Attributen und Operationen in einer Klasse beschreiben, um eine mehrfache Definition zu umgehen, suchen wir nach einer Möglichkeit, die übereinstimmenden Operationen und Attribute der Klassen Sparkonto und Girokonto nur einmal im System zu beschreiben. Genau dieses Ziel erreichen Sie durch den Einsatz des Vererbungsmechanismus.

Zunächst lagern Sie alle übereinstimmenden Attribute und Operationen der beiden Klassen in eine separate Klasse Konto aus. Damit umfasst die Klasse den kleinsten gemeinsamen Nenner aller beteiligten Konten. Die Klasse Konto wird folglich auch als Oberklasse, Superklasse oder Basisklasse der Unterklassen oder Subklassen Sparkonto und Girokonto bezeichnet.

Die in der Oberklasse Konto vereinbarten gemeinsamen Attribute und Operationen werden nun über den Vererbungsmechanismus

den Unterklassen Sparkonto und Girokonto wieder zur Verfügung gestellt.

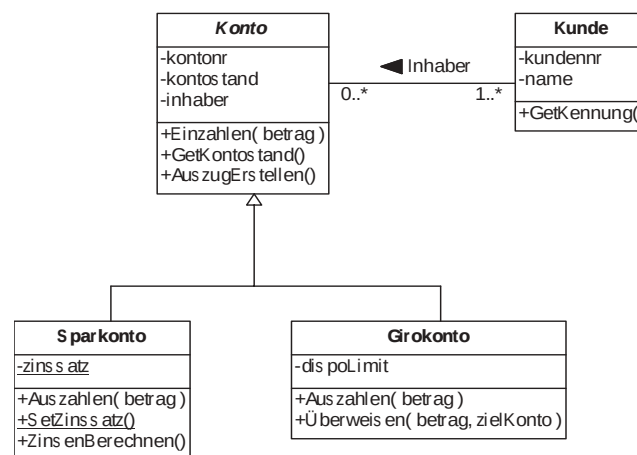


Abbildung 7.10: Konten – Vererbung

In der Abbildung stellen die Vererbungspfeile von den Unterklassen Sparkonto und Girokonto zu der Oberklasse Konto die Zugriffsrichtung dar. Die Unterklassen greifen auf die Attribute und Operationen der Oberklasse Konto zu und ersparen sich damit eine eigene Definition.

Darüber hinaus vereinfacht die Vererbung auch eine nachträgliche Änderung der Anweisungen innerhalb einer Operation. Über den Vererbungsmechanismus wirkt sie sich automatisch auf alle Unterklassen aus.

Nicht zuletzt können Sie neue Klassen mit erweiterter oder spezialisierter Struktur durch Vererbung und Anpassung mit wenig Aufwand bilden. So könnten Sie spezielle Girokonten für Geschäfts-

kunden, bei denen das Dispositionslimit automatisch nach dem durchschnittlichen Geldeingang jeden Monat angepasst wird, oder ein Zins-Girokonto, das nur online geführt wird, aus der bestehenden Klasse Girokonto ohne großen Aufwand ableiten. Ein weiterer unschätzbarer Vorteil liegt darin, dass Sie die bestehenden, erprobten Klassen getreu der Regel »Never touch a running system« dabei nicht verändern.

Die Operation `Auszahlen(...)` kann nicht in der Oberklasse `Konto` definiert werden, da sie in den Klassen `Sparkonto` und `Girokonto` unterschiedliche Anweisungen ausführt. Im `Girokonto` wird bis zum Dispositionslimit ausgezahlt, während ein `Sparkonto` nie einen negativen Kontostand aufweisen darf.

```
1: // Listing 7.10
2: using System;
3: using System.Collections;
4: public abstract class Konto {
5:     ArrayList inhaber;
6:     int kontonr;
7:     protected double kontostand;
8:
9:     public Konto(Kunde inhaber, int kontonr){
10:         this.inhaber = new ArrayList();
11:         this.inhaber.Add(inhaber);
12:         this.kontonr = kontonr;
13:     }
14:     public void Einzahlen(double betrag){
15:         kontostand += betrag;
16:     }
17:     public double GetKontostand(){
18:         return kontostand;
19:     }
20:     public void InhaberZufügen(Kunde inhaber){
21:         this.inhaber.Add(inhaber);
22:     }
23:     public void InhaberEntfernen(Kunde inhaber){
```

```
24:         if (this.inhaber.Count > 1) {
25:             this.inhaber.Remove(inhaber);
26:         }
27:     }
28:     public void AuszugErstellen(){
29:         string kennungen = null;
30:         foreach(Kunde kunde in inhaber) {
31:             kennungen += kunde.GetKennung() + " ";
32:         }
33:         Console.WriteLine("KontoNr.   : {0}\n" +
34:             "Inhaber    : {1}\n" +
35:             "Kontostand: {2:f} EURO\n",
36:             kontonr,
37:             kennungen,
38:             kontostand);
39:     }
40: }
```

Die Klasse Konto enthält exakt diejenigen Attribute und Operationen, die bisher in der Klasse Sparkonto implementiert waren.

Einzige Besonderheit ist die Bezeichnung der Klasse als `abstract`. Da die Klasse Konto nur dazu dient, die Gemeinsamkeiten der Klassen Sparkonto und Girokonto zu beherbergen, und aus ihr niemals ein Objekt erzeugt wird, können wir die Erzeugung mit dem Modifizierer `abstract` auch formal unterbinden. Der Compiler bestraft jetzt schon den Versuch mit einer Fehlermeldung.

Das Sparkonto erfährt einige Änderungen. Zunächst ist es entlastet von der Vereinbarung der Attribute und Operationen, die von der Klasse Konto zur Verfügung gestellt werden.

```
1: // Listing 7.11
2: using System;
3: public class Sparkonto : Konto{
```

Durch die Angabe der Oberklasse Konto, getrennt durch einen Doppelpunkt hinter dem Klassennamen Sparkonto, definieren Sie die Vererbungsbeziehung. Damit gibt die Oberklasse Konto Attribu-

te und Operationen an die Unterklasse Sparkonto weiter. Zusätzlich erhält die Unterklasse Sparkonto den Typ Konto als weiteren eigenen Typ.

```
4:    static double zinssatz;
```

Da Sparbücher Zinsen abwerfen, stattdessen wir jetzt unsere Klasse Sparkonto mit dem Attribut zinssatz aus. Da wir alle Sparkonten bei derselben Bank führen, besitzen Sie alle denselben Zinssatz. Ändern Sie den Zinssatz, gilt der vereinbarte Wert nun für alle Sparkonten gleichermaßen. Diese Übereinstimmung eines Attributwertes für alle aus dieser Klasse erzeugten Objekte erzwingen wir mit dem Zusatz **static**. Ein solches Klassenattribut ist, wie Sie bereits erfahren haben, an die Klasse gebunden, aber zusätzlich in allen zugehörigen Objekten wie ein Objekt-Attribut verfügbar.

Um den Zinssatz unabhängig von den erzeugten Objekten verändern zu können, definieren wir die Klassen-Operation `setZinssatz(...)`, die ebenfalls mit **static** gekennzeichnet wird.

```
5:    public static void setZinssatz(double
6:                                   zinssatz){
7:        if (zinssatz >= 0)
8:            Sparkonto.zinssatz = zinssatz;
9:    }
```

Diese Operation sorgt dafür, dass nur positive Zinssätze in dem Klassenattribut zinssatz gespeichert werden. Um das Klassenattribut vom gleichnamigen Übergabeparameter zu unterscheiden, rufen wir das an die Klasse gebundene Klassenattribut mit `Sparkonto.zinssatz` auf. Auch Klassen-Operationen können über den Klassennamen unabhängig von Objekten aufgerufen werden. So führt die Anweisung `Sparkonto.setZinssatz(2.6)` zu einem für alle Sparkonto-Objekte gültigen neuen Zinssatz.

Obwohl wir in der Oberklasse Konto einen Konstruktor definiert haben, benötigt die Unterklasse einen eigenen Konstruktor, da Konstruktoren nicht vererbt werden.

```

10:   public Sparkonto(Kunde inhaber, int kontonr)
11:       :base(inhaber, kontonr){
12:   }

```

Der Zusatz `: base(...)` bewirkt hier den Aufruf des in der Oberklasse `Konto` definierten Konstruktors mit den beiden Ganzzahlwerten `inhaber` und `kontonummer`. So können auch Konstruktoren von Unterklassen intern Konstruktoren der Oberklasse nutzen.

Die Operation `Auszahlen(...)` bleibt unverändert.

```

13:   public bool Auszahlen(double betrag){
14:       bool gedeckt;
15:       if (gedeckt = kontostand >= betrag)
16:           kontostand -= betrag;
17:       return gedeckt;
18:   }

```

Die neu vereinbarte Operation `ZinsenBerechnen()` berechnet die Zinsen vereinfacht als prozentualen Aufschlag auf den Kontostand. Das entspricht natürlich nicht dem Verfahren, das Banken anwenden.

```

19:   public void ZinsenBerechnen(){
20:       kontostand*= (100 + zinssatz) / 100;
21:   }
22: }

```

Zur Berechnung der Zinsen können Sie innerhalb einer Operation das Klassenattribut `zinssatz` genauso benutzen wie das »normale« Attribut `kontostand`.

Die Klasse `Girokonto` vereinbart ebenso wie die Klasse `Sparkonto` eine Vererbungsbeziehung zur Klasse `Konto`.

```

1: // Listing 7.12
2: using System;
3: public class Girokonto : Konto{
4:     double dispoLimit;
5:     public Girokonto(Kunde inhaber, int kontonr,

```

```
6:                double dispoLimit):  
7:                base(inhaber, kontonr){  
8:    this.dispoLimit = dispoLimit;  
9:    }
```

Der Konstruktor weist dem zusätzlichen Attribut `dispoLimit` bei der Erzeugung eines Girokonto-Objekts den als Argument des gleichnamigen Parameters übergebenen Wert zu. Die Festlegung von Inhaber und Kontonummer übernimmt wieder der Konstruktor der Oberklasse `Konto`, der mit `base(...)` in Zeile 7 aufgerufen wird.

Auszahlungen erfolgen nur, wenn das vereinbarte Dispositionslimit nicht überschritten wurde (Zeilen 11-13).

```
10: public bool Auszahlen(double betrag){  
11:     bool gedeckt;  
12:     if (gedeckt =kontostand+dispoLimit>=betrag)  
13:         kontostand -= betrag;  
14:     return gedeckt;  
15: }
```

Dabei gibt auch diese Operation eine Rückmeldung, ob ausgezahlt wurde oder nicht.

Die Operation `Überweisen(...)` transferiert einen anzugebenden Betrag auf ein beliebiges Zielkonto. Dabei erhält die Operation mit dem Übergabeparameter `zielKonto` einen Verweis auf das Konto, dem der Betrag gutgeschrieben wird.

An diesem Beispiel lässt sich die Umsetzung von einem Sequenzdiagramm in Programmcode sehr anschaulich zeigen (siehe Abbildung 7.11).

Wird die Operation `Überweisen(...)` aufgerufen, so führt sie intern die Operation `Auszahlen(...)` für das aktive Objekt und bei Erfolg die Operation `Einzahlen()` des `zielKonto`-Objektes aus.

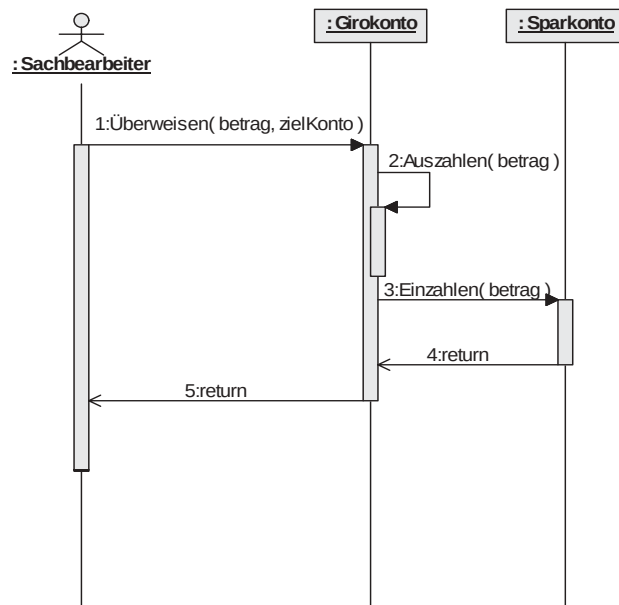


Abbildung 7.11: Sequenz: Überweisung

Als Antwort gibt sie den Erfolg oder Misserfolg als Wahrheitswert zurück.

```

16: public bool Überweisen(double betrag,
17:                        Konto zielKonto){
18:     bool gedeckt;
19:     if (gedeckt = this.Auszahlen(betrag))
20:         zielKonto.Einzahlen(betrag);
21:     return gedeckt;
22: }
23: }
  
```

Zunächst prüft die Operation, ob der Betrag von dem Girokonto, dessen Operation gerade aktiv ist, ausgezahlt werden kann. Dazu nutzen wir die Operation `Auszahlen(...)`, die mit dem Ausdruck `this.Auszahlen(...)` aufgerufen wird und einen Wahrheitswert als Antwort zurückgibt (Zeile 18), ob die Überweisung gedeckt ist. Nur wenn der Überweisungsbetrag ausgezahlt werden konnte, wird er dem Zielkonto mit dem Aufruf der Operation `zielKonto.Einzahlen(...)` gutgeschrieben (Zeile 19). Der Erfolg des Geldtransfers wird als Antwort zurückgemeldet.



Besitzt eine Klasse keine explizit angegebene Oberklasse, so wird sie aus der obersten Systemklasse `object` abgeleitet. Damit ist jede Klasse als Unterklasse der Klasse `object` zu dieser auch typkonform.

Die folgende `Main()`-Operation der Startklasse richtet ein Sparkonto und ein Girokonto ein, leistet zwei Einzahlungen, überweist 500 € auf das Sparbuch und erstellt zuletzt für beide Konten jeweils einen Kontoauszug.

```
1: // Listing 7.13
2: using System;
3: class Start {
4:     static void Main(string[] args) {
5:         Sparkonto.setZinssatz(2.8);
```

In Zeile 5 rufen Sie zunächst die Klassen-Operation `setZinssatz(...)` auf. Die Operation kann unabhängig von erzeugten Objekten über den Klassennamen aktiviert werden und legt den Zinssatz im Klassenattribut `zinssatz` für alle Sparkonten gleichermaßen auf den Wert 2.8 fest.

Erst danach richten die folgenden Anweisungen ein Kunden-Objekt und ein Sparkonto-Objekt ein, auf das 100 € eingezahlt werden.


```
6:      Kunde einKunde = new Kunde("Micha", 101);
7:      Sparkonto einSpar =
8:          new Sparkonto(einKunde, 4711);
9:      einSpar.Einzahlen(100);
```

Anschließend wird ein neues Girokonto-Objekt angelegt, das mit einem Dispositionslimit von 2000 € ausgestattet wird.

```
10:     Girokonto einGiro = new Girokonto(
11:         new Kunde("Chiron", 103),
12:         5813, 2000);
13:     einGiro.Einzahlen(300);
```

Beachten Sie, dass in Zeile 11 das Kunden-Objekt mit der Anweisung `new Kunde(...)` im Kontext der Parameter erzeugt und direkt als Übergabewert zugewiesen wird.

```
14:     einGiro.Überweisen(500, einSpar);
```

Anschließend überweist das Girokonto-Objekt `einGiro` 500 € an das Sparkonto-Objekt `einSpar`. Beachten Sie, dass die Operation `Überweisen(int betrag, Konto zielKonto)` einen Parameter vom Typ `Konto` erwartet, jedoch mit dem Objekt `einSpar` einen Wert vom Typ `Sparkonto` erhält. Jeder Klassen-Typ ist typkonform zu all seinen Oberklassen-Typen. Das bedeutet, wo ein `Konto` erwartet wird, dürfen auch spezielle Konten wie `Girokonto`-Objekte und `Sparkonto`-Objekte angegeben werden. Allerdings können dann ohne explizite Typumwandlung auch nur in der Klasse `Konto` vereinbarte Operationen aufgerufen werden.

```
15:     einGiro.AuszugErstellen();
16:     einSpar.AuszugErstellen();
```

Nachdem für beide Konten ein Auszug erstellt ist, werden die Zinsen für das Girokonto berechnet und ein weiterer Auszug erstellt.

```
17:     einSpar.ZinsenBerechnen();
18:     einSpar.AuszugErstellen();
19: }
20: }
21:
```

Die `Main()`-Operation erzeugt die folgende Bildschirmausgabe:

```
KontoNr. : 5813  
Inhaber  : Chiron (103)  
Kontostand: -200,00 EURO
```

```
KontoNr. : 4711  
Inhaber  : Micha (101)  
Kontostand: 600,00 EURO
```

```
KontoNr. : 4711  
Inhaber  : Micha (101)  
Kontostand: 616,80 EURO
```

Redefinition – Operationen werden zu Spezialisten

Wenn Sie die Auszüge der Konten betrachten, fällt auf, dass dort die Art des Kontos sowie Spezialinformationen, wie der Zinssatz beim Girokonto und das Dispositionslimit beim Sparkonto, nicht angegeben sind.

Um diese Informationen ebenfalls im Kontoauszug darzustellen, benötigen Sparkonto-Objekte und Girokonto-Objekte speziell an sie angepasste Operationen zum Erstellen eines Kontoauszugs. Die Klassen `Sparkonto` und `Girokonto` müssen also je eine spezialisierte Variante der Operation `AuszugErstellen()` vereinbaren. Die spezialisierten Operationen der Unterklassen `Sparkonto` und `Girokonto` besitzen dann denselben Namen wie die allgemein definierte Operation der Oberklasse `Konto`. Für Unterklassen-Objekte werden nun die spezialisierten Operationen aufgerufen, obwohl die allgemeine Variante der Oberklasse vererbt wurde.



Besitzen Operationen in verschiedenen Klassen denselben Namen, so liegen polymorphe Operationen vor. *Polymorphie* entstammt dem Griechischen und bedeutet »Vielgestalt«. Übertragen auf objektorientierte Systeme bedeutet dies, dass derselbe Operationsname in unterschiedlichen Klassen mit unterschied-

lichen Operationskörpern – also unterschiedlichem (vielgestaltigen) Verhalten – ausgestattet sein kann. Stehen die Klassen zusätzlich in einer Vererbungsbeziehung, wird die Operation der Oberklasse überschrieben. Daher lautet der Fachterminus für das Überschreiben einer Operation in einer Vererbungsbeziehung *Redefinition*.

Neben der Redefinition von Operationen bietet C# auch die Möglichkeit, die Standardoperatoren (+, -, *, / und andere) für eigene Klassen mit neuer Funktionalität zu versehen. Dieses Überschreiben von Operatoren bezeichnen wir als Überladen oder *Overloading*.

Zunächst geben Sie die Operation `AuszugErstellen()` der Oberklasse `Konto` zum Überschreiben frei, indem Sie dort den Modifizierer **virtual** angeben. Virtuelle Operationen werden erst zur Laufzeit passend zum jeweiligen Objekt ausgeführt. Auf diesen Aspekt kommen wir im nächsten Beispiel zurück.

```
1: // Listing 7.14
2: using System;
3: class Konto{
4:     /*...*/
5:     virtual public void AuszugErstellen(){
6:         /*...*/
7:     }
8:     /*...*/
9: }
```

Erweitern Sie anschließend die Klasse `Sparkonto`, indem Sie auch dort eine Operation `AuszugErstellen()` vereinbaren, die Sie mit dem Modifizierer **override** kennzeichnen. Die Kennzeichnung **override** bedeutet, dass eine Operation mit der gleichen Signatur, die in der Oberklasse `Konto` gegeben ist, in der Unterklasse `Sparkonto` neu definiert wird.

```
1: // Listing 7.15
2: using System;
3: class Sparkonto : Konto{
4:     /*...*/
5:     override public void AuszugErstellen(){
```

Die Spezialisierung für ein Sparkonto besteht darin, dass der Kontoauszug zusätzlich eine Kopfzeile erhält, die die Kontoart Sparkonto und den definierten Zinssatz anzeigt.

```
6:         Console.WriteLine("Sparkonto : ({0}%)",  
7:                             zinssatz);
```

Der Rest des Kontoauszugs ist identisch mit den Ausgaben der Operation `AuszugErstellen()`, die in der Oberklasse `Konto` vereinbart ist. Um diese jetzt überschriebene Operation innerhalb der neu vereinbarten Operation dennoch aufzurufen, verwenden Sie den `base`-Zeiger.

```
8:         base.AuszugErstellen();  
9:     }  
10:    /*...*/  
11: }
```

Der `base`-Zeiger verweist auf die Oberklasse `Konto`, so dass `base.AuszugErstellen()` die überschriebene Operation der Klasse `Konto` innerhalb der neu vereinbarten Operation der Klasse `Sparkonto` aufruft.

Erweitern Sie jetzt noch die Klasse `Girokonto` um eine Operation `AuszugErstellen()`, die ebenfalls eine Kopfzeile einfügt. In der Kopfzeile zeigt der Auszug jetzt die Kennzeichnung `Girokonto`, gefolgt von dem Dispositionslimit, das für das Konto vereinbart wurde.

```
1: // Listing 7.16  
2: using System;  
3: public class Girokonto : Konto{  
4:     /*...*/  
5:     override public void AuszugErstellen(){  
6:         Console.WriteLine("Girokonto : Dispo({0})",  
7:                             dispoLimit);  
8:         base.AuszugErstellen();  
9:     }  
10:    /*...*/  
11: }
```

Unterhalb der Kopfzeile werden durch Aufruf der »alten« Operation `AuszugErstellen()` die Angaben zur Kontonummer und zu den Kontoinhabern sowie der Kontostand ausgegeben.

```
1: // Listing 7.17
2: using System;
3: class Start {
4:     static void Main(string[] args) {
5:         Sparkonto.setZinssatz(2.7);
6:         Konto einGiro = new Girokonto(
7:             new Kunde("Anke", 105),
8:             5813,
9:             3000);
```

Nachdem der Zinssatz für alle Sparkonten auf 2,7 % festgelegt wurde, wird ein `Girokonto`-Objekt erzeugt, das als Übergabewert einen Verweis auf ein neu erzeugtes `Kunden`-Objekt als Inhaber erhält. Der Typ der Variablen `einGiro` ist `Konto`. Diese Variable vom Typ `Konto` verweist auf ein Objekt vom Typ `Girokonto`. Das ist in diesem Fall zulässig, da die Klasse `Girokonto` durch die Vererbungsbeziehung zur Klasse `Konto` nicht nur Attribute und Operationen, sondern darüber hinaus auch den Typ der Oberklasse `Konto` übernimmt.

```
10:         Konto einSpar = new Sparkonto(
11:             new Kunde("Gina", 104),
12:             4711);
```

Ebenso wird ein Objekt der Klasse `Sparkonto` erzeugt. Der Verweis auf dieses Objekt wird wiederum einer Variablen vom Typ `Konto` zugewiesen.

Da die Variable vom Typ `Konto` nur die Operationen kennt, die in dieser Klasse gegeben sind, wandeln Sie den Typ `Konto` der Variablen durch den Cast-Operator (`Girokonto`) in den Typ des `Girokontos`, um die in der Klasse `Girokonto` neu hinzugekommene Operation `Überweisen(...)` aufrufen zu können.



Klassen sind immer typkonform zu all ihren Oberklassen. Durch Einsatz des Cast-Operators können Variablen, die auf ein Objekt vom Typ einer ihrer Unterklassen verweisen, auch auf die Operationen zugreifen, die nicht im Typ der Oberklasse vereinbart sind. Der Typ einer Variablen bestimmt, welche Operations-Signaturen für die Variable zulässig sind. Das über diese Variable referenzierte Objekt bestimmt hingegen die Variante dieser Operation, die ausgeführt wird.

Ebenso verfahren Sie mit dem Aufruf der Operation `Auszahlen(...)`, da sie in der Klasse `Sparkonto`, nicht aber in der Klasse `Konto` vereinbart ist.

```
13:      ((Girokonto)einGiro).Überweisen(300,  
14:                                     einSpar);  
15:      ((Sparkonto)einSpar).Auszahlen(180);  
16:      einGiro.AuszugErstellen();  
17:      einSpar.AuszugErstellen();  
18:  }  
19: }
```

Im Gegensatz zu den nur in den Unterklassen definierten Operationen `Überweisen(...)` und `Auszahlen(...)` rufen Sie die Operation `AuszugErstellen()` ohne den Cast-Operator auf, da ihre Aufrufkonventionen ja in der Klasse `Konto` ebenfalls bekannt sind. Beachten Sie, dass dennoch die speziellen Auszüge für Girokonten und Sparkonten erstellt werden.



Unabhängig vom Typ einer Variablen werden immer die Operationen vom Typ des Objekts aufgerufen, wenn die Operationen mit `virtual` und `override` gekennzeichnet sind.

Das Programm stellt jetzt individuelle Kontoauszüge für `Sparkonto`-Objekte und `Girokonto`-Objekte dar.

Girokonto : Dispo(3000)
KontoNr. : 5813
Inhaber : Anke (105)
Kontostand: -300 EURO

Sparkonto : (2,7%)
KontoNr. : 4711
Inhaber : Gina (104)
Kontostand: 120 EURO

Mit der Vererbung steht Ihnen ein mächtiges Sprachmittel zur Verfügung, um Erweiterungen und Spezialisierungen bestehender Klassen zu implementieren.

Schnittstellen – mehr Schein als Sein

Neben der Vererbung von Klassen kennt C# auch noch die Vererbung von *Schnittstellen* (Interfaces). Dabei stellen Schnittstellen spezielle Klassen dar, die keine Attribute und Operationskörper enthalten. Sie definieren lediglich Operationsköpfe, die unter einem Schnittstellennamen zusammengefasst sind.

Damit vereinbart eine Schnittstelle einen Typ, der lediglich die Aufrufkonventionen seiner Operationen, nicht aber ihr Verhalten beschreibt.

Erbt eine Klasse eine Schnittstelle, so verpflichtet sie sich, alle Operationsköpfe der Schnittstelle vollständig mit Operationskörpern zu versehen. Im Gegenzug erhält die Klasse zusätzlich zu ihrem eigenen Typ wie bei Klassen auch den Typ der Schnittstelle, über den alle in der Schnittstelle definierten Operationen aufgerufen werden können.

Vereinbarung von Schnittstellen

Die Schnittstelle *IVerzinsbar* legt fest, dass alle Klassen, die diese Schnittstelle implementieren, gewährleisten, eine Operation *ZinsenBerechnen()* zu definieren.

**Abbildung 7.12:** Schnittstelle – IVerzinsbar

Der Kreis im UML-Klassendiagramm der obigen Abbildung unterscheidet Schnittstellen von Klassen.

```
1: // Listing 7.18
2: public interface IVerzinsbar {
3:     void ZinsenBerechnen();
4: }
```

Die Schnittstelle IVerzinsbar vereinbart die Aufrufkonventionen für eine Operation ZinsenBerechnen(). Die Klasse Sparkonto als Vermögensanlage, die Zinsen abwirft, erbt diese Schnittstelle und vereinbart eine Operation ZinsenBerechnen(), die in der Klasse Sparkonto jetzt auch einen Operationskörper beinhaltet.

Vererbung und Implementierung von Schnittstellen

**Abbildung 7.13:** Schnittstelle – IVerzinsbar mit Implementierung

Da die Operation ZinsenBerechnen() in der Klasse Sparkonto bereits besteht, zeigt das folgende Listing lediglich die Übernahme der Schnittstelle IVerzinsbar in die Klasse Sparkonto.


```
1: // Listing 7.19
2: using System;
3: public class Sparkonto : Konto, IVerzinsbar{
4:     /*...*/
5:     void ZinsenBerechnen(){
6:     {
7:         /*...*/
8:     }
9: }
```

Dabei werden Schnittstellen genauso gehandhabt wie Klassen, zu denen eine Vererbungsbeziehung besteht (Zeile 3). Die Klasse erbt neben den Attributen und Operationen auch alle Typen der Schnittstelle. Dementsprechend sind Sparkonto-Objekte typkonform zum Typ IVerzinsbar.

Im nächsten Schritt erweitern wir unsere Kontenverwaltung um die Klasse Wertpapier, die wie ein Sparkonto auch eine verzinsbare Vermögensanlage darstellt. Sie erbt folglich ebenso wie die Klasse Sparkonto das Interface IVerzinsbar und implementiert die Operation ZinsenBerechnen().

Die Attribute repräsentieren die Wertpapiernummer (wpnr), den Inhaber (inhaber), den Nominalwert (nominalwert), den Zinssatz (zinssatz), die Laufzeit (laufzeit), den aktuellen Kurswert (kurs) und einen Verweis auf ein Verrechnungskonto (verKonto), dem die Zinsen gutgeschrieben werden.

Operationen sehen Sie vor zur Aktualisierung des Kurswerts (SetKurs()) und zum Erstellen eines Auszugs (siehe Abbildung 7.14).

Um Konten und Wertpapiere unter einem Typ zusammenfassen zu können, vereinbaren wir das Interface IDepotfähig. Damit besitzen Sparkonto-Objekte, Girokonto-Objekte und Wertpapier-Objekte denselben Typ IDepotfähig.

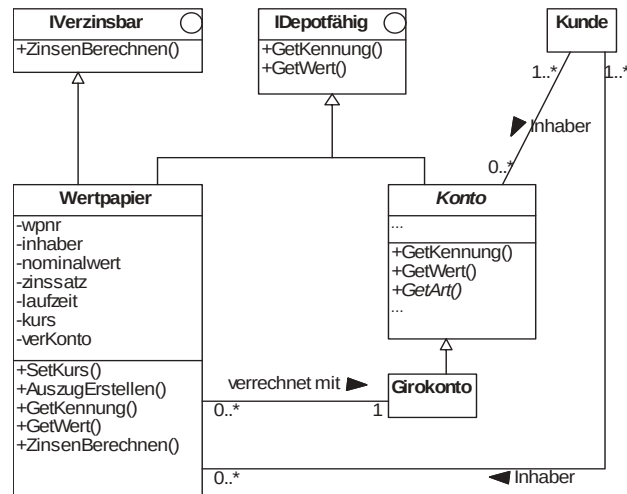


Abbildung 7.14: Schnittstelle – IDepotfähig mit Implementierung

Mit der Schnittstellenvererbung verpflichten sich die Klassen Konto und Wertpapier, die Operationen `GetKennung()` und `GetWert()` zu implementieren.

```

1: // Listing 7.20
2: public interface IDepotfähig {
3:     string GetKennung();
4:     double GetWert();
5: }
  
```

Das Interface IDepotfähig legt fest, dass alle Klassen, die dieses Interface erben, die Operationen `GetKennung()` und `GetWert()` implementieren müssen.

```
1: // Listing 7.21
2: using System;
3: using System.Collections;
4: public class Wertpapier: IDepotfähig, IVerzinsbar{
5:     int wpnr;
6:     ArrayList inhaber;
7:     double laufzeit;
8:     double nominalwert;
9:     double kurs;
10:    Girokonto verKonto;
11:    double zinssatz;
```

Die Attribute inhaber (Zeile 6) und verKonto (Zeile 10) definieren Assoziationen zu einem Kunden-Objekt, das auf den Inhaber verweist, und zu einem Girokonto-Objekt, dem die Wertpapierzinsen gutgeschrieben werden.

Bei der Erzeugung eines Objekts legen Sie über den Konstruktor alle Attributwerte fest.

```
12:    public Wertpapier(int wpnr,
13:                      Kunde inhaber,
14:                      double laufzeit,
15:                      double nominalwert,
16:                      double kurs,
17:                      Girokonto verKonto,
18:                      double zinssatz){
19:        this.inhaber = new ArrayList();
20:        this.wpnr = wpnr;
21:        this.inhaber.Add(inhaber);
22:        this.laufzeit = laufzeit;
23:        this.nominalwert = nominalwert;
24:        this.kurs = kurs;
25:        this.verKonto = verKonto;
26:        this.zinssatz = zinssatz;
27:    }
```

Da der Kurswert eines Wertpapiers schwankt, können Sie jederzeit über die Operation `SetKurs(...)` einen veränderten Wert eintragen.

```
28: public void SetKurs(double kurs){
29:     this.kurs = kurs;
30: }
```

Die Operation `AuszugErstellen()` gibt die Attributwerte auf dem Bildschirm aus.

```
31: public void AuszugErstellen(){
32:     Console.WriteLine("Wertpapiernr. : {0}\n" +
33:         "Inhaber      : {1}\n" +
34:         "Laufzeit      : {2} Jahre\n" +
35:         "Nominalwert   : {3:f} EURO\n" +
36:         "Kurswert      : {4:f} EURO\n" +
37:         "Zinssatz      : {5}%\n",
38:         wpnr,
39:         ((Kunde)inhaber[0]).GetKennung(),
40:         laufzeit,
41:         nominalwert, kurs,
42:         zinssatz);
43: }
```

Beachten Sie, dass der Zugriff auf ein `ArrayList`-Objekt (Zeile 39) über einen Indizierer erfolgt, über den auf ein `Inhaber`-Objekt zugegriffen wird. Da die `ArrayList`-Elemente immer vom Typ `object` sind, wandeln wir den Variablen-Typ `object` zunächst in den Typ des referenzierten Objekts `Kunde` um. Erst mit der Umwandlung erhalten wir die Möglichkeit, die Operation `GetKennung()` für das `Inhaber`-Objekt aufzurufen.

Die Operationen `GetWert()` und `GetKennung()` implementieren die gleichnamigen Operations-Schnittstellen von `IDepotfähig`.

```
44: public double GetWert(){
45:     return kurs;
46: }
47: public string GetKennung() {
48:     return "Wertpapier " +
```

```

49:         wpnr + ", " +
50:         ((Kunde)inhaber[0]).GetKennung();
51:     }

```

Die Operation `ZinsenBerechnen()` implementiert die Operationsschnittstelle von `IVerzinsbar`.

```

52:     public void ZinsenBerechnen(){
53:         double zinsen= nominalwert* zinssatz / 100;
54:         verKonto.Einzahlen(zinsen);
55:     }
56: }

```

Der berechnete Zins wird dem vereinbarten Verrechnungskonto `verKonto`, das bei der Einrichtung des Wertpapier-Objekts über den Konstruktor vereinbart wurde, gutgeschrieben (Zeile 54).

Damit sowohl Sparkonto- als auch Girokonto-Objekte in einem Depot verwaltet werden können, vereinbaren Sie in deren Oberklasse `Konto` eine Vererbungsbeziehung zur Schnittstelle `IDepotfähig` und implementieren die Operationsschnittstellen `GetKennung()` und `GetWert()`.

```

1: // Listing 7.22
2: using System;
3: using System.Collections;
4: public abstract class Konto : IDepotfähig {
5:     /*...*/

```

Die Operation `GetKennung()` erzeugt eine Zeichenfolge, die aus der Kontoart, der Kontonummer und den Kennungen der Kontoinhaber besteht.

```

6:     public string GetKennung() {
7:         string kennung = GetArt()+ kontonr+", ";
8:         foreach (Kunde kunde in inhaber) {
9:             kennung += kunde.GetKennung();
10:        }
11:         return kennung;
12:     }

```

Die Art des Kontos ermittelt die Operation durch den internen Aufruf der Operation `GetArt()`, die ebenfalls in dieser Klasse vereinbart ist.

```
13:    public abstract string GetArt();
```

Die Operation `GetArt()` ist mit dem Modifizierer **abstract** gekennzeichnet und enthält keinen Operationskörper. Eine solche abstrakte Operation darf nur in abstrakten Klassen vereinbart werden und muss in den konkreten Unterklassen vollständig implementiert werden. In der Klasse `Konto` stellt diese Operation lediglich einen Platzhalter dar. Folglich müssen die Unterklassen `Sparkonto` und `Girokonto` die Operation `GetArt()` vollständig implementieren.

Die abstrakte Operation wird niemals aufgerufen, da der interne Aufruf der Operation `GetArt()` innerhalb der Operation `GetKennung()` (Zeile 7) sich immer auf ein Objekt der konkreten Klassen `Sparkonto` oder `Girokonto` bezieht. Die Klassen `Girokonto` und `Sparkonto` werden wir mit einer vollständig implementierten Variante der Operation `GetArt()` versehen, so dass die Operation `GetKennung()` immer die Variante des vorliegenden Objekts vom Typ `Sparkonto` oder `Girokonto` aufruft.

Die Operation `GetWert()`, deren Implementierung vom geerbten Interface `IDepotfähig` erzwungen wird, delegiert die Ermittlung des Werts an die bereits bestehende Operation `GetKontostand()`.

```
14:    public double GetWert(){  
15:        return GetKontostand();  
16:    }  
17: }
```

Jetzt erweitern Sie die Klassen `Sparkonto` und `Girokonto` noch um die Operation `GetArt()`, deren Rückgabewert über die Operation `GetKennung()` der Klasse `Konto` eingesetzt wird.

```
1: // Listing 7.23  
2: using System;  
3: public class Girokonto : Konto{  
4:     /*...*/  
5:     override public string GetArt(){
```

```

6:         return "Girokonto ";
7:     }
8: }

```

Die Operationen `GetArt()` in den Klassen `Sparkonto` und `Girokonto` sind mit dem Modifizierer **override** versehen, da sie die abstrakte Platzhalter-Operation der Klasse `Konto` ersetzen.

```

1: // Listing 7.24
2: using System;
3: public class Sparkonto : Konto, IVerzinsbar{
4:     /*...*/
5:     override public string GetArt(){
6:         return "Sparkonto ";
7:     }
8: }

```

Die Klassen `Sparkonto` und `Wertpapier` besitzen jetzt den Typ `IVerzinsbar` und können als verzinsbare Vermögensanlagen zusammengefasst werden.

Schnittstellentyp als Attributtyp

Die Klassen `Sparkonto`, `Girokonto` und `Wertpapier` sind darauf vorbereitet, dass sie in einem `Depot` als depotfähige Vermögenswerte aufgenommen werden. Ein `Depot` soll dabei beliebig viele depotfähige Elemente ($0..*$) enthalten. Ein depotfähiges Element hingegen darf maximal einem `Depot` ($0..1$) zugeordnet werden.



Abbildung 7.15: Schnittstellen – `IDepotfähig` als Typ von Depot-elementen

Das Depot besitzt eine Depotnummer (`deponr`) und soll in der Lage sein, depotfähige Vermögenswerte (`vermWerte`) in das Depot aufzunehmen (`Einstellen()`), sie wieder aus dem Depot zu entfernen (`Entnehmen()`), einen Auszug zu erstellen (`AuszugErstellen()`) und für alle verzinsbaren Vermögenswerte im Depot die Zinsberechnung anzustoßen (`ZinsenBerechnen()`).

```
1: // Listing 7.25
2: using System;
3: using System.Collections;
4: public class Depot{
5:     int deponr;
6:     ArrayList vermWerte;
7:     public Depot(int deponr) {
8:         this.deponr = deponr;
9:         this.vermWerte = new ArrayList();
10:    }
```

Die im Depot enthaltenen Vermögenswerte werden als assoziierte Objekte im `ArrayList`-Objekt `vermWerte` (Zeile 6) hinterlegt. Der `Individual`-Konstruktor legt die Depotnummer fest und erzeugt das `ArrayList`-Objekt `vermWerte`, das später die Vermögenswerte aufnimmt.

Die Operation `Einstellen(...)` fügt dem Depot einen neuen Vermögenswert hinzu. Dazu erhält sie als Übergabewert ein depotfähiges Objekt vom Typ der Schnittstelle `IDepotfähig`.

```
11: public void Einstellen(IDepotfähig vermWert){
12:     this.vermWerte.Add(vermWert);
13: }
```

Durch den Aufruf der Operation `Add(...)` (Zeile 12) wird der übergebene Vermögenswert dem `ArrayList`-Objekt `vermWerte` hinzugefügt.

Mit der Operation `Entnehmen(...)`, die ebenfalls einen Verweis auf das zu entnehmende Objekt erhält, entnehmen Sie dem Depot einen Vermögenswert.


```

14:  public void Entnehmen(IDepotfähig vermWert){
15:      this.vermWerte.Remove(vermWert);
16:  }

```

Die Operation `Remove(...)` entfernt den angegebenen Vermögenswert aus dem `ArrayList`-Objekt `vermWerte`.

Die Operation `AuszugErstellen()` nutzt die im Interface `IDepotfähig` vereinbarten Operationen `GetKennung()` (Zeile 23) und `GetWert()` (Zeile 24), um eine Übersicht aller im Depot enthaltenen Vermögenswerte zu erstellen.

```

17:  public void AuszugErstellen(){
18:      double gesamtWert = 0;
19:      Console.WriteLine("\nAuszug Depot: {0}",
20:                          deponr);
21:      foreach(IDepotfähig vermWert in vermWerte){
22:          Console.WriteLine("- {0} , {1:f} EURO",
23:                          vermWert.GetKennung(),
24:                          vermWert.GetWert());
25:          gesamtWert += vermWert.GetWert();
26:      }
27:      Console.WriteLine(
28:          "Gesamtwert: {0:f} EURO\n", gesamtWert );
29:  }

```

Nach der Bildschirmausgabe der Kopfzeile (Zeilen 19-20), die die Depotnummer enthält, durchkämmt die `foreach`-Schleife alle Vermögenswerte des `ArrayList`-Objekts `vermWerte`, gibt die ermittelte Kennung (Zeile 23) und den Wert (Zeile 24) auf dem Bildschirm aus und summiert die Werte zu einem Gesamtwert auf (Zeile 25), der anschließend ausgegeben wird (Zeilen 27-28).

Die Operation `ZinsenBerechnen()` aktiviert für alle verzinsbaren Vermögenswerte im Depot die Berechnung der Zinsen.

```

30:  public void ZinsenBerechnen() {
31:      foreach(IDepotfähig vermWert in vermWerte){
32:          if (vermWert is IVerzinsbar){
33:              ((IVerzinsbar)vermWert).

```

```
34:                ZinsenBerechnen();  
35:            }  
36:        }  
37:    }  
38: }
```

Dazu prüft die Operation in Zeile 32 für jeden Vermögenswert, ob er vom Typ `IVerzinsbar` ist. Vom Typ `IVerzinsbar` sind alle Vermögens-Objekte, die die geerbte Schnittstelle `IVerzinsbar` implementieren, also Sparkonto-Objekte und Wertpapier-Objekte. Der zur Prüfung eingesetzte *is-Operator* erkennt alle Typen einer Klasse, die sich entlang der Vererbungshierarchie angesammelt haben.

Um die Operation `ZinsenBerechnen()` für die verzinsbaren Vermögenswerte aufzurufen, muss der Typ der Elementvariablen in der `ArrayList (object)` in den Typ der referenzierten Objekte `IVerzinsbar` umgewandelt werden. Jetzt ist die Operation erreichbar und wird ausgeführt.

Depotverwaltung – Schnittstellen im Überblick

Die folgende Abbildung zeigt die Struktur unseres Depots noch einmal im Überblick. Alle Kontenarten und Wertpapiere können Sie als depotfähige Vermögenswerte in das Depot einstellen. Von diesen Vermögenswerten sind jedoch nur Sparkonten und Wertpapiere verzinsbar (siehe Abbildung 7.16).

Schnittstellen ermöglichen also die gemeinsame Behandlung von Objekten, die zunächst nicht typkonform sind, wie die Klassen Wertpapier und Sparkonto in unserem Beispiel. Durch die Vererbungsbeziehung zu einer gemeinsamen Schnittstelle erhalten beide einen zusätzlichen Typ `IVerzinsbar`, für den sie die in der Schnittstelle vereinbarte Operation `ZinsenBerechnen()` aktivieren können.

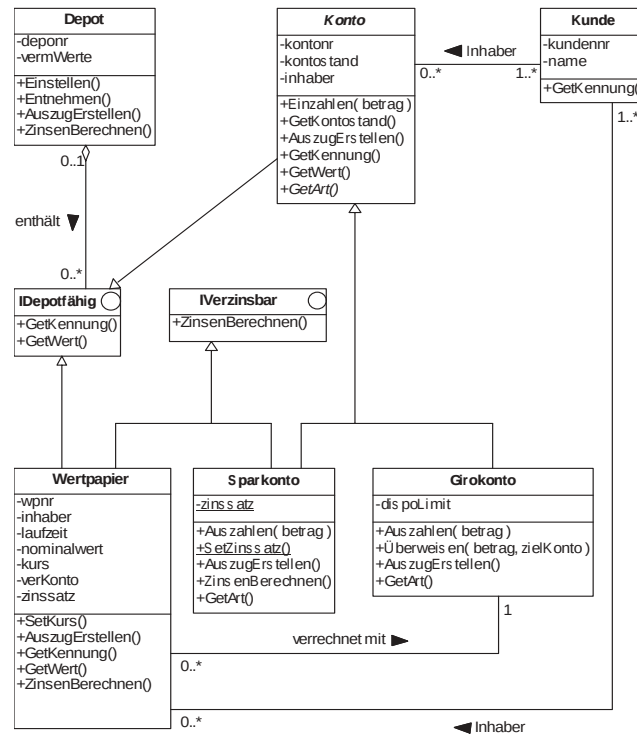


Abbildung 7.16: Schnittstellen – Schaffung typkonformer Klassen

Die folgende Startklasse testet unser Depot. Nachdem der Zinssatz für alle Sparkonten auf 2,7 % festgelegt wurde, erzeugt die `Main()`-Operation ein `Sparkonto`-Objekt, ein `Girokonto`-Objekt und ein `Wertpapier`-Objekt. Anschließend erhält das `Sparkonto` eine Einzahlung von 100 €, bevor das Depot alle drei Vermögenswerte aufnimmt.

Das Depot stößt daraufhin die Zinsberechnung für alle in ihm enthaltenen zinsfähigen Vermögenswerte an und gibt einen Depotauszug auf dem Bildschirm aus. Zur Kontrolle werden schließlich die detaillierten Auszüge der drei Vermögenswerte erstellt.

```
1: // Listing 7.26
2: using System;
3: class Start {
4:     static void Main(string[] args) {
5:         Sparkonto.setZinssatz(2.7);
6:
7:         Sparkonto einSpar = new Sparkonto(
8:             new Kunde("Anke", 105), 6813);
9:
10:        Girokonto einGiro = new Girokonto(
11:            new Kunde("Mike", 101), 4711, 4000);
12:
13:        Wertpapier einWP = new Wertpapier(
14:            11, new Kunde("Gina", 104),
15:            3, 200, 265, einGiro, 4.5);
16:
17:        einSpar.Einzahlen(100);
```

Sparkonten und Girokonten arbeiten wie bisher. Das Wertpapier-Objekt einWP erhält über den Konstruktor (Zeilen 13-15) die Wertpapiernummer 11, als Inhaber einen Verweis auf das neu erzeugte Kunden-Objekt »Gina«, eine Laufzeit von drei Jahren, den Nominalwert von 200 €, einen Kurswert von 265 € und als Verrechnungskonto einen Verweis auf das Girokonto einGiro. Der Zinssatz ist auf 4,5 %, bezogen auf den Nominalwert, festgelegt.

Das Sparkonto einSpar erhält eine Einzahlung von 100 € (Zeile 17), bevor ein Depot angelegt wird.

```
18:        Depot einDepot = new Depot(1);
19:        einDepot.Einstellen(einSpar);
20:        einDepot.Einstellen(einGiro);
21:        einDepot.Einstellen(einWP);
```

Das Depot-Objekt `einDepot` erhält die Depotnummer 1 (Zeile 18). Anschließend werden das Sparkonto-Objekt `einSpar` (Zeile 19), das Girokonto-Objekt `einGiro` (Zeile 20) und das Wertpapier-Objekt `einWP` (Zeile 21) in das Depot eingestellt.

Der Aufruf der Operation `ZinsenBerechnen()` des Depot-Objekts `einDepot` in Zeile 22 aktiviert eine Zinsberechnung für das Sparkonto `einSpar`, das 2,7 % von 100 € dem Konto gutschreibt und somit ein Guthaben von 102,70 € aufweist. Für das Wertpapier-Objekt `einWP` führt die Zinsberechnung von 4,5%, gerechnet auf den Nominalwert von 200 €, zu einem Zinswert von 9 €, der dem vereinbarten Girokonto `einGiro` gutgeschrieben wird. Damit besitzt das Girokonto `einGiro` einen Kontostand von 9 €.

```
22:      einDepot.ZinsenBerechnen();
23:      einDepot.AuszugErstellen();
24:      einSpar.AuszugErstellen();
25:      einGiro.AuszugErstellen();
26:      einWP.AuszugErstellen();
27:  }
28: }
```

Der Auszug, den die Operation `AuszugErstellen()` (Zeile 23) für das Depot `einDepot` erstellt, zeigt für alle im Depot enthaltenen Vermögenswerte die Kennung und den Wert an. Die letzte Zeile enthält den über alle Einzelposten aufsummierten Gesamtwert des Depots.

Anschließend sehen Sie zur Kontrolle die Auszüge der einzelnen Vermögenswerte.

Führen Sie die Startklasse aus, dann erstellt die `Main()`-Operation nach Zuordnung der Konten und des Wertpapiers sowie der Berechnung der Zinsen für das Sparkonto und das Wertpapier die folgenden vier Auszüge:

Auszug Depot: 1

- Sparkonto 6813, Anke (105) , 102,70 EURO
- Girokonto 4711, Mike (101) , 9,00 EURO
- Wertpapier 11, Gina (104) , 265,00 EURO

Gesamtwert: 376,70 EURO

Sparkonto : (2,7%)

KontoNr. : 6813

Inhaber : Anke (105)

Kontostand: 102,70 EURO

Girokonto : Dispo(4000)

KontoNr. : 4711

Inhaber : Mike (101)

Kontostand: 9,00 EURO

Wertpapiernr. : 11

Inhaber : Gina (104)

Laufzeit : 3 Jahre

Nominalwert : 200 EURO

Kurswert : 265 EURO

Zinssatz : 4,5%

Das Sparkonto enthält die 100 € zuzüglich der berechneten 2,70 € Zinsen, das Girokonto die Gutschrift der Wertpapierzinsen von 9 €. Das Wertpapier besitzt immer noch den Kurswert von 265 €, der im Konstruktor angegeben wurde. Damit ergibt sich ein Gesamtwert von 376,70 €.

Die zur Kontrolle ausgegebenen Auszüge für das Sparkonto, das Girokonto und das Wertpapier bestätigen die im Depotauszug enthaltenen Angaben.

Namensräume – Klassen organisieren

Nachdem Sie bereits Anwendungen mit mehreren Klassen und Schnittstellen erstellt haben, lernen Sie in diesem Abschnitt, wie

Sie große Anwendungen mit vielen Klassen so strukturieren, dass Sie den Überblick nicht verlieren.

C# bietet Ihnen mit seinen Namensräumen (Namespaces) die Möglichkeit, Klassen und Interfaces zu Organisationseinheiten zusammenzufassen.



Anders als in anderen Programmiersprachen ist die Zuordnung von Klassen zu Namensräumen eine virtuelle Organisation, die nicht mit der physischen Organisation übereinstimmen muss. Organisieren Sie für eine bessere Übersicht Ihre Klassen dennoch immer so, dass Sie für jeden Namensraum ein eigenes Unterverzeichnis definieren, in dem Sie alle zugehörigen Klassen ablegen.

Sie finden ähnliche Strukturen wie Namensräume auch in Organigrammen, die Organisationsstrukturen von Unternehmungen übersichtlich strukturieren. Organigramme ordnen die Mitarbeiter bestimmten Stellen zu, die eine Aufgabenbeschreibung enthalten. Die Stellen werden dann nach bestimmten Kriterien (z.B. funktional in Einkauf, Produktion, Verkauf) zu Abteilungen zusammengefasst. Abteilungen können auf einer höheren Ebene wiederum nach einem bestimmten Kriterium zu übergeordneten Abteilungen zusammengefasst werden.

Vergleichen Sie Mitarbeiter mit aktiven Objekten und Stellenbeschreibungen mit Klassen, dann sind die Abteilungen, in denen sie zusammengefasst werden, die Namensräume, die ihrerseits wieder auf einer höheren Ebene zu Namensräumen gruppiert werden können.

using – Einbinden von Namensräumen

Zum Einstieg haben Sie bereits gelernt, wie Sie mit der *using*-Anweisung Klassen fremder Namensräume zur Benutzung in eigenen Klassen verfügbar machen. Bereits in den ersten Beispielen, in de-

nen Sie eine Auto-Klasse erstellt haben, haben Sie die `using`-Anweisung eingesetzt, um den Namensraum `System` einzubinden, der die Klasse `Console` für Bildschirmausgaben und Tastatureingaben zur Verfügung stellt.

namespace – Zuordnen zu Namensräumen

In diesem Abschnitt organisieren wir unsere Klassen in Namensräumen und legen fest, welche Kontakte zwischen Klassen sowie zwischen Operationen und Attributen zugelassen werden.

Ein UML-Klassendiagramm besitzt mit seinen Paketen eine Strukturierungseinheit, die den Namensräumen von C# und .NET entsprechen. Namensräume sind eine logische oder virtuelle Zusammenfassung von Typen. Innerhalb eines Namensraums ist jeder Typname eindeutig, so dass keine identischen Typbezeichnungen vorkommen können.

Wollen wir Kriterien aufstellen, nach denen Klassen, Interfaces und andere Typen zu Namensräumen zusammengefasst werden können, so bieten sich zwei elementare Möglichkeiten. Entweder Sie gruppieren die Elemente nach fachlicher Zusammengehörigkeit oder Sie ordnen nach konstruktionstechnischen Gesichtspunkten, nach denen möglichst wenige Beziehungen zwischen Elementen verschiedener Namensräume bestehen sollten (Prinzip der schmalen Kopplung von Moduln).

Für die Depotverwaltung ordnen wir die Klassen und Schnittstellen nach fachlicher Zusammengehörigkeit vier Namensräumen zu. Um die mit der `uses`-Anweisung einzubindenden Namensräume leichter identifizieren zu können, stellt das Klassendiagramm für alle Assoziationen die Zugriffsrichtung (Navigierbarkeit) dar. So muss die Klasse `Konto` den Namensraum `Personen` einbinden, da sie die Inhaber-Beziehung aufbaut und den Inhaber registriert. Sie greift dazu auf Informationen der Klasse `Kunde` zu.

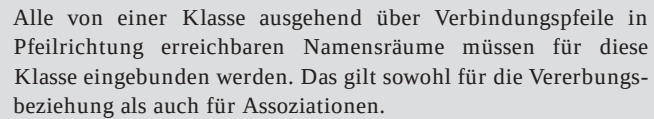
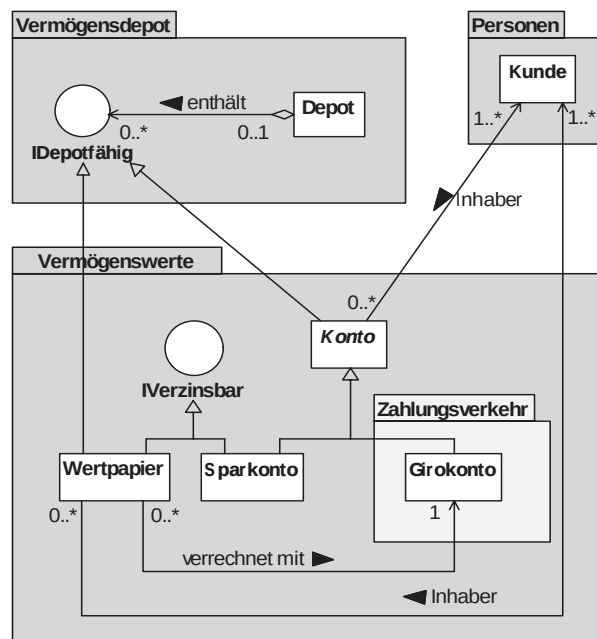


Abbildung 7.17: Namensräume – Depotverwaltung



Die Schnittstelle IDepot fähig und die Klasse Depot ordnen Sie dem Namensraum Vermögensdepot und die Klasse Kunde dem Namensraum Personen zu. Der Namensraum Vermögenswerte enthält das Interface IVerzinsbar sowie die Klassen Konto, Wertpapier und Sparkonto sowie den untergeordneten Namensraum Zahlungsverkehr, der die Klasse Girokonto aufnimmt.

In C# vereinbaren Sie einen Namensraum mit der namespace-Anweisung. Die folgenden Quelltexte zeigen die Vereinbarung und die Verwendung von Namensräumen für die Klassen der Depotverwaltung.

```
1: // Listing 7.27
2: /*...*/
3: using Vermögenswerte
4: namespace Vermögensdepot{
5:     public class Depot{/*...*/}
6: }
```

Zeile 4 zeigt die namespace-Anweisung, die alle im folgenden Block enthaltenen Elemente dem Namensraum Vermögensdepot zuordnet. Da im Vermögensdepot Konten und Wertpapiere verarbeitet werden, müssen Sie den Namensraum Vermögenswerte mit der using-Anweisung (Zeile 3) einbinden.

```
1: // Listing 7.28
2: namespace Vermögensdepot{
3:     public interface IDepotfähig
4:     { /*...*/ }
5: }
```

Schnittstellen werden als Typen in einem Namensraum ähnlich wie Klassen behandelt und müssen wie Klassen eingebunden werden.

```
1: // Listing 7.29
2: namespace Personen{
3:     public class Kunde
4:     { /*...*/ }
5: }
```

Da von der Klasse Kunde weder Vererbungs- noch Assoziationsbeziehungen ausgehen, wird hier kein Namensraum eingebunden.

```
1: // Listing 7.30
2: using System;
3: using System.Collections;
4: using Personen;
5: using Vermögensdepot;
6: using Vermögenswerte.Zahlungsverkehr;
7: namespace Vermögenswerte{
8:     public class Wertpapier : IDepotfähig,
9:                               IVerzinsbar
10:    { /*...*/ }
11: }
```

Die Klasse Wertpapier verweist auf alle fremden Namensräume und bindet sie dementsprechend ein.

```
1: // Listing 7.31
2: namespace Vermögenswerte{
3:     public interface IVerzinsbar
4:     { /*...*/ }
5: }
```

Das Interface IVerzinsbar benötigt keinen Zugriff auf fremde Namensräume.

```
1: // Listing 7.32
2: using System;
3: using System.Collections;
4: using Vermögensdepot;
5: using Personen;
6: namespace Vermögenswerte{
7:     public abstract class Konto : IDepotfähig
8:     { /*...*/ }
9: }
```

Die abstrakte Klasse `Konto` bindet für die Schnittstellenvererbung den Namensraum `Vermögensdepot` und für die Inhaber-Assoziation den Namensraum `Personen` ein.

```
1: // Listing 7.33
2: using System;
3: using Personen;
4: namespace Vermögenswerte{
5:     public class Sparkonto : Konto, IVerzinsbar
6:     { /*...*/ }
7: }
```

Da die in der Klasse `Konto` benötigte Schnittstelle `IVerzinsbar` demselben Namensraum `Vermögenswerte` wie die Klasse `Sparkonto` angehört, ist sie bereits bekannt. Das explizite Einbinden ist damit nicht notwendig.

```
1: // Listing 7.34
2: using System;
3: using Vermögenswerte;
4: using Personen;
5: namespace Vermögenswerte.Zahlungsverkehr{
6:     public class Girokonto : Konto
7:     { /*...*/ }
8: }
```

Da der Namensraum `Zahlungsverkehr` dem Namensraum `Vermögenswerte` untergeordnet ist, werden die Namensbestandteile entlang der Namenshierarchie durch einen Punkt getrennt nacheinander angegeben (Zeile 5). Alternativ zu dieser Syntax lassen sich Namensräume auch ineinander verschachteln. Beachten Sie, dass der übergeordnete Namensraum `Vermögenswerte`, wie alle fremden Namensräume auch, explizit mit einer `using`-Anweisung bekannt gemacht werden muss.

Die Startklasse bindet schließlich alle Namensräume ein, da Objekte aller Klassen verwendet werden.

```
1: // Listing 7.35
2: using System;
3: using Personen;
4: using Vermögensdepot;
5: using Vermögenswerte;
6: using Vermögenswerte.Zahlungsverkehr;
7: class Start
8: { /* ... */ }
```

Die Startklasse selber ist nicht in einen bestimmten Namensraum eingebunden und somit Bestandteil des globalen Namensraums.



Wenn Sie auf ein Element eines fremden Namensraums zugreifen, ohne den Namensraum einzubinden, müssen Sie den vollständig qualifizierten Namen angeben, der aus der Bezeichnung des Namensraums und dem durch einen Punkt getrennten Klassennamen besteht. Den vollständig qualifizierten Namen verwenden Sie auch zur Unterscheidung bei Namenskonflikten, wenn zwei Typen unterschiedlicher Namensräumen dieselbe Typbezeichnung besitzen.

Sichtbarkeit von Elementen

Neben den Namensräumen, die eine logische Struktur definieren, und den an dieser Stelle nicht besprochenen Assemblies, die eine physische Zuordnung definieren, können wir für alle Typen, Attribute und Operationen bestimmen, wer Zugriff auf das Element erhält.

Klassen, Schnittstellen, Aufzählungs-Typen

Jede Klasse gehört einem Namensraum an. Ist ein Namensraum nicht explizit vereinbart, ist die Klasse Mitglied des *global declarati-on space*. Somit ist jeder Typ Mitglied eines Namensraums. In einem Namensraum kennen sich alle Klassen gegenseitig. Das bedeutet, dass jede Klasse auf alle anderen Klassen zugreifen kann.

Begrenzen können Sie diesen Zugriff auf alle Typen einer Komponente (Assembly) mit dem Modifizierer `internal`. Durch Kennzeichnung mit `public` geben Sie den Zugriff auf die Klasse uneingeschränkt frei.

Geben Sie weder `internal` noch `public` an, so ist die Sichtbarkeit automatisch auf `internal`, also begrenzt auf die Komponente, eingestellt.

Attribute und Operationen

Falls Operationen oder Attribute, die Sie in einer Klasse verwenden wollen, in einem fremden Namensraum vereinbart sind, müssen Sie diesen zunächst über die `using`-Anweisung verfügbar machen.

Ist der Namensraum verfügbar, so kommt es zunächst darauf an, ob die neu zu erstellende Klasse in einer Vererbungsbeziehung zu der Klasse steht, die Sie erstellen. Liegt eine Vererbungsbeziehung vor, so können Sie auf alle Operationen und Attribute der Klasse zugreifen, die mit einem der Modifizierer `protected`, `protected internal` oder `public` versehen sind.

Stehen die Klassen nicht in einer Vererbungsbeziehung, sind aber Bestandteil einer Komponente, so können Sie nur die mit `internal` und `public` gekennzeichneten Attribute und Operationen der bestehenden Klasse verwenden.



Sollten Sie Komponenten (so genannte Assemblies) erstellen, kann die Sichtbarkeit mit dem Modifizierer `internal` auf die im Manifest festgelegten Mitgliederelemente (Klassen, Strukturen, Aufzählungs-Typen, Ereignisse) der Komponente beschränkt werden. Mit der Einschränkung der Sichtbarkeit auf `protected internal` erlauben Sie neben den Komponenten-Typen auch ererbenden Klassen außerhalb der Komponente den Zugriff.

Ist die neu zu erstellende Klasse nicht Komponentenmitglied der Klasse, auf die Sie zugreifen, und erbt die neue Klasse auch nicht deren Attribute und Operationen, so können Sie ausschließlich die public-Elemente der bestehenden Klasse erreichen.

Die folgende Abbildung fasst die Sichtbarkeiten noch einmal zusammen.

	Klasse-Intern	Unterkasse	Fremde Klasse	Unterkasse	Fremde Klasse
(private)	✓				
internal	✓	✓	✓		
protected	✓	✓		✓	
protected internal	✓	✓	✓	✓	
public	✓	✓	✓	✓	✓
	Assembly-Intern			Fremde Assembly	

Abbildung 7.18: Sichtbarkeit – Attribute und Operationen

Benutzen Sie die Abbildung als Entscheidungshilfe bei der Festlegung der Sichtbarkeit von Attributen und Operationen. Versuchen Sie immer den Zugriff so weit wie möglich einzuschränken, um ungewollte Manipulationen an Attributen und nur intern gebrauchte Operationen wirksam vor dem Zugriff anderer Klassen zu schützen. Berücksichtigen Sie dabei nicht nur bestehende Zugriffe auf Attribute und Operationen, sondern auch geplante zukünftige Zugriffe eigener und fremder Komponenten und Anwendungen mit ein.

- ✓ Wählen Sie zunächst im Kopf der Abbildung die Elemente aus, die einen Zugriff erhalten sollen. Gehen Sie dabei von links nach rechts vor und wählen Sie nur die notwendigen Typen (und nicht mehr) aus.
- ✓ Gehen Sie im zweiten Schritt für die ausgewählten Elemente die Modifizierer von oben nach unten durch, bis zum ersten Mal ein Modifizierer erreicht ist, der allen ausgewählten Typen den Zugriff erlaubt. Damit haben Sie den passenden Modifizierer gefunden.

Sollen zum Beispiel alle Unterklassen und zusätzlich alle Komponenten-Typen eine Operation aufrufen können, so ist `protected internal` der passende Modifizierer. Verwenden nur Operationen derselben Klasse ein Attribut, so kennzeichnen Sie es als `private`.

Depotverwaltung – Sichtbarkeiten

Nachdem Sie die Klassen der Depotverwaltung auf verschiedene Namensräume verteilt haben, vereinbaren wir im letzten Schritt die Sichtbarkeit von Klassen und Schnittstellen sowie deren Attributen und Operationen, so dass ein nicht gewollter Gebrauch möglichst ausgeschlossen wird.

Die folgende Abbildung zeigt die Sichtbarkeiten für die Attribute und Operationen. Dabei entspricht das Minus-Zeichen (-) dem Modifizierer `private`, die Raute (#) `protected` und das Plus-Zeichen (+) der globalen Sichtbarkeit `public`.

Die meisten Attribute sind nur für klasseninterne Operationen sichtbar, die meisten Operationen dagegen öffentlich verfügbar.

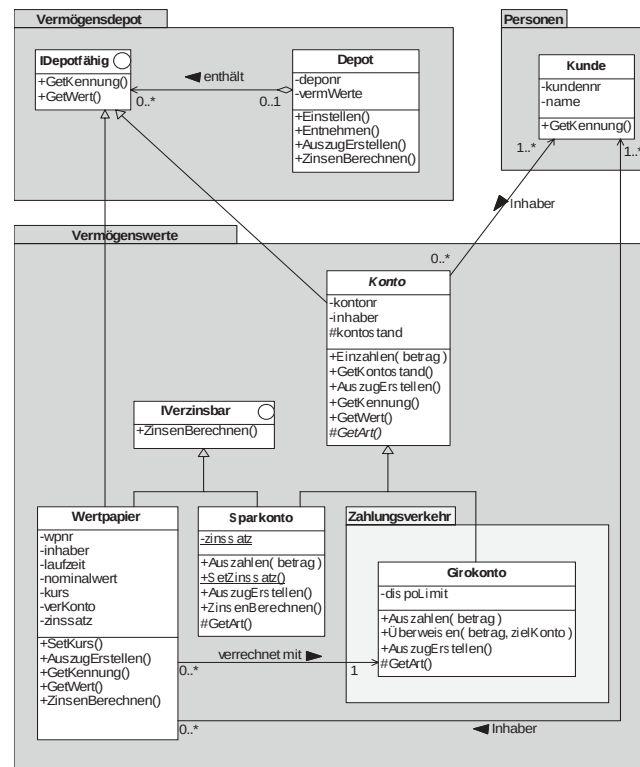


Abbildung 7.19: Namensräume – Strukturierung von Anwendungen

Abweichend davon ist das Attribut `kontostand` der Klasse **Konto** als `protected` definiert, da die Operation `Auszahlen(...)` der Unter-

klassen Sparkonto und Girokonto auf das Attribut zugreift, um den durch eine Auszahlung geänderten Kontostand zu speichern.

Schnittstellen-Operationen sind in C# grundsätzlich öffentlich und dürfen nicht mit einem Sichtbarkeits-Modifizierer gekennzeichnet werden.

Die Operation `GetArt()` ist nur innerhalb der Vererbungshierarchie sichtbar (`protected`), da die Operation `GetKennung()` sie ausschließlich intern verwendet. `Private` kann sie aber dennoch nicht vereinbart sein, da die abstrakte Operation der Klasse `Konto` als Platzhalter dient und für `Sparkonto`-Objekte und `Girokonto`-Objekte durch deren konkrete Definition ersetzt wird. Dabei sind ausschließlich sichtbare Operationen überschreibbar.

Da die Standard-Einstellung für Attribute und Operationen `private` ist, müssen Sie lediglich die drei Operationen `GetArt()` mit dem Modifizierer `protected` kennzeichnen. Damit besitzt die Depotverwaltung auch für eine komplexe Erweiterung eine tragfähige Grundstruktur.





TEIL

Praxis – Spezialaspekte und Werkzeuge der UML

Der vierte Teil beschäftigt sich mit praktischen Aspekten der UML. Hier erhalten Sie nützliche Checklisten, die Sie bei der objektorientierten Analyse mit der UML unterstützen. Ein Fallbeispiel zeigt die Anwendung der objektorientierten Modellierung auf Geschäftsprozesse.

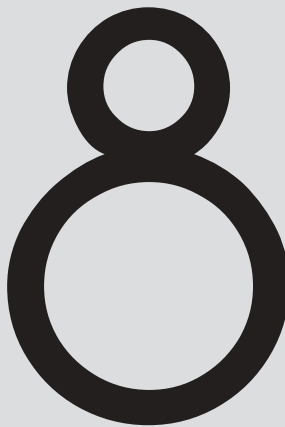
IV



KAPITEL

Checklisten für die objektorientierte Analyse

Die Checklisten in diesem Kapitel bieten Ihnen einen nützlichen Leitfaden zur objektorientierten Analyse. Sie enthalten Hinweise für die in der praktischen Projektarbeit oft schwierige Identifikation objektorientierter Strukturen.





Checklisten für die objektorientierte Analyse

Die Vorgehensweise im Rahmen des objektorientierten Software-Entwicklungsprozesses wurde im dritten Teil dieses Taschenbuchs bereits ausführlich anhand eines konkreten Beispiels vorgestellt. Dort haben Sie erfahren, wie die Diagramme der UML zur Modellierung eingesetzt und schließlich in ein objektorientiertes Programm umgesetzt werden.

In diesem Kapitel wollen wir Ihnen einige Checklisten an die Hand geben, die Ihnen die objektorientierte Modellierung mit der UML erleichtern sollen.

Problemstellung

Trotz eingehender Kenntnis der objektorientierten Strukturen und der Modelle, die Ihnen die UML zur Verfügung stellt, um einen Realitätsausschnitt abzubilden, werden Sie in der Praxis häufig mit einigen Problemen konfrontiert. Die zu bewältigenden Fragen, die sich Ihnen stellen werden, sind u.a.:

- ✓ *Wie grenze ich den für mich relevanten Problemausschnitt ab?*
Die Abgrenzung des Problemausschnitts hat weit reichende Konsequenzen für den Projekterfolg insgesamt – unabhängig davon, ob Sie Software entwickeln oder einen Geschäftsprozess modellieren wollen. Im Wesentlichen geht es darum, einen für Ihre individuellen Ziele geeigneten Blickwinkel auf die Problemstellung einzunehmen und die für Ihre Aufgabe relevanten Bereiche aus der Realität herauszufiltern.

- ✓ *Wie finde ich die Elemente, die in meinem Modell abgebildet werden sollen?*

Nachdem Sie Ihren Realitätsausschnitt abgegrenzt haben, stehen Sie vor dem Problem, darin geeignete Elemente zu finden, die Sie mit Hilfe der UML-Konzepte abbilden können. Wichtige Aspekte einer Problemstellung sind häufig nicht auf den ersten Blick erkennbar. Die Gefahr, ein unvollständiges Modell zu erhalten, ist entsprechend groß.

- ✓ *Nach welchen Kriterien entscheide ich, wie ein Element im Objektmodell repräsentiert wird?*

In der Praxis erweist es sich oft als schwierig, Objekte zu identifizieren, Klassen und Beziehungen zu bilden oder Abläufe zu beschreiben. Für die Zuordnung identifizierter Objekte zu Klassen und die Definition von Klassenstrukturen oder –beziehungen bieten sich meist zahlreiche Alternativen an, die eine Auswahl erschweren.

- ✓ *Wie erkenne ich, dass ich alle relevanten Aspekte berücksichtigt habe und mein Modell vollständig ist?*

Gerade die evolutionäre Vorgehensweise im Rahmen der objektorientierten Entwicklung führt dazu, dass sich das Ende dieses Prozesses kaum vorhersagen lässt. Zeit- und Kostenbudget können im Voraus nur annähernd geschätzt werden. Sie werden hier meist auf Erfahrungswerte zurückgreifen müssen, zumal Methoden zur Aufwandschätzung objektorientierter Projekte bisher erst ansatzweise zur Verfügung stehen. Wann ein Objektmodell vollständig ist, hängt von der konkreten Aufgabenstellung ab. Einige allgemeine Hinweise können jedoch dafür sorgen, dass wichtige Aspekte aus Modellsicht nicht unter den Tisch fallen.

Entwicklung objektorientierter Modelle

Die Vorgehensweise bei der objektorientierten Systementwicklung, die Sie in Teil III kennen gelernt haben, gliedert sich in die drei Phasen:

- ✓ Analyse (OOA)
- ✓ Design (OOD)
- ✓ Implementierung (Programmierung, OOP)

Der Schwerpunkt der UML liegt in der objektorientierten Analyse, auf die sich auch die eingangs genannten Hürden des Entwicklungsprozesses beziehen. Mit Hilfe einer groben Vorgehensweise sind die Schwierigkeiten der Analysephase leichter zu bewältigen.

Im Wesentlichen stehen Sie bei der Entwicklung eines objektorientierten Analysemodells vor den folgenden Aufgaben:

- ✓ Beschreibung eines Anwendungsfalls oder Geschäftsvorfalls
- ✓ Abgrenzung des relevanten Problemausschnitts
- ✓ Abgrenzung von Teilsystemen (Paketen) innerhalb der Problemstellung
- ✓ Definition von Schnittstellen zwischen Paketen
- ✓ Identifikation von Klassen und Attributen
- ✓ Ermittlung von Vererbungsstrukturen
- ✓ Identifikation von Assoziationen
- ✓ Identifikation von Interaktionen
- ✓ Identifikation von Ereignissen
- ✓ Definition von Zuständen und Zustandsübergängen
- ✓ Identifikation von Aktivitäten
- ✓ Beschreibung von Operationen

Diese umfangreichen Aufgaben sind in eine geeignete Abfolge zu bringen. Dabei ist es hilfreich, das System aus verschiedenen Blickwinkeln zu beleuchten.



In der Praxis hat sich in der Objektorientierung eine evolutionäre Vorgehensweise, bei der die einzelnen Phasen des Analyseprozesses wiederholt durchlaufen werden, als geeignet herausgestellt. So gelangt man durch die schrittweise Verfeinerung zunächst noch recht grober Strukturen zu einem ausgefeilten Modell, das den relevanten Problemausschnitt in geeigneter Form repräsentiert.

Zunächst betrachten Sie das System auf einem relativ hohen Abstraktionsniveau von außen. Dabei untersuchen Sie lediglich, was das System im Groben leisten soll bzw. welche Ergebnisse von einem Geschäftsprozess erwartet werden. Der Anwendungsfall oder Geschäftsvorfall bildet also quasi eine black box, in die Sie vorerst nicht hineinsehen können (bzw. wollen).

Zu dieser *groben Analyse* in der Außensicht gehören die Aufgaben

- ✓ Beschreibung des Anwendungsfalls,
- ✓ Abgrenzung des relevanten Problemausschnitts,
- ✓ Abgrenzung von Teilsystemen (Paketen) und
- ✓ Definition von Schnittstellen zwischen Paketen.

Nach der groben Systemanalyse in der Außensicht haben Sie einen Überblick darüber, *was* Ihr System leisten soll. Nun können Sie sich der Frage zuwenden, *wie* diese Leistung erbracht werden soll. Sie nehmen also einen neuen Blickwinkel ein und betrachten Ihr im Groben bereits abgegrenztes System nun aus der Innensicht.

Als grundsätzliche Vorgehensweise in der Innensicht bietet sich eine Anpassung an den objektorientierten Denkstil an. Dieser unterscheidet, wie Sie ja bereits wissen, zwischen einer *statischen* und einer *dynamischen* Struktur. Die verbliebenen Aufgaben für die Innensicht der objektorientierten Analyse lassen sich recht gut diesen beiden Kategorien zuordnen:

- ✓ Entwicklung des *statischen Modells*
 - ✓ Identifikation von Klassen, Attributen und Methoden
 - ✓ Beschreibung von Klassen und Attributen
 - ✓ Ermittlung von Vererbungsstrukturen
 - ✓ Identifikation von Assoziationen
- ✓ Entwicklung des *dynamischen Modells*
 - ✓ Identifikation von Interaktionen
 - ✓ Identifikation von Ereignissen
 - ✓ Definition von Zuständen und Zustandsübergängen
 - ✓ Identifikation von Aktivitäten
 - ✓ Beschreibung von Operationen

Damit haben wir die Aufgaben, die im Rahmen der objektorientierten Analyse anfallen, in drei Bereiche aufgeteilt:

- ✓ grobe Systemanalyse in der Außensicht
- ✓ statische Analyse in der Innensicht
- ✓ dynamische Analyse in der Innensicht

Zu Beginn eines Projektes ist es sinnvoll diese drei Aufgabenbereiche auch in der genannten Reihenfolge anzugehen, um sich zunächst einen genaueren Überblick über die Besonderheiten der jeweiligen Problemstellung zu verschaffen.

Je tiefer Sie jedoch in einen Anwendungsfall eintauchen, desto stärker treten die Interdependenzen zwischen Statik und Dynamik, Außen- und Innensicht sowie zwischen den identifizierten Elementen und Beziehungen zu Tage. Kleine Änderungen in der konkreten Gestaltung der Klassen mit ihren Attributen und Operationen sowie den Assoziationen wirken sich schnell auf Struktur und Verhalten des gesamten Systems aus. Die Abgrenzung der Teilsysteme

und die Definition ihrer Elemente haben Einfluss auf die Schnittstellen zwischen den Paketen.

Daraus folgt, dass sich die Entwicklung objektorientierter Modelle – wie auch die objektorientierte Systementwicklung insgesamt – nicht in eine lineare Abfolge pressen lässt.



Die einzig adäquate Vorgehensweise im Rahmen der objektorientierten Analyse ist evolutionär. Sowohl die drei Bereiche (grobe Systemanalyse, statische und dynamische Analyse) als auch die einzelnen Arbeitsschritte innerhalb dieser Bereiche werden in einem sich ständig wiederholenden Prozess durchlaufen.

So gelangt man durch die schrittweise Verfeinerung zunächst noch recht grober Strukturen nach und nach zu einem ausgefeilten Modell, das den relevanten Problemausschnitt in geeigneter Form repräsentiert.

Mehrfache Korrekturen und Anpassungen bereits definierter Modellbestandteile sind dabei absolut üblich. Mit jedem Zyklus des evolutionären Entwicklungsprozesses erhalten Sie neue Erkenntnisse über Strukturen und Zusammenhänge, die sich aufgrund der Interdependenzen in objektorientierten Modellen auf das gesamte System auswirken.

In den folgenden Abschnitten werden die genannten Aufgaben näher untersucht. Die Checklisten sollen Sie bei den einzelnen Arbeitsschritten der objektorientierten Analyse unterstützen.

Grobe Systemanalyse

Im Rahmen der *groben Systemanalyse* geht es vor allem darum, den in einem objektorientierten Modell abzubildenden Problembereich inhaltlich zu erfassen und gegenüber anderen, nicht relevanten Problembereichen abzugrenzen.

Die Hauptaufgabe besteht hier in der Entscheidung, welche Elemente einer Problemstellung in das Modell aufzunehmen sind. Meist ist jedoch die viel schwierigere und umstrittenere Frage, welche Bereiche ausgeklammert werden sollen.



Dabei geht es nicht nur um die Begrenzung der Komplexität des späteren Modells, die der Vielzahl wünschenswerter Eigenschaften und Bestandteile gegenübersteht. Viel wichtiger ist es in der Regel, den abzubildenden Problembereich dem meist sehr knappen Budget anzupassen.

Externe Restriktionen, die bereits in dieser ersten Phase der objektorientierten Analyse eine wesentliche Rolle spielen, sind:

- ✓ *Zeitliche Restriktionen:* Das Projekt muss in einem eng begrenzten Zeitrahmen realisiert werden. Gerade objektorientierte Projekte – unabhängig davon, ob es sich um die Softwareentwicklung oder die Geschäftsprozessmodellierung handelt – sind aufgrund der evolutionären Vorgehensweise bezüglich ihres Zeitbedarfs nur sehr schwer einzuschätzen.
- ✓ *Personelle Restriktionen:* Insbesondere die Fachabteilung kann neben dem normalen Tagesgeschäft nur wenige Mitarbeiter zeitlich begrenzt für andere Projekte abstellen. Das Projektmanagement wird daher stark von externen Einflüssen, die aus dem Tagesgeschäft resultieren, beeinflusst.
- ✓ *Finanzielle Restriktionen:* Budgets sind stets knapp bemessen. Die Problematik der Einschätzung des Projektaufwands gilt sowohl für den zeitlichen als auch für den finanziellen Ressourceneinsatz.
- ✓ *Sachmittel-Restriktionen:* Sachmittel wie Hardware, Software, Räume und Präsentationstechnik stehen oft nicht in ausreichender Anzahl zur Verfügung.

Die Konsequenz aus den genannten Restriktionen ist meist, dass sich ein Projekt zur objektorientierten Analyse nicht nur an der ab-

zubildenden Problemstellung und ihren Anforderungen orientiert, sondern das im Rahmen der knappen Ressourcen Mögliche versucht.

Diese Problematik ist bereits bei der Zielanalyse eines Projekts zu berücksichtigen, spielt aber auch hier in der groben Systemanalyse eine wichtige Rolle.

Im Einzelnen besteht die grobe Systemanalyse aus den folgenden Aktivitäten, die anschließend näher untersucht werden:

- ✓ Beschreibung des Anwendungsfalls
- ✓ Problemabgrenzung
- ✓ Definition und Abgrenzung von Teilsystemen
- ✓ Definition von Schnittstellen zwischen den Teilsystemen

Checkliste Anwendungsfall

Anwendungsfall-Diagramme wurden im dritten Kapitel vorgestellt. Sie beschreiben Anwendungsfälle oder Geschäftsvorfälle mit ihren Beziehungen untereinander und zu beteiligten Akteuren, Ereignissen oder Prozessen.



Bevor Sie beginnen, ein Use-Case-Diagramm zu einem Anwendungsfall zu entwerfen, sollten Sie diesen zunächst verbal beschreiben. Die schriftliche Fixierung eines Anwendungsfalls oder Geschäftsprozesses führt zu einem besseren Verständnis der Vorgänge und Zusammenhänge. Zudem ist die Dokumentation des Problembereichs ein wichtiges Hilfsmittel im weiteren Verlauf der Analyse, wenn es darum geht, einzelne Elemente und ihre Beziehungen zu identifizieren und zu beschreiben.

Die verbale Beschreibung eines Anwendungsfalls oder Geschäftsprozesses sollte folgende Bestandteile haben:

- ✓ Gewünschte Ergebnisse (Ziele) des betrachteten Prozesses
- ✓ Bedingungen oder Zusicherungen, die bei der Durchführung des Prozesses einzuhalten sind
- ✓ Regeln für den Prozessablauf
- ✓ Alternativen im Prozessablauf und Voraussetzungen, die einen alternativen Prozessverlauf auslösen



Vermeiden Sie eine zu detaillierte Beschreibung des Anwendungsfalls. Bedenken Sie, dass es hier lediglich um eine *grobe* Systemanalyse geht. Formulieren Sie zunächst nur das Wichtigste auf einer relativ hohen Abstraktionsstufe. Später wird diese erste grobe Abgrenzung im Rahmen des evolutionären Entwicklungsprozesses ständig verfeinert und konkretisiert.

Die folgende Checkliste hilft Ihnen bei der Beschreibung von Anwendungsfällen und beim Erstellen eines Use-Case-Diagramms. Sie enthält die erforderlichen Aktivitäten sowie die im Rahmen der einzelnen Aktivitäten zu klärenden Fragestellungen. Durch die Beantwortung der Fragen erhalten Sie einen ausreichend genauen Überblick über Ihren Anwendungsfall.

Aktivität	Zu klärende Fragen
Ermittlung von Akteuren	Welcher Akteur löst den Anwendungsfall aus? Welche Personen sind beteiligt? Wer ist verantwortlich für die Planung des Prozesses? Gibt es außer den beteiligten Personen noch weitere Akteure (z.B. Software-Systeme)? Wer ist verantwortlich für die Durchführung des Prozesses? Wer ist verantwortlich für die Kontrolle des Prozesses? Wer benutzt das System bzw. den Prozess? Wer benötigt Daten oder Ergebnisse aus dem Prozess?

Aktivität	Zu klärende Fragen
Ermittlung von Aufgaben	Welche Ziele sind mit dem Anwendungsfall verbunden? Welche Ergebnisse soll das System liefern? Welche Arbeitsschritte sind dazu erforderlich? Welche Ziele sind mit jedem dieser Arbeitsschritte verbunden?
Ermittlung von Schnittstellen	Zu welchen anderen Anwendungsfällen bestehen Beziehungen? Benötigt das System Vorleistungen aus einem anderen System? Gibt das System Daten oder Ergebnisse an andere Systeme weiter? Existieren Vorgaben für die Schnittstellen zu bestimmten Akteuren (z.B. Benutzungsoberfläche für menschliche Akteure)?
Berücksichtigung von Restriktionen	Welche Abhängigkeiten bestehen zwischen den durchzuführenden Aufgaben? Welche Vor- und Nachbedingungen sind bei den einzelnen Aufgaben zu beachten? Welche Aufgaben unterliegen organisatorischen, technischen, gesetzlichen oder natürlichen Beschränkungen?
Ermittlung von Ereignissen	Welche Ereignisse treten standardmäßig in dem Anwendungsfall auf? Welche Auswirkungen haben diese Ereignisse? Welche außergewöhnlichen Ereignisse können auftreten? Welche Auswirkungen haben diese außergewöhnlichen Ereignisse? Sollen für den Fall solcher Ereignisse Alternativen im Prozessablauf geplant werden?

Aktivität	Zu klärende Fragen
Strukturierung	Welche Standardaufgaben und welche Spezialfälle gibt es? Welche Teilprozesse können gebildet werden, um ein komplexes System zu vereinfachen? Welche Gemeinsamkeiten gibt es zwischen den Prozessen? Wie können die anfallenden Aufgaben alternativ gruppiert werden? Wie wirken sich alternative Strukturen auf die Ergebnisse und den Zielerreichungsgrad aus?
Erweiterung	Welche zukünftigen Erweiterungen sind bereits absehbar? Welche Erweiterungsmöglichkeiten bietet das System? Ist das System hinreichend flexibel?

Tabelle 8.1: Checkliste Anwendungsfall

Als Ergebnis der Bearbeitung dieser Checkliste erstellen Sie

- ✓ ein Use-Case-Diagramm und
- ✓ eine verbale Beschreibung Ihres Anwendungsfalls bzw. Geschäftsprozesses.



Beachten Sie bei der Beschreibung von Anwendungsfällen, dass Sie sich hier in der Außensicht befinden. Das System ist für Sie (zunächst) noch eine black box. Die Versuchung ist groß, hier schnell zu sehr ins Detail zu gehen.

Die Beantwortung dieser Fragen ermöglicht eine (grobe) Beschreibung der Inhalte, der beteiligten Akteure und der Rahmenbedingungen eines Anwendungsfalls. Außerdem lassen sich so Interde-

pendenzen mit anderen Anwendungsfällen identifizieren, die in die Betrachtung mit einbezogen werden müssen.

Checkliste Problemabgrenzung

Die Abgrenzung des relevanten Problemausschnitts ist eng mit der Beschreibung des Anwendungsfalls verzahnt. Das Ausloten der Systemgrenzen ist dabei oft mit großen Schwierigkeiten verbunden. Die Aufgabe der Projektbeteiligten besteht darin, eine geeignete Abgrenzung zu finden, die einerseits alle notwendigen Elemente berücksichtigt, andererseits aber auch Überflüssiges und Unnötiges ausschließt.

Die verbale Beschreibung des Anwendungsfalls kann hier eine große Hilfe sein. Dort haben Sie bereits wichtige Elemente Ihres Anwendungsfalls identifiziert und erste Beziehungen zwischen Elementen und anderen Systemen ermittelt.

Die Bedeutung dieser Phase des Modellierungsprozesses für den Projekterfolg ist nicht zu unterschätzen. Zu weite oder zu enge Systemgrenzen können die Qualität der zu entwickelnden Software oder des zu organisierenden Geschäftsprozesses negativ beeinflussen. Sie sollten diese Phase daher mit besonderer Sorgfalt angehen.

Mehr noch als bei der Modellierung von Geschäftsprozessen besteht diese Problem in der Softwareentwicklung. Die Haltung der Fachabteilung, die das zu entwickelnde System später einsetzen soll, geht häufig dahin, wünschenswerte inhaltliche Funktionen als unbedingt notwendig darzustellen. Umgekehrt neigen Softwareentwickler dazu, ihren »Spieltrieb« auszuleben und technisch aufwändige oder anspruchsvolle Realisierungen zu favorisieren, die in dieser Form vielleicht gar nicht gebraucht werden.



Die Zusammenarbeit von Modellierern oder Softwareentwicklern und Fachabteilung ist nicht nur von rein professionellen Zielen geprägt. Stets spielen auch gruppenspezifische und individuelle persönliche Ziele der Beteiligten eine Rolle. Die Anfor-

derungen der eigentlichen Aufgabe mit diesen widerstrebenden Interessen zu vereinbaren, ist eine nicht zu unterschätzende Herausforderung für das Projektmanagement.

Um dem tatsächlich relevanten Problembereich näher zu kommen, sollten Sie die folgenden Aktivitäten durchführen. Die Beantwortung der in der Checkliste aufgeführten Fragen unterstützt Sie dabei.

Aktivität	Zu klärende Fragen
Zielsetzung	Welche Ziele werden mit dem Geschäftsprozess oder Anwendungsfall verfolgt?
Relevanzprüfung	Ist das betrachtete Element für die Ziele und Aufgaben des Anwendungsfalls oder Geschäftsvorfalles von Bedeutung? Welche Auswirkungen hat der Verzicht auf die Integration des betrachteten Elements für die Erfüllung der Aufgabe oder des Ziels?
Ermittlung von Alternativen	Welche Rollen (Stellen) sind an der Aufgabe beteiligt? Welche anderen Rollen sind von der Ausführung der Aufgaben betroffen? Kann die Erfüllung von Aufgaben durch eine Umgestaltung in aufbau- oder ablauforganisatorischer Hinsicht optimiert werden? Können komplexe Wechselwirkungen durch die Einbeziehung weiterer Systeme (Systemvergrößerung) internalisiert werden? Können komplexe Wechselwirkungen durch eine reduzierte Systemsicht (Systemverkleinerung) vernachlässigt werden?

Tabelle 8.2: Checkliste Problemabgrenzung

Bei der Bearbeitung der zugehörigen Checklisten wird Ihnen die Verzahnung der beiden Aufgaben im Rahmen der Modellierung auffallen. Da die Problemabgrenzung und die Beschreibung des Anwendungsfalls eng miteinander verbunden sind, sollten sie in einem evolutionären Prozess gemeinsam durchgeführt werden.

Häufig treten erst bei der detaillierten Beschäftigung mit den Klassen, den Objekten und ihren Beziehungen zusätzliche Aspekte zu Tage, die im Rahmen der groben Systemanalyse zunächst nicht berücksichtigt wurden. Wahrscheinlich werden Sie daher später, bei der Entwicklung des statischen und des dynamischen Modells, weitere Anpassungen vornehmen müssen.

Checkliste Teilsysteme

Die Komplexität von Anwendungsfällen lässt sich durch die Bildung von Teilsystemen reduzieren. Das Gesamtsystem wird dabei in überschaubare Einheiten, so genannte *Pakete*, aufgeteilt, die einen in sich geschlossenen Teilbereich der Problemstellung abbilden und miteinander durch eindeutig definierte Schnittstellen verbunden sind.

Nach welchen Kriterien Sie die Paketbildung vornehmen, hängt von Ihrem konkreten Anwendungsfall ab. Einige mögliche Gliederungskriterien enthält die folgende Tabelle.

Gliederungskriterium	Beschreibung
Inhaltliche Gliederung	Bei der inhaltlichen Gliederung wird nach gemeinsamen Eigenschaften von Aufgaben und Aktivitäten gesucht.
Prozessorientierte Gliederung	Die prozessorientierte Gliederung richtet sich nach der Reihenfolge der anfallenden Aufgaben in einem Anwendungsfall oder Geschäftsprozess. Aktivitäten, die z.B. in einem abgegrenzten Produktionsschritt stattfinden, werden zu Paketen zusammengefasst.

Gliederungskriterium	Beschreibung
Lokale Gliederung	Bei der lokalen Gliederung bilden Sie Teilsysteme mit Aufgaben, die an demselben Ort (z.B. in der derselben Produktionsstätte oder auch in derselben Datenbank) anfallen.
Funktionale Gliederung	Die funktionale Gliederung unterscheidet Pakete nach der Ausführung der enthaltenen Aufgaben durch spezialisierte Rollen. Dabei kann es sich beispielsweise um eine bestimmte Abteilung handeln, in der funktional gleiche Aufgaben anfallen bzw. erledigt werden.

Tabelle 8.3: Gliederungskriterien für die Paketbildung

Insbesondere bei umfangreichen Aufgaben bietet es sich an, durch Paketbildung kleinere überschaubare Einheiten zu schaffen.

Problematisch ist dabei die Wahl der richtigen Größe der Pakete. Sie sollten weder zu viele noch zu wenige Teilaufgaben und Elemente umfassen.



Zu kleine Pakete konterkarieren den positiven Effekt der Komplexitätsreduktion, da die Zahl der Interdependenzen zwischen den Paketen in der Regel sehr hoch sein dürfte. Zu große Pakete haben den Nachteil, in sich zu komplex zu sein – der mit der Paketbildung beabsichtigte Effekt tritt gar nicht oder nur in geringem Maße ein.

Aktivität	Zu klärende Fragen
Ermittlung von Gliederungskriterien	Ist das betrachtete System in der vorliegenden Form überhaupt teilbar? Ergeben sich bereits aus der Aufbau- oder Ablaufstruktur des Geschäftsprozesses oder

Aktivität	Zu klärende Fragen
	Anwendungsfalls Kriterien für die Paketbildung? Erlaubt die Gesamtaufgabe eine inhaltliche, prozessorientierte, lokale oder funktionale Gliederung?
Definition von Paketen	Handelt es sich bei dem betrachteten Element um ein isolierbares Subsystem? Bildet das betrachtete Paket einen Themenbereich ab, der weitgehend für sich alleine gesehen werden kann? Liegen die Vererbungsstrukturen der beteiligten Klassen weitgehend innerhalb des Pakets? Können umfangreiche Subsysteme in weitere kleinere Pakete zerlegt werden? Können durch die Umgestaltung von Aufgaben, Prozessen und Zuständigkeiten Pakete gebildet werden, ohne die Ziele zu gefährden?
Berücksichtigung von Interdependenzen	Welche Beziehungen und Abhängigkeiten besitzt das betrachtete Element zu anderen Elementen innerhalb des Systems? Sind von der Integration des betrachteten Elements in ein (Teil-) System weitere Elemente in anderen (Teil-) Systemen betroffen, die ihrerseits wiederum Abhängigkeiten aufweisen?

Tabelle 8.4: Checkliste Teilsysteme

Die Paketbildung steht in engem Zusammenhang mit der Abgrenzung des relevanten Problemausschnitts. Zugleich hat diese Phase im Rahmen der objektorientierten Analyse bereits weitgehende Auswirkungen auf das statische Objektmodell.

Da jedes Paket eine in sich geschlossene Einheit darstellt, die über Schnittstellen mit anderen Systemen kommuniziert, hängt die Definition der Klassen und ihrer Beziehungsstrukturen stark von den identifizierten Paketen ab.

Sie können diese Wechselwirkungen der Paketbildung mit der Problemabgrenzung einerseits und der Klassenbildung andererseits an den Fragen in der Checkliste erkennen. Diese greifen Ergebnisse der Problemabgrenzung auf und erfordern gleichzeitig bereits erste Überlegungen zur Klassenstruktur.

Auch hier zeigt sich somit wieder einmal das Ineinandergreifen der einzelnen Schritte im Modellierungsprozess. Eine streng sequentielle Vorgehensweise ist dabei zwangsläufig zum Scheitern verurteilt.

Bei der Paketbildung bieten sich Ihnen grundsätzlich zwei alternative Vorgehensweisen:

- ✓ Bei einer *Top-down*-Vorgehensweise zerlegen Sie zunächst das Gesamtsystem in Teilsysteme. Anschließend versuchen Sie, größere Pakete in weitere Subsysteme zu gliedern, bis Sie ein für Ihre Bedürfnisse angemessenes Komplexitätsniveau erreicht haben.
- ✓ Im Rahmen der *Bottom-up*-Vorgehensweise kehren Sie das Verfahren um. Hier erstellen Sie zuerst Ihre Klassenstruktur und suchen dann nach Gemeinsamkeiten der identifizierten Klassen. Klassen mit gleichen Eigenschaften oder Aufgaben fassen Sie zu Paketen zusammen. Solange es im Rahmen der Projektaufgabe sinnvoll ist, generalisieren Sie diese Pakete zu weiteren Oberpaketen.

Checkliste Schnittstellen

Nachdem Sie das Gesamtsystem in Pakete zerlegt haben, definieren Sie Schnittstellen, um die Kommunikation zwischen den einzelnen Teilsystemen sicherzustellen. Über die Schnittstellen tauschen die

Pakete Nachrichten aus, fordern Dienste an oder leiten Zwischenergebnisse weiter.

Verbindungen können sowohl direkt zwischen zwei Paketen als auch zwischen Klassen unterschiedlicher Pakete bestehen. Bereits im Schritt zuvor ist bei der Paketbildung darauf zu achten, dass Abhängigkeiten zwischen Paketen bzw. ihren Klassen möglichst vermieden werden. So bewahren Sie ein Höchstmaß an Modularität in Ihrem System.



Abhängigkeitsbeziehungen zwischen Elementen verschiedener Pakete erschweren den modularen Aufbau eines Modells. Die Wiederverwendung einzelner Module kann dann zu so vielfältigen gegenseitigen Abhängigkeiten führen, dass Änderungen in einem Paket mit erheblichem Aufwand auch in anderen Paketen verbunden sind.

Die Schnittstellen selbst müssen in ihrer Struktur sämtliche bestehenden Assoziationen zwischen Elementen berücksichtigen und sollten möglichst auch auf zukünftige Anpassungen des Modells vorbereitet sein. Dies gilt insbesondere dann, wenn die Wiederverwendung einzelner Pakete vorgesehen ist. Ein ausreichendes Maß an Flexibilität darf auf der anderen Seite aber auch nicht zu Lasten der Effizienz, Stabilität und Übersichtlichkeit eines Pakets gehen.

Die folgende Checkliste unterstützt Sie bei der Ermittlung und Gestaltung der erforderlichen Schnittstellen.

Aktivität	Zu klärende Fragen
Ermittlung von Beziehungen	Welche Kommunikationsbeziehungen bestehen zwischen den Paketen? Welche Kommunikationsbeziehungen bestehen zwischen den Klassen verschiedener Pakete? Welcher Art sind die Daten, die über die identifizierten Schnittstellen ausgetauscht werden?

Aktivität	Zu klärende Fragen
Ermittlung von Abhängigkeiten	Welche Abhängigkeiten bestehen zwischen den Paketen? Welche Abhängigkeiten bestehen zwischen den Klassen verschiedener Pakete?
Reorganisation von Paketen	Erscheint aufgrund der ermittelten Kommunikationsbeziehungen und Abhängigkeiten eine Umgestaltung der Pakete sinnvoll?

Tabelle 8.5: Checkliste Schnittstellen zwischen Paketen

Einige Fragen dieser Checkliste können erst nach der vollständigen Definition der Pakete und ihrer Elemente beantwortet werden. Andere führen möglicherweise zu der Erkenntnis, dass die ursprüngliche Abgrenzung der Pakete einer Überarbeitung bedarf.

Statische Analyse

Nachdem Sie das System nun in groben Zügen beschrieben und in einzelne Pakete mit klar definierten Schnittstellen zerlegt haben, wechseln Sie nun die Sichtweise.

Bisher haben Sie das System aus der Außensicht weitgehend als black box betrachtet. Zur Modellierung der statischen Struktur analysieren Sie Ihre Problemstellung nun detaillierter, indem Sie die beteiligten Klassen mit ihren Attributen und Operationen identifizieren.

Hierzu gehört auch die Bildung von Klassenhierarchien und die Ermittlung von Assoziationen sowohl zwischen den Klassen eines Pakets als auch paketübergreifend.

Im Einzelnen besteht die Entwicklung des statischen Modells somit aus den folgenden Aktivitäten:

- ✓ Bei der *Identifikation von Klassen und ihren Eigenschaften* werden in einem ersten Schritt zunächst alle Elemente gesammelt, die für die Problemstellung relevant sind.
- ✓ Die *Klassenbeschreibung* besteht aus einer verbalen Beschreibung in Form einer Dokumentation.
- ✓ Mit der *Klassenstruktur* legen Sie Vererbungsbeziehungen zwischen den zuvor identifizierten und beschriebenen Klassen fest.
- ✓ Schließlich sind die weiter gehenden *Assoziationen* zwischen den ermittelten Klassen zu definieren.



Bitte beachten Sie, dass es sich auch hier wiederum nicht um einen streng linearen Ablauf von Aktivitäten handelt. Zwar ist es in den meisten Fällen durchaus sinnvoll, zuerst Klassen und Eigenschaften zu identifizieren, bevor Sie mit den übrigen Aktivitäten fortfahren. Um eine zyklische Anpassung Ihres Modells werden Sie jedoch insbesondere bei größeren Gestaltungsprojekten nicht herumkommen.

Die Checklisten in den folgenden Abschnitten helfen Ihnen bei den genannten Aktivitäten zur Entwicklung des statischen Modells.

Checkliste Klassenbildung

Die Identifikation von Klassen, Attributen und Operationen ist zu Beginn des Modellierungsprozesses nicht allzu schwer. Im Gegenteil werden Sie häufig sehr schnell eine Vielzahl von Elementen ermitteln. Die Schwierigkeit besteht dann eher darin, die gefundenen Elemente einander sinnvoll zuzuordnen.

Als Informationsquellen können Sie dabei u.a. die folgenden heranziehen:

- ✓ *Ergebnisse der groben Systemanalyse*
Einige Elemente des statischen Modells haben Sie möglicherweise bereits im Rahmen der groben Systemanalyse ermittelt. In der Regel kristallisieren sich bereits bei der Paketbildung einzelne Klassen und Eigenschaften heraus.
- ✓ *Formulare, Listen, Verträge und andere Dokumente*
Viele Informationen über Ihre Problemdomäne werden in Form schriftlicher Dokumente vorliegen. Die Durchsicht solcher Texte fördert eine Vielzahl von Elementen zu Tage. Auch hier besteht das Hauptproblem weniger darin, überhaupt »Kandidaten« für Modellelemente zu finden, als vielmehr in deren Auswahl und Anordnung.
- ✓ *Dokumentationen von Geschäftsprozessen, Projekten und Software*
Geschäftsprozesse, Projekte und selbst entwickelte Software sind in der Regel dokumentiert. Ebenso wie Formulare und Listen bieten diese Dokumente einen ersten Überblick über die Problemstellung und helfen bei der Identifikation von Klassen und Eigenschaften.
- ✓ *Datenmodelle (Datenbanken), Bildschirmmasken etc.*
Datenmodelle geben einen tiefen Einblick in die Datenstrukturen ihres Anwendungsfalls. Gerade für die Klassenbildung lassen sich hier wertvolle Informationen finden, unabhängig davon, ob es sich um klassische oder objektorientierte Datenmodelle handelt. Auch Bildschirmmasken von Software enthalten nützliche Informationen über die Struktur und die Zusammenhänge der zu verarbeitenden Daten. Oft lässt sich aus einer einzelnen Maske sogar eine komplette Klasse ableiten. Die in der Bildschirmmaske einzugebenden Daten gehören meist thematisch zusammen und entsprechen dann den Attributen der zu bildenden Klasse.
- ✓ *Organigramme*
Organigramme enthalten Informationen über die Anordnung von Stellen. Auch Kommunikationswege und Datenflüsse lassen sich eventuell daraus entnehmen.

✓ *Stellenbeschreibungen*

Gerade für die Modellierung von Geschäftsprozessen sind Stellenbeschreibungen eine wichtige Informationsquelle. Sie beschreiben die konkreten Aktivitäten, die mit einer Stelle verbunden sind. Insbesondere bei der Suche nach Operationen von Objekten sollten Sie hier fündig werden.

✓ *Interviews mit Akteuren*

Interviews sind ein wichtiges Hilfsmittel, um Informationen zu sammeln, die in schriftlichen Dokumenten nicht oder nicht ausführlich genug vorliegen. Viele Zusammenhänge sind nicht formal erfassbar und nur direkt von den beteiligten Akteuren zu erfahren. Dabei handelt es sich insbesondere um Vorgänge oder Verhaltensweisen, die eher »aus dem Bauch heraus« oder »auf dem kleinen Dienstweg« erfolgen. Solche Elemente zu erfassen ist oft nicht einfach und stößt unter Umständen sogar auf aktiven Widerstand der Befragten. Durch die Kombination geeigneter Interviewtechniken lassen sich derartige Schwierigkeiten jedoch oftmals bewältigen.

✓ *Dokumentation von Modellen vergleichbarer Problemdomänen (andere Abteilungen, Fachliteratur, Erfahrungsberichte)*

Vielleicht haben Sie ähnliche Problemdomänen bereits früher schon einmal modelliert. Modelle zu verbreiteten Themen finden Sie auch in der Literatur oder im Internet. Diese werden Sie zwar nicht einfach übernehmen können – zumindest liefern sie Ihnen aber hilfreiche Ansätze und Anregungen für Ihre eigenen Modelle.



Bei der Auswertung dieser Informationsquellen deuten die gefundenen Substantive (z.B. in Textdokumenten) häufig auf potenzielle Klassen hin. Verben bieten sich als Bezeichnungen für Operationen oder Klassen an. Hinter Gruppen beschreibender Substantive verbergen sich möglicherweise Attribute (z.B. Name, Adresse und Telefonnummer auf einem Formular als Attribute einer Klasse Person).

Die genannten Informationsquellen sollten zunächst in einer Art Brainstorming zur Sammlung potenzieller Kandidaten für Klassen herangezogen werden. Planen Sie auch hier wieder ein zyklisches Vorgehen ein und verfeinern Sie so Ihre Suche nach Klassen und Eigenschaften immer weiter.



Verwenden Sie aussagekräftige Klassennamen, um die Übersichtlichkeit und Verständlichkeit des späteren statischen Objektmodells von Beginn an zu fördern. Es könnte Ihnen sonst passieren, dass Sie nach einiger Zeit nicht mehr wissen, welchen Themenbereich eine Klasse ursprünglich beschreiben sollte.

Die folgende Checkliste hilft Ihnen bei der Identifikation von Klassen, Attributen und Operationen.

Aktivität	Zu klärende Fragen
Identifikation von Klassen	Welche Klassen ergeben sich aus der Beschreibung der Geschäftsprozesse bzw. Anwendungsfälle? Welche Akteure (Personen, Abteilungen, Organisationen) sind an dem Geschäftsvorfall aktiv oder passiv beteiligt? Welche Rollen übernehmen Personen in dem Geschäftsvorfall? Welche Gegenstände werden für die zu beschreibende Aufgabe benötigt? Welche Hardware und Software wird eingesetzt? Welche Informationen werden von den Akteuren verarbeitet? An welchen Orten werden die mit den Aufgaben verbundenen Aktivitäten durchgeführt? Können Aktionen zu Funktionsbereichen zusammengefasst werden? Lassen sich strukturelle Ähnlichkeiten zwischen einzelnen Elementen feststellen?

Aktivität	Zu klärende Fragen
	Welche (internen und externen) Ereignisse treten in dem Geschäftsvorfall auf?
Identifikation von Attributen	Wodurch ist die betrachtete Klasse eindeutig gekennzeichnet? Welche Daten beschreiben die betrachtete Klasse? Ergeben sich Attribute aus der Vererbungsstruktur? Was unterscheidet die betrachtete von den übrigen identifizierten Klassen? Welche Zustände können die Exemplare einer Klasse annehmen? Ist das betrachtete Attribut Gegenstand von Operationen? Handelt es sich um ein Klassenattribut oder besitzt jedes Objekt individuelle Werte für dieses Attribut? Ist das Attribut so komplex, dass sich eine nähere Beschreibung in einer eigenen Klasse anbietet? Verlangen die zugreifenden Operationen die Zusicherung eines bestimmten Ergebnistyps? Welche Informationen muss die betrachtete Klasse für andere Klassen zur Verfügung stellen?
Identifikation von Operationen	Welche (Teil-) Prozesse wurden in der Beschreibung der Geschäftsprozesse bzw. Anwendungsfälle ermittelt? Welche Verhaltensweisen zeigen die beteiligten Akteure zur Bewältigung ihrer Aufgaben? Welche Aktionen werden von den beteiligten Akteuren durchgeführt? Welche Prozesse laufen in der eingesetzten Hardware und Software ab?

Aktivität	Zu klärende Fragen
	Wer gibt Informationen weiter? Wer erhält Informationen? Über welche Kanäle werden Informationen weitergegeben? Welche anderen Klassen (bzw. Objekte) benötigen Informationen? In welcher Form benötigen andere Klassen oder Objekte diese Informationen? Durch welche Ereignisse oder Aktionen ändern die Exemplare einer Klasse ihren Zustand?
Validierung	Ist der Klassenname aussagekräftig? Ist der Klassenname eindeutig (innerhalb des Pakets)? Gibt es außer dem Klassennamen Eigenschaften, welche die betrachtete Klasse von anderen Klassen signifikant unterscheidet? Enthält die Klassenbeschreibung alle erforderlichen Attribute und Operationen? Sind die geerbten Attribute und Operationen in der geerbten Form (Datentyp, Sichtbarkeit) sinnvoll? Ist die betrachtete Klasse überschaubar und übersichtlich (Anzahl und Bezeichnung der Attribute und Operationen)? Ist die betrachtete Klasse zwingend erforderlich oder kann sie auch in anderen Elementen aufgehen (z.B. als Attribut einer anderen Klasse)? Sind die Informationen der Klassenbeschreibung für den Anwender relevant? Gehört die betrachtete Klasse eindeutig zu diesem Paket oder ist unter Umständen die Zuordnung zu einem anderen Teilsystem sinnvoller? Welche Auswirkungen hat das Fehlen der betrachteten Klasse (bzw. des Attributs oder

Aktivität	Zu klärende Fragen
	der Operation) auf die korrekte Abbildung des Problembereichs? Welche Auswirkungen hat das Fehlen der betrachteten Klasse auf das Objektmodell?

Tabelle 8.6: Checkliste Klassenbildung



Achten Sie bei der Namensgebung auf die Konventionen der UML!

Nach der Identifikation der Klassen und ihrer Eigenschaften werden diese in einer verbalen Klassenbeschreibung dokumentiert.

Checkliste Klassenbeschreibung

Ziel der Dokumentation ist die eindeutige Formulierung der Aufgaben, Inhalte und Strukturen einer Klasse. Sie beugt Missverständnissen zwischen Modellierern und Programmierern vor, da bei der späteren Implementierung die ursprüngliche Bedeutung einer Klasse stets nachvollzogen werden kann.

Ein weiterer Vorteil der schriftlichen Dokumentation liegt darin, dass Sie sich bei der Formulierung der verbalen Beschreibung noch einmal intensiv und im Detail mit Ihrem Modell auseinander setzen müssen. Dabei werden Sie häufig Fehler, Schwächen oder notwendige Erweiterungen des Modells aufdecken, die bis dahin nicht aufgefallen sind.

Von großer Bedeutung ist die schriftliche Dokumentation schließlich für die Wartbarkeit des Systems. Sie dient nach Inbetriebnahme des Systems als wichtiges Hilfsmittel für Änderungen und Ergänzungen des Objektmodells.



Ist Ihr System erst einmal einige Zeit in Betrieb, werden die Modellierer und Entwickler Einzelheiten des Modells bzw. der Implementierung recht schnell vergessen haben. Oft sind sie sogar überhaupt nicht mehr erreichbar. Wenn Sie dann ein solches, Ihnen völlig fremdes System ändern oder ergänzen müssen, werden Sie für jede schriftliche Dokumentation des Objektmodells und des Programmcodes dankbar sein.

Die folgenden Elemente sind Bestandteil einer Klassenbeschreibung.

Bestandteile	Inhalte
Klassenname	Aussagekräftiger Klassenname Auflistung (ggf. auch nähere Beschreibung) der Unterklassen
Paketname	Benennung der Paketzugehörigkeit Beschreibung der Aufgabe der Klasse innerhalb des Pakets Sichtbarkeit der Klasse innerhalb und außerhalb des Pakets
Vollständige Vererbungsstruktur zur Herkunft der	Sämtliche Oberklassen, aus denen die Klasse abgeleitet wird Klasse
Attribute	Name Datentyp Sichtbarkeit Aufteilung in geerbte und selbst definierte Attribute
Operationen	Name Sichtbarkeit Parameterliste mit Datentypen Rückgabewerte mit Datentypen Datentyp der Operation Aufteilung in geerbte und selbst definierte Operationen

Bestandteile	Inhalte
Verschiedenes	Beispiele für Exemplare der Klasse Beispiele für die Benutzung der Klasse Hinweise auf Beziehungen zu anderen Klassen

Tabelle 8.7: Checkliste Klassenbeschreibung

Hypertext-Dokumente sind für die Klassen-Dokumentation besonders geeignet. Verweise auf Ober- oder Unterklassen, geerbte bzw. vererbte Attribute und Operationen sowie Hinweise auf Beziehungen zu anderen Klassen lassen sich so sehr einfach darstellen. Zudem erleichtern sie dem Benutzer die Navigation innerhalb des Dokuments.

Zusammen mit dem Paketnamen ermöglicht der Klassenname die eindeutige Identifikation einer Klasse in einem Objektmodell, da jeder Klassenname laut UML-Konvention in einem Paket nur ein einziges Mal vergeben wird. Die Angabe der Unterklassen liefert wichtige Informationen über die Konsequenzen, die eine Änderung der Klasseneigenschaften nach sich zieht.

Über die bloße Benennung der Paketzugehörigkeit hinaus sollte auch kurz die Aufgabe oder der Zweck der Klasse innerhalb des Pakets bzw. im Rahmen des Gesamtsystems erläutert werden. Meist ist der Klassenname allein nicht aussagekräftig genug, auch wenn Sie einen sinnvollen Namen gewählt haben.

Die vollständige Vererbungsstruktur ordnet die beschriebene Klasse in eine Klassenstruktur ein und erlaubt die Feststellung des Ursprungs geerbter Attribute und Operationen.

Anhand der Attribute und ihrer Datentypen sind Aussagen über potenzielle Zustände von Objekten möglich. Die Sichtbarkeit bestimmt die Verwendung von Attributwerten bzw. deren Zugriff aus

anderen Klassen. Aus der Aufteilung in geerbte und selbst definierte Attribute lässt sich deren Herkunft ersehen.

Anhand der Operationen und ihrer Sichtbarkeit lassen sich Zugriffe auf eine Klasse und ihre Instanzen sowie angebotene Dienste und deren Ergebnisse (z.B. Rückgabewerte) ermitteln. Parameterlisten informieren über verschiedene Arten des Aufrufs einer Operation. Die Aufteilung in geerbte und selbst definierte Operationen zeigt wie bei den Attributen deren Herkunft an und ermöglicht außerdem einen Überblick darüber, welche Operationen überladen oder überschrieben werden.

Mit diesen Informationen ist eine Klasse ausreichend beschrieben, um ihren Inhalt, ihre Struktur, ihren Zustandsraum und ihr Verhalten zu erkennen und sie gegebenenfalls zu modifizieren oder zu erweitern.

Checkliste Klassenstruktur

Parallel zu einer ersten detaillierteren Klassenbeschreibung sind die *Vererbungsstrukturen* der Klassen zu installieren. Bei der Definition von Attributen und der Identifikation von Operationen werden Sie schnell auf Gemeinsamkeiten einzelner Klassen stoßen, die über Vererbungsstrukturen abgebildet werden können.

Für die Vorgehensweise bei der Definition von Vererbungsstrukturen bestehen grundsätzlich zwei Möglichkeiten.

- ✓ Bei der *Generalisierung* sammeln Sie Klassen mit gleichen Eigenschaften und fassen die Gemeinsamkeiten der Klassen in einer eigenen Oberklasse zusammen.
- ✓ Umgekehrt werden bei der *Spezialisierung* bestehende Klassen daraufhin untersucht, ob sie allgemeine Sachverhalte definieren, die in speziellen Unterklassen konkretisiert werden können.



Vermeiden Sie es unbedingt, »wild drauflos« zu vererben. Untersuchen Sie jede Generalisierung oder Spezialisierung kritisch daraufhin, ob in diesem Fall eine Vererbung wirklich sinnvoll ist. Die Versuchung, jede spezielle Art einer Klasse in einer Unterklasse zu konkretisieren, ist groß. Eine wirklich sinnvolle Vererbungsstruktur liegt in den meisten Fällen nur dann vor, wenn die Unterklassen sich durch eigene Attribute und Operationen unterscheiden, die sie auch tatsächlich benötigen.

Die beiden folgenden Abbildungen zeigen die alternativen Vorgehensweisen bei der Bildung von Vererbungsstrukturen an dem bekannten Kontenbeispiel.

In der ersten Abbildung ist die Zusammenfassung von Klassen durch Generalisierung dargestellt.

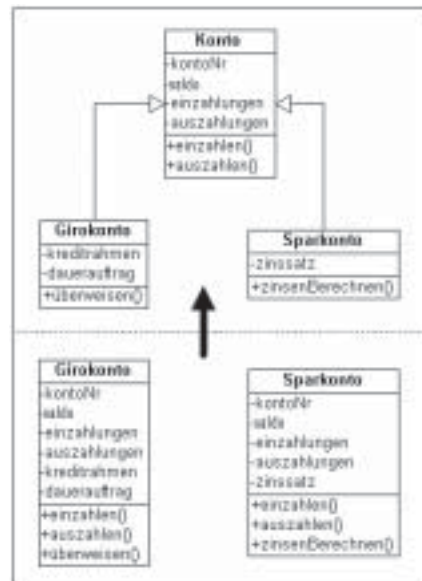


Abbildung 8.1: Zusammenfassung von Klassen durch Generalisierung

Zunächst wurden die beiden Klassen Sparkonto und Girokonto als Elemente der zu modellierenden Problemdomäne »Bank« identifiziert (untere Hälfte der obigen Abbildung). Beim Vergleich der beiden Klassen ist festzustellen, dass diese einige gemeinsame Eigenschaften aufweisen. Sie besitzen jeweils die Attribute `kontoNr`, `saldo`, `einzahlungen` und `auszahlungen`. Außerdem wird das Verhalten in beiden Fällen u.a. durch die Operationen `einzahlen` und `auszahlen` repräsentiert.

Diese gleichen Eigenschaften werden in einer gemeinsamen Oberklasse `Konto` zusammengefasst. Die Klasse `Konto` beschreibt nun generelle Eigenschaften von Konten (obere Hälfte der Abbildung). `Girokonto` und `Sparkonto` sind spezielle Ausprägungen von Konten, die den von der Oberklasse geerbten (generellen) Eigenschaften eigene (spezielle) Attribute und Operationen hinzufügen. Beim `Girokonto` sind dies die Attribute `kreditrahmen` und `dauerauftrag` sowie die Operation `überweisen`. Die Klasse `Sparkonto` besitzt als zusätzliche Eigenschaften das Attribut `zinssatz` und die Operation `zinsenBerechnen`.

Abbildung 8.2 zeigt das umgekehrte Vorgehen bei der Konkretisierung von Klassen durch Spezialisierung.

Hier wurde zunächst die Klasse `Konto` als Teil der Problemdomäne identifiziert. Wie in der oberen Hälfte der Abbildung zu sehen, besitzt sie die Attribute `kontoNr`, `saldo`, `einzahlungen` und `auszahlungen` sowie die Operationen `einzahlen` und `auszahlen`.

Da eine Bank verschiedene Arten von Konten führt, können diese allgemeinen Eigenschaften für spezielle Konten konkretisiert werden. Inhaber von Girokonten erhalten zusätzlich einen bestimmten Kreditrahmen und dürfen Überweisungen vornehmen, für die auch Daueraufträge eingerichtet werden können. Die Klasse `Konto` wird daher erweitert um die Attribute `kreditrahmen` und `dauerauftrag` sowie um die Operation `überweisen`. Inhaber von Sparkonten erhalten hingegen Sparzinsen, die in dem Beispiel durch das Attribut `zinssatz` und die Methode `zinsenBerechnen` repräsentiert werden.

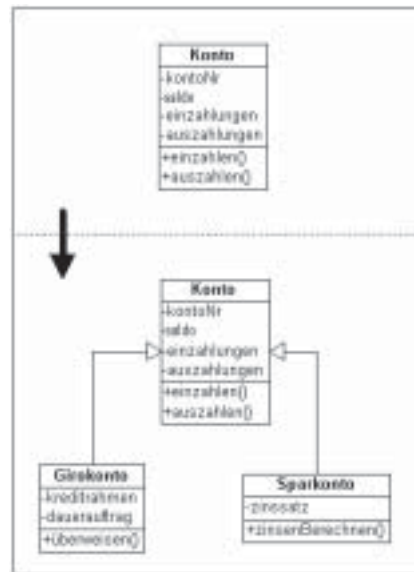


Abbildung 8.2: Konkretisierung von Klassen durch Spezialisierung

In der unteren Hälfte der Abbildung ist die Vererbungsstruktur dargestellt, die sich aus diesen Überlegungen ergibt. Die gemeinsamen Eigenschaften von Girokonto und Sparkonto stehen in der Oberklasse Konto (Generalisierung). Sie werden durch weitere konkrete Eigenschaften in den beiden Unterklassen erweitert (Spezialisierung).



Bei der Ermittlung von Klassenstrukturen bietet es sich an, beide Verfahren zu kombinieren. Sowohl durch Generalisierung als auch durch Spezialisierung werden Sie unter den von Ihnen identifizierten Klassen geeignete Kandidaten für die Bildung von Ober- und Unterklassen finden.



Vererbungsstrukturen finden Sie auch, indem Sie nach so genannten »Ist ein«-Beziehungen suchen. Eine solche Beziehung liegt vor, wenn ein Objekt der Unterklasse sich auch als (spezielles) Objekt der Oberklasse bezeichnen lässt.

Zutreffend ist die Aussage »ist ein« beispielsweise in folgenden Fällen:

- ✓ »Ein PKW *ist ein* Fahrzeug.«
- ✓ »Ein Hund *ist ein* Tier.«
- ✓ »Ein Sparkonto *ist ein* Konto.«

Daraus ergeben sich sinnvolle Vererbungsbeziehungen zwischen der Oberklasse Fahrzeug und der Unterklasse PKW, zwischen der Oberklasse Tier und der Unterklasse Hund sowie zwischen der Oberklasse Konto und der Unterklasse Sparkonto.

Die folgende Checkliste unterstützt Sie bei der Bildung von Klassenstrukturen.

Aktivität	Zu klärende Fragen
Ermittlung von Vererbungsstrukturen	Gibt es Gemeinsamkeiten zwischen bereits identifizierten Klassen? Besitzen bereits identifizierte Klassen spezielle Ausprägungen, die sich ihrerseits durch zusätzliche Eigenschaften auszeichnen? Sind in der Problemdomäne hierarchische Strukturen erkennbar? Bestehen Ähnlichkeiten zwischen den Elementen der hierarchischen Struktur? Lassen sich Beziehungen der Form »A ist ein B« bilden?
Validierung	Werden alle vererbten Attribute zur Beschreibung des Zustands der Unterklasse benötigt?

Aktivität	Zu klärende Fragen
	Repräsentieren alle vererbten Operationen das Verhalten der Unterklasse? Unterscheiden sich die Unterklassen in ihren Attributen und Operationen? Fördert die Klassenstruktur das Verständnis des Modells?

Tabelle 8.8: Checkliste Klassenstrukturen

Gehen Sie mit dem Vererbungskonzept vorsichtig um. Mehr als fünf Hierarchiestufen erschweren das Verständnis eines objekt-orientierten Modells.

Andererseits hängt die Vererbungstiefe von der individuellen Problemstellung ab. Im konkreten Fall können daher u.U. auch weit mehr Hierarchiestufen sinnvoll sein. Umso genauer und ausführlicher sollte dann aber auch die zugehörige schriftliche Dokumentation ausfallen. Prüfen Sie in solchen Fällen auch, ob sich die Bildung zusätzlicher Pakete anbietet.



Multiple Vererbungsstrukturen sollten vermieden werden, um Konflikte zwischen gleichnamigen Eigenschaften auszuschließen.

Wenn Sie dennoch nicht um die Mehrfachvererbung herumkommen, achten Sie auch hier auf eine ausreichende Dokumentation, die genau festhält, unter welchen Umständen welche Eigenschaften aus welcher Klasse wohin vererbt werden. Idealerweise enthält die Dokumentation zusätzliche Hinweise darüber, wann in solchen Fällen mit Konflikten zu rechnen ist, und bietet Konfliktlösungsstrategien an.

Die Vererbungsstruktur ist besonders gut geeignet, Zusammenhänge in einem Problembereich in einfacher und übersichtlicher Weise

zu verdeutlichen. Sie sollte daher eine möglichst genaue Abbildung der Struktur der realen Problemstellung darstellen. Allerdings sollten Sie auch nicht versuchen, um jeden Preis Vererbungsstrukturen zu bilden. Prüfen Sie anhand der oben aufgeführten Fragen kritisch, ob sich im konkreten Fall eine Generalisierung oder Spezialisierung anbietet oder ob eine andere der objektorientierten Beziehungsstrukturen vielleicht besser geeignet ist.

Checkliste Klassenbeziehungen

Neben der Bildung von Klassenhierarchien gehört auch die Identifikation von Klassenbeziehungen zur Entwicklung des statischen Objektmodells. Mögliche Formen der Klassenbeziehungen sind die einfache Assoziation, die Aggregation und die Komposition.

Als Quellen können Sie wiederum die bereits bei der Ermittlung von Klassen verwendeten Dokumente bzw. Techniken heranziehen.

Als sinnvolle Vorgehensweise hat sich die schrittweise Konkretisierung bewährt. Zunächst werden anhand der verfügbaren Quellen Assoziationen innerhalb der Problemstellung identifiziert, ohne bereits festzulegen, ob es sich um einfache Assoziationen, Aggregationen oder Kompositionen handelt. Auch die Kardinalitäten interessieren zunächst nicht.



Beinahe jedes Element der Problemdomäne wird mit den anderen in irgendeiner Beziehung stehen. Die Hauptaufgabe des Modellierers ist es, diejenigen Beziehungen herauszufiltern, die für die Problemlösung von Bedeutung sind.



Gehen Sie bei der Modellierung von Assoziationen zu Beginn möglichst restriktiv – z.B. anhand einer Prioritätenliste – vor. So bleibt Ihr Modell zunächst überschaubar. Falls es sich als zweckmäßig erweist, können Sie die Klassenbeziehungen später immer noch ausweiten oder zur näheren Beschreibung der damit verbundenen Verarbeitungen Assoziationsklassen bilden.

Folgende Checkliste hilft Ihnen beim Aufspüren und Konkretisieren von Beziehungen.

Aktivitäten	Zu klärende Fragen
Beziehungen finden	Welche Elemente tauschen Nachrichten aus? Welche Elemente bieten Dienste für andere Elemente an? Sind Elemente vorhanden, die bestimmte Rollen einnehmen? Lassen sich zusammengehörige organisatorische Einheiten identifizieren? Welche Beziehungen lassen sich aus Datenbanken entnehmen (Primär- und Fremdschlüssel)?
Beziehungen bestimmen	Setzt sich ein Element aus anderen Elementen zusammen (Aggregation)? Ist die Existenz eines Elements abhängig von der Existenz eines anderen Elements (Komposition)? Ist mit der Assoziation eine Verarbeitung verbunden (Assoziationsklasse)? Wie viele Objekte sind auf beiden Seiten mindestens an der Beziehung beteiligt? Wie viele Objekte sind auf beiden Seiten höchstens an der Beziehung beteiligt? Ist die Zahl der beteiligten Objekte auf einer der beiden Seiten stets gleich? Unterliegt die Beziehung irgendwelchen besonderen Zusicherungen?
Validierung	Ist die identifizierte Assoziation für das Modell relevant? Handelt es sich um eine Assoziation oder um eine Vererbungsbeziehung?

Tabelle 8.9: Checkliste Klassenbeziehungen



Sie können jedes Attribut aus der zugehörigen Klasse herausziehen, in einer eigenen Klasse näher beschreiben und diese dann als Aggregat der ursprünglichen Klasse wieder zuordnen. Gehen Sie aber mit dieser Option sparsam um, da sie Ihr Modell kompliziert. Verwenden Sie Aggregationen nur dann, wenn sie dem Modell wirklich wichtige Informationen hinzufügen.

Assoziationen finden Sie auch, indem Sie Ihre Problemdomäne auf Beziehungen untersuchen, die einer der folgenden Aussagen entsprechen:

- ✓ A hat ein B
- ✓ A ist Komponente von B
- ✓ A ist B zugeordnet
- ✓ A enthält B
- ✓ A besteht aus B
- ✓ A kommuniziert mit B
- ✓ A besitzt B
- ✓ A gehört zu B
- ✓ A benutzt B
- ✓ A verwaltet B
- ✓ A plant B
- ✓ A führt B aus
- ✓ A kontrolliert B
- ✓ A beschreibt B



In einigen Fällen bietet es sich an, zwei Elemente in einer Klasse zusammenzufassen, statt eine Assoziation zu bilden. Das eine Element bildet dann ein Attribut des anderen. Dies ist z.B. dann

zweckmäßig, wenn zwischen zwei Klassen mehrere verschiedene Assoziationen bestehen oder eine der Klassen nur sehr wenige eigene beschreibende Attribute besitzt.

Assoziationen sollten stets mit aussagekräftigen Bezeichnungen versehen werden. Reichen diese zum Verständnis nicht aus, kann zusätzlicher erklärender Text in das Objektmodell aufgenommen werden. Auch hier ist zur Förderung des Verständnisses und zur Erleichterung späterer Anpassungen und Erweiterungen des Modells eine ausführliche Dokumentation dringend zu empfehlen.

Dynamische Analyse

Mit dem statischen Objektmodell haben Sie die Strukturen Ihrer Problemdomäne nun weitgehend erfasst und beschrieben. Die Definition der Klassen mit ihren Eigenschaften und Beziehungen ist bis hierher jedoch nicht viel mehr als eine Sammlung, Beschreibung und Gliederung von Begriffen. Im Rahmen der dynamischen Analyse gilt es nun, diese Strukturen »in Bewegung zu setzen«.



Die Beschreibung der Abläufe innerhalb des Objektmodells deckt eventuelle Schwächen der Problemabgrenzung und der Klassendefinition auf. Hier muss eindeutig festgelegt werden, wie die Objekte miteinander interagieren. Dabei stellt sich heraus, ob die in den Objekten gekapselten Informationen und die installierten Beziehungen der Problemlösung gerecht werden oder unter Umständen revisionsbedürftig sind.

Das dynamische Modell repräsentiert das Verhalten von Objekten. Es bildet Interaktionen, Zustände und Aktivitäten ab. Im Einzelnen sind mit der Erstellung des dynamischen Modells die folgenden Aufgaben verbunden:

- ✓ Ermittlung von Interaktionen
- ✓ Identifikation von Ereignissen
- ✓ Definition von Zuständen und Zustandsübergängen
- ✓ Identifikation von Aktivitäten
- ✓ konkrete Beschreibung der im Zusammenhang mit der Klassendefinition ermittelten Operationen

Aus der statischen Analyse liegen die modellierten Klassen und ihre Eigenschaften (Attribute und Operationen) vor. Außerdem wurden dort bereits Vererbungsstrukturen und Beziehungen gebildet.

Auf der Grundlage dieser Informationen können nun für das dynamische Modell die Interaktionen zwischen den beteiligten Objekten ermittelt und in Interaktionsdiagrammen näher beschrieben werden. Sequenz- und Kollaborationsdiagramme spezifizieren die Kommunikationsbeziehungen, die durch die Assoziationen in den Klassendiagrammen zunächst noch auf recht abstraktem Niveau definiert sind.

Anschließend werden die Zustände der Objekte und ihre Zustandsübergänge in Zustands- und Aktivitätsdiagrammen beschrieben. Zu diesem Teil des Modellierungsprozesses gehört auch die Identifizierung von Ereignissen, die Zustandsübergänge auslösen.

In einem weiteren Schritt sind dann die eingangs bei der Entwicklung des statischen Modells benannten Methoden auf der Grundlage der bisherigen Spezifikation des dynamischen Modells zu konkretisieren. Die Modellierungsaktivitäten umfassen dabei die Formulierung von Algorithmen zur exakten Beschreibung der von den Methoden durchgeführten Prozesse sowie die Definition von Parametern zur Realisierung von Schnittstellen.

Die folgenden Abschnitte gehen ausführlich auf diese Aufgaben bei der Entwicklung des dynamischen Modells ein und stellen Ihnen wiederum Checklisten zur Verfügung, die Sie im Verlauf des Modellierungsprozesses unterstützen.

Checkliste Interaktionen

Zahlreiche potenzielle Quellen bieten sich zur Ermittlung von Interaktionsbeziehungen an. Erste Erkenntnisse können Sie aus den Ergebnissen der bereits durchgeführten statischen Analyse beziehen:

- ✓ Die *Problemabgrenzung* liefert meist nur recht allgemeine Hinweise auf Interaktionen. Zumindest können Sie hieraus entnehmen, welche Elemente potenzielle Kandidaten für Interaktionsbeziehungen sind und welche nicht. Oft enthält auch eine verbale Beschreibung oder Begründung der Problemabgrenzung Anzeichen für Interaktionen zwischen Objekten.
- ✓ Die *Klassenbeschreibungen* enthalten Informationen über die beteiligten Operationen, die wiederum für das Verhalten der Objekte verantwortlich sind.
- ✓ *Vererbungsbeziehungen* können auch auf Interaktionen hinweisen, wenn beispielsweise Klassenmethoden benutzt werden oder polymorphe Zugriffe auf Operationen erfolgen.
- ✓ *Assoziationen* zeigen Ihnen, welche Objekte miteinander kommunizieren, um Daten oder Dienste auszutauschen.

Weitere geeignete Quellen für Interaktionsbeziehungen sind:

- ✓ *Verteilerlisten*
Aus Verteilerlisten können Sie entnehmen, wer von wem welche Informationen erhält.
- ✓ *Organigramme*
Organigramme zeigen statische Strukturen. Dennoch weisen sie durch die Zusammenfassung von Stellen auch auf Kommunikationsstrukturen hin.
- ✓ *Telefon- und Faxverzeichnisse*
In vielen Unternehmungen existieren speziell für die einzelnen Stellen oder Abteilungen abgestimmte Telefon- und Faxverzeichnisse. Sie enthalten nicht nur Telefon- und Faxnummern, sondern geben zudem Auskunft darüber, mit welchen Partnern

die entsprechende Abteilung oder der Stelleninhaber bei der Bewältigung der anfallenden Aufgaben kommuniziert.

- ✓ *Netzwerkstrukturen (lokales Netzwerk, Intranet)*
Die Verbindung von Rechnern oder Teilnetzen zu Netzstrukturen zeigt Kommunikationswege auf.
- ✓ *Stellenbeschreibungen*
Stellenbeschreibungen sind eine gute Quelle für die Beschreibung der konkreten Aktivitäten, die mit einer Stelle verbunden sind. Diese Aktivitäten werden im Objektmodell durch Operationen repräsentiert, die wiederum das Verhalten der zugehörigen Objekte abbilden. So gelangen Sie zu den Informationsflüssen, die einen Produktions- oder Dienstleistungsprozess begleiten.
- ✓ *Berichtswesen*
Berichte geben Aufschluss darüber, wer an wen welche Informationen in welchen zeitlichen Abständen weitergibt.
- ✓ *Dokumentationen der eingesetzten Hard- und Software*
Die Schnittstellen zwischen den einzelnen Elementen der Hardware- und Software-Infrastruktur geben Aufschluss über elektronische Kommunikationswege und Datenflüsse.
- ✓ *Interviews*
Wie schon im Rahmen der Ermittlung von Klassenstrukturen helfen Ihnen Interviews auch bei der Identifikation von Interaktionsbeziehungen. Sie sind vor allem ein geeignetes Mittel, um informale Kommunikationswege zu erfassen, die nicht schriftlich dokumentiert sind.
- ✓ *Ablagesystem (abgeschlossene Vorgänge)*
Aus der Dokumentation abgeschlossener Vorgänge können Sie Informationsflüsse und Kommunikationswege nachträglich erheben, die typisch für bestimmte Geschäftsvorfälle sind oder die sich auf ähnlich gelagerte, zukünftige Fälle übertragen lassen.

✓ *Projektberichte*

In Projektberichten sind zahlreiche Hinweise auf Kommunikationswege zu finden. Verantwortliche für bestimmte Aufgabenbereiche und ihr Informationsaustausch mit verschiedenen Kommunikationspartnern sind hier dokumentiert.



Bei der Vielzahl der zur Verfügung stehenden Quellen führt deren Auswertung häufig zu einer riesigen Datenflut. Um daraus die relevanten Informationen herauszufiltern, ist neben der bereits mehrfach betonten ausführlichen Dokumentation des Anwendungsfalls und der damit verfolgten Ziele auch eine systematische Vorgehensweise bei der Auswertung erforderlich.

Eine mehrfache Auswertung von Quellen ist in der Regel nicht zu vermeiden, da Sie erst im weiteren Verlauf des Modellierungsprozesses feststellen, ob Sie alle benötigten Informationen gesammelt haben. Häufig sind Sie erst in einem späteren Stadium Ihres Modellierungsprojekts in der Lage, den Informationsstand – gerade was die dynamischen Aspekte anbelangt – erschöpfend beurteilen zu können.



Die mit der Informationsbeschaffung und -verarbeitung betrauten Personen sollten eng in die Erstellung des Objektmodells eingebunden sein, um die Relevanz der gesammelten Informationen besser beurteilen zu können. Von der Auswertung durch Hilfskräfte, die wenige oder keine Kenntnisse der objektorientierten Analyse und des konkreten Gestaltungsprojekts haben, ist ohne eingehende Schulung dringend abzuraten.

Die folgende Checkliste erleichtert Ihnen die Feststellung und Beschreibung von Interaktionsbeziehungen.

Aktivität	Zu klärende Fragen
Erfassung von Interaktionen	Welche Assoziationen enthält das statische Modell? Welche Objekte arbeiten an einer gemeinsamen Aufgabe? Welche Objekte sind Dienstleister für andere Objekte? Welche Objekte benötigen Informationen von anderen Objekten? Welche Objekte kommunizieren miteinander?
Beschreibung von Interaktionen	Durch welche Ereignisse wird eine Interaktion ausgelöst? Über welche Schnittstellen tauschen die Objekte Informationen aus? Über welche Kommunikationswege werden Informationen ausgetauscht? Welche Informationen werden ausgetauscht? Wann finden Interaktionen statt? Wo werden Informationen ausgetauscht? In welcher Form werden Informationen übertragen? In welchen Zuständen werden Informationen ausgetauscht? Unterliegt die Kommunikation zweier Objekte speziellen Bedingungen? Wie ist der Lebenszyklus der beteiligten Objekte definiert? Wann werden Objekte erzeugt oder gelöscht? Welche Operationen kooperieren bei der Abbildung einer Interaktion? In welcher Reihenfolge sind die Operationen durchzuführen?

Aktivität	Zu klärende Fragen
Validierung	Was ist das Ziel der Kommunikation zweier Objekte? Ist die Aufnahme der betrachteten Kommunikationsbeziehung in das Modell erforderlich? Sind alle Objekte nach dem derzeitigen Modell in der Lage, ihre Interaktionspartner zu erreichen (bestehen entsprechende Verbindungen im Modell)? Wurden alle benötigten Varianten einer Interaktion im Modell berücksichtigt? Lässt sich das Kommunikationsgeflecht durch eine Umstrukturierung von Aufgaben vereinfachen? Welche Auswirkungen hat die Umstrukturierung von Aufgaben auf die Zielerreichung des Systems? Stimmen die Interaktionsdiagramme mit den Klassendiagrammen überein?

Tabelle 8.10: Checkliste Interaktionen

Ergebnis der Analyse der Interaktionen sind verschiedene Sequenz- und Kollaborationsdiagramme.



Insbesondere dann, wenn das Ziel Ihrer objektorientierten Analyse die Entwicklung einer objektorientierten Software zur Automatisierung oder Unterstützung eines Geschäftsprozesses ist, sollten Sie bei der Erstellung des dynamischen Modells mit äußerster Sorgfalt vorgehen. Stimmen die Abläufe in der Software bzw. deren Resultate nicht mit den realen Vorgängen überein, hat dies direkte Konsequenzen für die Geschäftsprozesse selbst.

Das objektorientierte Modell ist zwar zunächst lediglich eine Repräsentation der Problemdomäne – letztendlich beeinflusst es jedoch durch die Implementierung in einem Programm auch die realen Strukturen und Vorgänge.

Die Darstellung dynamischer Aspekte einer Problemstellung ist stets mit Schwierigkeiten hinsichtlich ihrer Übersichtlichkeit verbunden. Die UML-Notation ist gegenüber ihren Vorgängerinnen und anderen Modellierungssprachen schon recht gut geeignet, Abläufe vollständig, konsistent und zugleich verständlich zu visualisieren. Dennoch werden Modelle unübersichtlich, wenn sie den Umfang weniger Bildschirm- oder Papierseiten wesentlich überschreiten. In einem gewissen Maße kann hier die Verkleinerung der Ansicht bzw. der Ausdruck auf mehrere Papierseiten oder größere Papierformate Abhilfe schaffen. Befriedigende Lösungen sind das aber nicht.



Versuchen Sie daher, möglichst übersichtliche Sequenzen zu bilden. Trennen Sie dazu, wenn möglich, umfangreiche Vorgänge in mehrere Teilsequenzen auf. Dadurch erhöhen Sie die Übersichtlichkeit und auch die Verständlichkeit Ihres Objektmodells. Änderungen und Ergänzungen sind so zudem leichter vorzunehmen. Eine Zusammenschau der Teilsequenzen kann in einem zusätzlichen Diagramm realisiert werden, das einen abstrakten Gesamtüberblick ohne Details bietet.

Alternativ können Sie auch Sequenzdiagramme einsetzen, um vollständige Abläufe abzubilden. Diese ergänzen Sie dann um Kollaborationsdiagramme, mit denen Sie die Zusammenarbeit einzelner Objekte detailliert hervorheben.

Checkliste Zustände

Neben der Abbildung von Interaktionen in Sequenz- und Kollaborationsdiagrammen gehört zu einem dynamischen Objektmodell auch die Beschreibung von Zuständen und Zustandsübergängen.

Sie werden repräsentiert durch Zustandsdiagramme, die der Darstellung der Zustände eines Objekts in seinem Lebenszyklus sowie der Abbildung von Zustandsänderungen und auslösenden Ereignissen dienen.



Denken Sie daran: Ein Zustandsdiagramm beschreibt immer nur die Zustände *eines* Objekts.

Bei der Erstellung eines Zustandsdiagramms gehen Sie am besten wie folgt vor. Ermitteln Sie zunächst den Anfangszustand des Objekts nach seiner Entstehung. Anschließend beschreiben Sie die verschiedenen Zustände, die im Laufe des Lebenszyklus eingenommen werden können. Berücksichtigen Sie dabei auch die Ereignisse, die zu einem Zustandswechsel führen.

Ein Zustand, aus dem heraus kein Zustandsübergang mehr erfolgen kann, ist ein Endzustand. Nachdem Sie die einzelnen Zustände benannt und eventuell grob beschrieben haben, wiederholen Sie den Prozess und detaillieren Sie Ihr Zustandsdiagramm so weit wie nötig.

Die Aktivitäten zur Erstellung eines Zustandsdiagramms überlappen sich mit der im nächsten Abschnitt beschriebenen Identifikation von Ereignissen. Einige potenzielle Zustände ihrer Objekte werden Sie u.U. erst bei der Ermittlung und Beschreibung der Ereignisse erkennen, die Zustandsänderungen auslösen. Somit ist auch hier wieder ein zyklisches Vorgehen angebracht.

Bei der Ermittlung von Zuständen und Zustandsübergängen hilft die folgende Checkliste.

Aktivität	Zu klärende Fragen
Ermittlung von Zuständen	Wann wird das Objekt erzeugt? Welche Initialwerte besitzt das Objekt? Wie ist der Anfangszustand des Objekts? Welche Werte können die Attribute des Objekts annehmen?

Aktivität	Zu klärende Fragen
	Führt das Objekt beim Eintritt in den betrachteten Zustand Aktionen aus (Entry-Aktionen)? Führt das Objekt in dem betrachteten Zustand Aktionen aus (Do-Aktionen)? Führt das Objekt beim Austritt aus dem betrachteten Zustand Aktionen aus (Exit-Aktionen)? Rufen die mit dem betrachteten Zustand verbundenen Aktionen externe Subroutinen auf (Include-Aktionen)? Welche Endzustände besitzt das Objekt? Gibt es anonyme Zustände? Endet die Existenz des Objekts mit seinem Endzustand?
Ermittlung von Zustandsübergängen (vgl. auch die Checkliste zu den Ereignissen)	Welche Schnittstellen besitzt das Objekt? Welche eigenen Operationen manipulieren die Attributwerte (= Zustand) des Objekts? Wodurch werden diese Operationen aktiviert? Welche konkreten Ereignisse führen zu einem Zustandsübergang? Unter welchen Bedingungen führt ein Ereignis zu einem Zustandsübergang? Welche impliziten Ereignisse können auftreten? Kann ein und dasselbe Ereignis unterschiedliche Reaktionen und Zustandsübergänge auslösen?
Validierung	Sind die identifizierten Zustände für das Systemverständnis erforderlich? Sind alle im Diagramm definierten Zustände vom Anfangszustand aus durch Ereignisse und Zustandsübergänge erreichbar?

Aktivität	Zu klärende Fragen
	Können alle Zustände (abgesehen von den Endzuständen) durch Ereignisse verlassen werden? Sind alle Folgezustände eindeutig? Gibt es zu allen identifizierten Aktionen eine entsprechende Operation im Klassendiagramm?

Tabelle 8.11: Checkliste Zustände



Eine »beliebte« Fehlerquelle bei der Entwicklung von Zustandsdiagrammen ist die Darstellung von Abläufen. Die – auch im Hinblick auf die grafischen Darstellungselemente – große Ähnlichkeit mit den Aktivitätsdiagrammen verführt dazu, die Reihenfolge von Arbeitsschritten abzubilden. Dies ist aber gerade nicht Zweck der Zustandsdiagramme.

Die Zustände an sich sind statisch. Ein Objekt verharrt für eine gewisse Zeit in einem bestimmten Zustand. Erst durch die Zustandsübergänge kommt die Dynamik ins Spiel, die Ihnen anzeigt, wann und durch welche Ereignisse ein Objekt seinen Zustand wechselt.



Erfahrungsgemäß fällt es leichter, sich auf die Beschreibung von Zuständen zu konzentrieren, indem Sie deren Benennung durch Verben vermeiden. Verwenden Sie stattdessen Adjektive. Die Bezeichnungen wirken dann zwar oft ein wenig befremdlich. Es wird Ihnen so aber weniger häufig passieren, dass Sie Abläufe statt Zustände beschreiben.

Im folgenden Abschnitt finden Sie weitere Hinweise zur Identifikation von Ereignissen, die für die Zustände bzw. die Zustandsänderungen eine wichtige Rolle spielen.

Checkliste Ereignisse

Die dynamischen Aspekte eines objektorientierten Modells sind äußerst vielfältig und in ihrer Gesamtheit meist kaum zu überblicken. Die Interaktionen der Objekte über den Aufruf von Operationen werden sehr schnell ziemlich komplex. Sie lassen sich oft nur schwer verfolgen, da jede Operation wiederum Operationen anderer Objekte aktivieren kann.

Bei diesen Vorgängen werden Ereignisse ausgelöst, die wiederum zu Zustandsänderungen führen. Im Rahmen eines komplizierten Interaktionsgeflechts ändern sich die Zustände gleich mehrerer Objekte.

Bei der Analyse von Zuständen, Zustandsübergängen und deren auslösenden Ereignissen bildet das statische Modell eine gute Ausgangsbasis. Anhand der Beziehungen zwischen Klassen bzw. Objekten lassen sich potenzielle Quellen von Ereignissen erkennen. Diese können näher daraufhin untersucht werden, inwieweit sie einzelne Objekte in ihren Zuständen beeinflussen.

Im Rahmen der dynamischen Analyse können Sie auf der Suche nach Ereignissen die folgende Checkliste zu Rate ziehen.

Aktivität	Zu klärende Fragen
Ermittlung von Ereignissen	Welche Beziehungen besitzt die Klasse bzw. das Objekt laut statischem Objektmodell? Welche Zustände sind bereits ausgemacht worden? Was unterscheidet diese Zustände? Welche externen Operationen greifen auf das Objekt zu? Welche internen Operationen führen zu Zustandsänderungen und wodurch werden diese Operationen ausgelöst? Welche Ereignisse werden von Objekten ausgelöst? Welche Ereignisse werden von Personen (z.B. Anwender der Software) ausgelöst?

Aktivität	Zu klärende Fragen
	Welche impliziten Ereignisse können auftreten? Welche zeitlichen Ereignisse (Zeitpunkt, Zeitspanne) können auftreten? Welche Bedingungen sind selbst Ereignisse?
Beschreibung der Ereignisse	Unter welchen Bedingungen werden die identifizierten Ereignisse ausgelöst? Führt ein und dasselbe Ereignis bei verschiedenen Zuständen zu unterschiedlichen Reaktionen bzw. Ergebnissen? Betrifft ein Ereignis mehrere Objekte?
Validierung	Ist das betrachtete Ereignis für das Modell von grundsätzlicher Bedeutung? Welche Konsequenzen hat der Verzicht auf die Abbildung des Ereignisses für das Modell? Ist das betrachtete Ereignis bereits in anderer Form oder unter einer anderen Bezeichnung berücksichtigt worden? Kann das betrachtete Ereignis als Spezialfall vernachlässigt werden? Kann das betrachtete Ereignis als Spezialfall in ein eigenes (Teil-) Modell ausgelagert werden?

Tabelle 8.12: Checkliste Ereignisse

Die Dynamik des Verhaltens der Objekte führt auch bei überschaubarer Problemstellung zu komplexen Wechselwirkungen zwischen Zuständen, Ereignissen und Bedingungen, so dass deren vollständige Erfassung nur in einem zyklischen Prozess gelingen kann.



Gerade im Zusammenhang mit der Identifikation von Zuständen, Zustandsübergängen und Ereignissen zeigt sich die Notwendigkeit eines evolutionären Vorgehens bei der Gestaltung des Objektmodells. Beginnen Sie mit der Sammlung und groben Strukturierung offensichtlicher Zustände und Ereignisse. Anschließend ergänzen und verfeinern Sie diese nach und nach. Mit einem streng linearen Gestaltungsprozess kommen Sie hier nicht weit.

Checkliste Aktivitäten

Aktivitätsdiagramme werden häufig mit den Zustandsdiagrammen verwechselt, da sie diesen sehr ähnlich sind. Schließlich ist eine Aktivität auch eine spezielle Art von Zustand, der mit einer internen Aktion verbunden ist und dem ein oder mehrere, durch Bedingungen verknüpfte Zustandsübergänge folgen. Als Aktivität wird dabei ein einzelner Verarbeitungsschritt im Rahmen eines Ablaufs bezeichnet. Aktivitätsdiagramme dienen somit der Beschreibung interner Prozesse in Objekten.

Zur Ermittlung von Aktivitäten können Sie die gleichen Quellen heranziehen, die bereits für die bisherigen Schritte zur Erstellung eines Objektmodells genannt wurden. Darüber hinaus können Sie hier einige zusätzliche Methoden einsetzen, um Aktivitäten zu ermitteln:

✓ *Denkprotokolle*

Lassen Sie Mitarbeiter der Fachabteilung bei der Ausführung ihrer Tätigkeiten Denkprotokolle erstellen. Durch lautes Denken beschreiben die Mitarbeiter ihre Vorgehensweise. Dabei werden häufig eher unbewusste Aktivitäten aufgedeckt, die durch Interviews nicht zur Sprache kommen würden.

✓ *Beobachtung*

Schauen Sie den Mitarbeitern der Fachabteilung während ihrer Arbeit über die Schultern. Als Außenstehender erkennen Sie

eventuell Aktivitäten, von denen Sie mit anderen Methoden nicht erfahren, da die Mitarbeiter sie für unwichtig halten oder vergessen zu erwähnen.

✓ *Protokollanalysen*

Analysieren Sie Vorgangs- und Besprechungsprotokolle, um mehr über Regeln, das Zustandekommen von Entscheidungen und die Vorgehensweise bei der Aufgabenlösung zu erfahren.



Berücksichtigen Sie bei der Analyse von Denkprotokollen und der direkten Beobachtung von Mitarbeitern aber auch die Widerstände, die Ihnen möglicherweise entgegentreten. Die Mitarbeiter fühlen sich (nicht zu Unrecht) beobachtet und kontrolliert. Dadurch könnten Ihre Erkenntnisse verfälscht und unter Umständen, bewusst oder unbewusst, manipuliert werden.

Bei der Analyse der genannten Dokumente und der Verwendung der oben angeführten Erhebungsmethoden können Sie die folgende Checkliste zu Rate ziehen.

Aktivität	Zu klärende Fragen
Identifikation von Aktivitäten	Welche Aktivitäten ergeben sich aus den Zustandsdiagrammen? Welche internen Prozesse laufen in einem Objekt ab? Von welchen Bedingungen hängt die Ausführung der Prozesse ab? Welchen Regeln unterliegt ein Prozess? Wer ist für die Durchführung der Aktivitäten verantwortlich (swimlanes)? Welche Ergebnisse erzeugen die Prozesse? Welche alternativen Zustände sind mit einer Aktivität verknüpft? Wie lauten jeweils die Bedingungen (Entscheidungen) für das Feuern der alternativen Zustandsübergänge?

Aktivität	Zu klärende Fragen
	Führt ein Zustandsübergang zu mehreren verschiedenen Aktivitäten (Splitting)? Haben mehrere parallel laufende Aktivitäten eine gemeinsame Folgeaktivität (Synchronisation)? Ist es sinnvoll, einzelne Objektzustände mit in das Aktivitätsdiagramm aufzunehmen, um Zusammenhänge zu verdeutlichen? Werden Signale an Objekte gesendet? Werden Signale von Objekten empfangen? Mit welchen Objekten werden diese Signale ausgetauscht? Wann werden die Signale ausgetauscht? Unter welchen Bedingungen werden die Signale ausgetauscht?
Validierung	Ist die betrachtete Aktivität wichtig für das Verständnis der Abläufe in dem System? Wurden alle relevanten Aktivitäten berücksichtigt? Wurden alle vorhergehenden Aktivitäten erfasst? Wurden alle Folgeaktivitäten erfasst? Sind alle Aktivitäten als Operationen im Klassenmodell definiert?

Tabelle 8.13: Checkliste Aktivitäten

Die in Ihrem System ablaufenden Prozesse haben Sie damit schon weitgehend beschrieben. Was nun noch fehlt, ist die Spezifikation der Operationen.

Checkliste Operationen

Nachdem Sie im statischen Modell die Operationen identifiziert und mit einem Namen bezeichnet haben, wurden in den letzten

Abschnitten Auslöser und Ergebnisse von Operationen ermittelt. Nun müssen die Operationen noch in ihrer konkreten internen Struktur und in ihren Abläufen definiert werden.

Sammeln Sie zunächst aus dem statischen Modell sämtliche Operationen und ordnen Sie diese den einzelnen Objekten zu.



Vergessen Sie dabei nicht die Operationen, die aus Oberklassen vererbt werden.

Falls noch nicht bei der Identifikation und Beschreibung der Klassen im Rahmen der Entwicklung des statischen Modells geschehen, beschreiben Sie die Methoden verbal. Diese Dokumentation sollte einige wichtige Komponenten enthalten, die für das Verständnis und eine spätere Ergänzung oder Anpassung der Methoden unentbehrlich sind:

- ✓ Name der Operation
- ✓ Parameterliste mit zugehörigen Datentypen
- ✓ Zweck
- ✓ Datentyp der verarbeiteten Daten
- ✓ Datentyp der Operation selbst
- ✓ Vorgehensweise
- ✓ Ergebnis
- ✓ Rückgabewerte
- ✓ Sichtbarkeit
- ✓ Herkunft der Operation (Vererbungsstruktur)

Die folgende Checkliste unterstützt Sie bei der näheren Beschreibung der Operationen.

Aktivität	Zu klärende Fragen
Beschreibung der Operationen	Welchem Zweck dient die Operation? Wodurch wird die Operation ausgelöst? Welche Daten benötigt die Operation zur Durchführung ihrer Aufgabe? Wie führt die Operation ihre Aufgabe aus? In welcher Reihenfolge werden bestimmte Teilaktivitäten ausgeführt? Unter welchen Bedingungen werden bestimmte Teilaktivitäten ausgeführt? Auf welche internen Attribute wird bei der Verarbeitung zugegriffen? Welche internen Attribute werden manipuliert? Welche Dienste anderer Objekte benötigt die Operation? Welche Informationen benötigt die Operation von anderen Objekten? Welche Informationen benötigen andere Objekte von dieser Operation? Welche Ergebnisse erzielt die Operation? Sendet die Operation eine Antwort an das aufrufende Objekt? Übermittelt die Operation ihre (Zwischen-) Ergebnisse an das aufrufende Objekt? In welchem Bereich ist die Operation sichtbar (public, protected, private)?
Validierung	Ist der in der Klassenbeschreibung gewählte Name der Operation aussagekräftig? Ist der Name der Operation in dem jeweiligen Namensraum (Paket) eindeutig? Wird die Operation in allen möglichen Fällen ordnungsgemäß beendet? Kann die Operation auf alle erforderlichen Daten (Attribute, Parameter) zugreifen? Hat die Operation Zugriff auf alle benötigten externen Operationen (Sichtbarkeit)?

Aktivität	Zu klärende Fragen
	Ist die Operation für alle anderen Objekte, die auf sie zugreifen müssen, sichtbar? Ist die Parameterliste vollständig? Haben alle Parameter den korrekten Datentyp? Gibt die Operation die richtigen Werte zurück? Ist der Datentyp der Operation für alle möglichen Aufrufe korrekt?

Tabelle 8.14: Checkliste Operationen

Nachdem Sie die Operationen zunächst verbal beschrieben haben, wird sie anschließend in eine formale Sprache umgesetzt. Sie können hierzu eine Programmiersprache wie z.B. Java verwenden.

Häufig wird aber an dieser Stelle auch auf die formale Definition verzichtet. Im Rahmen des objektorientierten Programmierungsprozesses fällt erst später eine Entscheidung über die zu verwendende Programmiersprache. Die formale Umsetzung der Operationen wird dann im Rahmen der Implementierung vorgenommen.

KAPITEL

Modellierung von Geschäftsprozessen

In diesem Kapitel stellen wir Ihnen die objektorientierte Analyse von Geschäftsprozessen anhand des Beispiels eines Web-Shops vor.

9



Modellierung von Geschäftsprozessen

Die bisher vorgestellten Konzepte der UML werden in diesem Kapitel noch einmal im Zusammenhang an einem Fallbeispiel zur Geschäftsprozessanalyse verdeutlicht. Von der groben Systemanalyse über die Erstellung des statischen Modells bis hin zur Spezifikation des dynamischen Modells werden die einzelnen Schritte im Rahmen der objektorientierten Analyse und die zugehörigen UML-Diagramme für eine konkrete Anwendung entwickelt.

Als Fallbeispiel dient hierzu der Online-Shop eines Internet-Händlers. Die Vorgänge von der Bestellung durch den Kunden bis hin zur Auslieferung werden in Auszügen in einem objektorientierten Modell abgebildet.

Grobe Systemanalyse

Zunächst ist eine grobe Systemanalyse durchzuführen. Die dabei eingenommene Außensicht auf die Problemstellung behandelt die internen Abläufe des Geschäftsprozesses als black box. In dieser ersten Abgrenzung des zu modellierenden Systems geht es lediglich darum, grundsätzliche Elemente und Vorgänge des Geschäftsprozesses zu erfassen und zu benennen.

Im Mittelpunkt des Interesses steht hier, *was* in dem betrachteten Geschäftsprozess vorgeht. Nicht beachtet wird zunächst, *wie* diese Vorgänge im Einzelnen ablaufen.

Im Rahmen der groben Systemanalyse sind folgende Schritte durchzuführen, die im vorangegangenen Kapitel näher erläutert wurden:

- ✓ Beschreibung des Geschäftsprozesses
- ✓ Abgrenzung des relevanten Problemausschnitts
- ✓ Abgrenzung von Teilsystemen (Paketen)
- ✓ Definition von Schnittstellen zwischen Paketen

Beschreibung des Geschäftsprozesses

Zunächst ist der Geschäftsprozess in groben Zügen verbal zu beschreiben. Daraus werden die beteiligten Akteure, ihre Aufgaben und auftretende Ereignisse abgeleitet. Im Laufe der weiteren Ermittlung von Elementen und ihrer Beschreibung wird der Geschäftsprozess ständig verfeinert und erweitert, unter Umständen aber auch aufgrund neuer Erkenntnisse eingeschränkt.

Die Beschreibung des Geschäftsprozesses soll im Wesentlichen die folgenden Aspekte beinhalten:

- ✓ Gewünschte Ergebnisse (Ziele) des betrachteten Prozesses
- ✓ Bedingungen oder Zusicherungen, die bei der Durchführung des Prozesses einzuhalten sind
- ✓ Regeln für den Prozessablauf
- ✓ Alternativen im Prozessablauf und Voraussetzungen, die einen alternativen Prozessverlauf auslösen

Um die benötigten Informationen zu sammeln, verwenden Sie die im achten Kapitel genannten Dokumente und Techniken.

Wir nehmen an, dass sich in unserem konkreten Beispiel nach der Befragung des Auftraggebers folgende, zunächst noch relativ grobe, verbale Beschreibung des Geschäftsprozesses »Online-Kauf« ergibt:

Der Internet-Händler verkauft online über seine Webseite Bücher und Musik-CDs. Der potenzielle Kunde ruft dazu die Internetseite auf und sucht in der angebundenen Datenbank nach den Artikeln, die ihn interessieren. Die Artikel werden ihm mit allen benötigten Informationen ange-

zeigt. Dazu zählen u.a. Artikelbezeichnungen, Autoren, Bestellnummern und Preise. Zu den Büchern erscheint ein Abdruck des Klappentextes und das Inhaltsverzeichnis, bei den Musik-CDs sind die Titel der einzelnen Stücke aufgeführt.

Der Kunde kann sich einen Warenkorb mit Büchern und CDs zusammenstellen und diese dann online bestellen. Die Bezahlung erfolgt per Bankinzug. Allerdings muss der Kunde zunächst registriert sein, um diesen Service nutzen zu können. Falls ein Artikel zurzeit nicht lieferbar ist, kann er auch nicht bestellt werden. Der Kunde erhält dann einen entsprechenden Hinweis.

Für die Abwicklung des Auftrags benötigt der Händler neben dem Namen und der Bankverbindung auch eine Rechnungs- und eine Lieferadresse. Außerdem wird die E-Mail-Adresse des Kunden gespeichert, um ihn über den Status seiner Bestellung zu informieren.

Die bestellten Artikel werden im weiteren Verlauf des Geschäftsprozesses gesammelt, verpackt und an den Kunden ausgeliefert. Parallel dazu wird der entsprechende Geldbetrag vom Konto des Kunden abgebucht. Sobald der Auftrag abgeschlossen ist, erhält der Kunde per E-Mail eine Bestätigung, dass die von ihm bestellten Artikel in den nächsten Tagen ausgeliefert werden.

Aus dieser verbalen Beschreibung ergeben sich einige Anhaltspunkte für die eingangs beschriebenen Aspekte, die eine solche Beschreibung enthalten soll. Sie sind in der folgenden Tabelle zusammengefasst.

Merkmale	Online-Kauf
Ziele des Geschäftsprozesses	Abwicklung der Bestellung von Büchern und Musik-CDs über das Internet
Bedingungen und Zusicherung im Rahmen des Geschäftsprozesses	Kunde muss registriert sein Ware muss lieferbar sein

Merkmale	Online-Kauf
Regeln für den Prozessablauf	u.a. Reihenfolge der einzelnen Schritte (s.o.)
Alternativen und Voraussetzungen im Prozessablauf	u.a.: Falls der Kunde noch nicht registriert ist, kann er nicht bestellen

Tabelle 9.1: Merkmale des Geschäftsprozesses

Die Tabelle zeigt lediglich einige beispielhafte Merkmale des Geschäftsprozesses. Bei der Modellierung umfangreicher Prozesse ist es durchaus sinnvoll, die verbale Beschreibung noch einmal in einer Übersicht zusammenzufassen und so die einzelnen Merkmale des Prozesses deutlich hervorzuheben. Dies gilt insbesondere dann, wenn ein Prozess so komplex ist, dass die verbale Beschreibung sehr umfangreich wird.

Zur Vorbereitung des Use-Case-Diagramms für die Beschreibung des Online-Kaufs wird die Checkliste aus Kapitel 8 herangezogen.

Ermittlung von Akteuren

Zunächst wird nach möglichen Akteuren gesucht.

- ✓ *Welcher Akteur löst den Geschäftsprozess aus?*
Auslöser des Geschäftsprozesses ist der Kunde, der eine Bestellung aufgeben möchte.
- ✓ *Welche Personen sind beteiligt?*
Außer dem Kunden sind zunächst keine konkreten *Personen* erkennbar, die an dem Geschäftsprozess beteiligt sind. Weitere Akteure sind Abteilungen oder Rollen (s.u.).
- ✓ *Wer ist verantwortlich für die Planung des Prozesses?*
Dieser Aspekt ist hier nicht relevant.

- ✓ *Gibt es außer den beteiligten Personen noch weitere Akteure (z.B. Software-Systeme)?*
 - ✓ Die Abteilung *Auftragsabwicklung* ist für die Zusammenstellung der bestellten Waren verantwortlich.
 - ✓ Der *Lieferservice* übernimmt den Versand der Artikel.
 - ✓ Es gibt eine weitere Rolle, die für die Abwicklung der Bezahlung (Abbuchung vom Konto) zuständig ist. Sie erhält den Namen *Inkasso*.
 - ✓ Eventuell ist es sinnvoll, als weiteren Akteur eine *Datenbank* in das Modell aufzunehmen, wo die Bestellungen der Kunden und ihre persönlichen Daten gespeichert sind.
- ✓ *Wer ist verantwortlich für die Durchführung des Prozesses?*

Für die Durchführung sind die bereits genannten Rollen zuständig.
- ✓ *Wer ist verantwortlich für die Kontrolle des Prozesses?*

Es wird angenommen, dass die beteiligten Rollen selbst die Kontrolle übernehmen.
- ✓ *Wer benutzt das System bzw. den Prozess?*

Das System kann an sich als geschlossen betrachtet werden. Lediglich innerhalb des Systems gibt es einige Beziehungen zwischen den Anwendungsfällen.
- ✓ *Wer benötigt Daten oder Ergebnisse aus dem Prozess?*

Auch hier wird angenommen, dass keinerlei Daten oder Ergebnisse mit externen Instanzen oder Prozessen ausgetauscht werden.

Ermittlung von Aufgaben

Im nächsten Schritt werden die Aufgaben identifiziert, die von den ermittelten Akteuren auszuführen sind.

- ✓ *Welche Ziele sind mit dem Geschäftsprozess verbunden?*
Ziel des Geschäftsprozesses ist die Abwicklung des Kaufs von Waren über das Internet.
- ✓ *Welche Ergebnisse soll das System liefern?*
Das Ergebnis des Geschäftsprozesses ist die Lieferung der vom Kunden bestellten Artikel.
- ✓ *Welche Arbeitsschritte sind dazu erforderlich?*
 - ✓ Bestellung der Artikel
 - ✓ Registrierung des Kunden (falls noch nicht geschehen)
 - ✓ Zusammenstellung der Artikel (interne Abwicklung des Auftrags)
 - ✓ Abwicklung der Zahlung
 - ✓ Auslieferung der Artikel
- ✓ *Welche Ziele sind mit jedem dieser Arbeitsschritte verbunden?*
 - ✓ Bestellung: Erfassung der zu liefernden Artikel und Identifikation des Kunden
 - ✓ Registrierung: Erfassung der Kundendaten, Verifikation der Kundendaten, Erhebung der Daten für die Zahlungsabwicklung
 - ✓ Auftragsabwicklung: Zusammenstellung der bestellten Artikel, Vorbereitung der Auslieferung
 - ✓ Abwicklung der Zahlung: Eingang der Zahlung
 - ✓ Auslieferung: Transport der bestellten Artikel zum Kunden

Ermittlung von Schnittstellen

Anschließend werden die Schnittstellen zu anderen Systemen oder Anwendungsfällen ermittelt.

- ✓ *Zu welchen anderen Anwendungsfällen bestehen Beziehungen?*
Beziehungen zu anderen Anwendungsfällen sind vorhanden (u.a. zur Beschaffung oder zur Lagerhaltung), werden hier aber nicht weiter beachtet.
- ✓ *Benötigt das System Vorleistungen aus einem anderen System?*
Bei bereits früher erfolgter Registrierung werden die Kundendaten aus der Datenbank benötigt.
- ✓ *Gibt das System Daten oder Ergebnisse an andere Systeme weiter?*
Dieser Aspekt ist hier nicht relevant.
- ✓ *Existieren Vorgaben für die Schnittstellen zu bestimmten Akteuren (z.B. Benutzungsoberfläche für menschliche Akteure)?*
Die Benutzungsschnittstelle ist nicht Gegenstand der Analyse des Geschäftsprozesses.

Berücksichtigung von Restriktionen

Es folgt die Analyse eventueller Restriktionen, die für die einzelnen Aufgaben bzw. deren Zusammenwirken gelten.

- ✓ *Welche Abhängigkeiten bestehen zwischen den durchzuführenden Aufgaben?*
 - ✓ Für die Zusammenstellung der gewünschten Artikel und die Bezahlung werden die Informationen aus der Bestellung benötigt.
 - ✓ Abhängigkeiten ergeben sich außerdem aus den Vor- und Nachbedingungen im nächsten Punkt.
- ✓ *Welche Vor- und Nachbedingungen sind bei den einzelnen Aufgaben zu beachten?*
 - ✓ Eine Bestellung ist nicht ohne bestehende Registrierung möglich.
 - ✓ Zur Überprüfung der bestehenden Registrierung muss der Benutzer sich anmelden (Login).

- ✓ Die Artikel können nur dann ausgeliefert werden, wenn sie verfügbar sind.
- ✓ Die Auslieferung erfolgt erst, wenn die Bezahlung gesichert ist (Bonitätsprüfung).
- ✓ *Welche Aufgaben unterliegen organisatorischen, technischen, gesetzlichen oder natürlichen Beschränkungen?*
Beschränkungen dieser Art sind zunächst nicht bekannt. Eventuell ergeben sie sich aber später noch.

Ermittlung von Ereignissen

Nach der Ermittlung der Restriktionen sind die Ereignisse zu analysieren, die in dem Geschäftsprozess auftreten können.

- ✓ *Welche Ereignisse treten standardmäßig in dem Anwendungsfall auf?*
 - ✓ Der Kunde sendet eine Bestellung.
 - ✓ Ein Artikel ist nicht verfügbar.
 - ✓ Die bestellten Artikel sind versandfertig.
 - ✓ Weitere (Detail-) Ereignisse werden später ermittelt.
- ✓ *Welche Auswirkungen haben diese Ereignisse?*
 - ✓ Wenn der Kunde eine Bestellung sendet, ist zu prüfen, ob er registriert ist.
 - ✓ Ist ein Artikel nicht verfügbar, erhält der Kunde eine entsprechende Rückmeldung.
 - ✓ Sobald die Artikel versandfertig sind, wird der geschuldete Betrag abgebucht, die Artikel werden versendet und der Kunde erhält eine Benachrichtigung per E-Mail.
- ✓ *Welche außergewöhnlichen Ereignisse können auftreten?*
 - ✓ Der Kunde ist noch nicht registriert.
 - ✓ Der Kunde widerruft seine Bestellung.

- ✓ Das Konto des Kunden weist keine ausreichende Deckung auf.
- ✓ *Welche Auswirkungen haben diese außergewöhnlichen Ereignisse?*
 - ✓ Ohne Registrierung kann die Bestellung nicht ausgeführt werden.
 - ✓ Bei Widerruf der Bestellung durch den Kunde wird die Ausführung des Auftrags gestoppt.
 - ✓ Bei nicht ausreichender Deckung des Kontos wird der Kunde benachrichtigt und der Auftrag storniert.
- ✓ *Sollen für den Fall solcher Ereignisse Alternativen im Prozessablauf geplant werden?*

Die Alternativen sind bereits im vorherigen Punkt benannt. Weitere Ereignisse und Alternativen ergeben sich u.U. bei der detaillierten Analyse des Geschäftsprozesses.

Strukturierung

Auch die Strukturen innerhalb des Geschäftsprozesses gehören zu einer groben Analyse hinzu. Zu deren Ermittlung dient die Beantwortung der nächsten Fragen aus der Checkliste des achten Kapitels (Zeile mit der Bezeichnung *Strukturierung*). Für die Bearbeitung dieser Fragen ist aber in der Regel, wie auch hier, bereits eine tiefere Kenntnis der internen Abläufe erforderlich. Sie werden daher normalerweise erst in einem späteren Zyklus des Analyseprozesses bearbeitet. Für das relativ stark eingegrenzte Fallbeispiel des Web-Shops sind sie weniger relevant.

Erweiterung

Gleiches gilt für den letzten Fragenkomplex der Checkliste, wo es um potenzielle Erweiterungen des Systems geht. Auch dieser Aspekt soll im Rahmen unseres Fallbeispiels nicht weiter verfolgt werden.

Quintessenz

Aus den mit Hilfe der Checkliste gesammelten Erkenntnissen für die grobe Systemanalyse lässt sich nun das folgende Use-Case-Diagramm für den Geschäftsprozess ableiten.

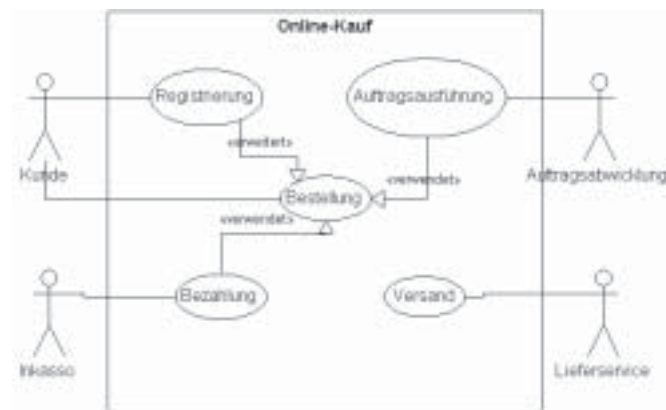


Abbildung 9.1: Geschäftsprozessdiagramm *Online-Kauf*

Dem Diagramm ist zu entnehmen, dass insgesamt fünf Anwendungsfälle unterschieden werden. Dies sind die Registrierung, die Auftragsabwicklung, die Bestellung, die Bezahlung und der Versand.

Mit diesen fünf Anwendungsfällen lässt sich der Geschäftsprozess zunächst grob beschreiben. An der Abwicklung des Prozesses sind die folgenden Rollen beteiligt:

- ✓ Der Kunde gibt seine Bestellung auf und führt eine Registrierung durch. Letztere erfolgt aber nur dann, wenn der Kunde bisher noch nicht in der Datenbank registriert ist. Damit ist die Registrierung eine *Erweiterung* der Bestellung, die nur unter bestimmten Bedingungen auftritt.

- ✓ Die Rolle Auftragsabwicklung ist für die Durchführung des Auftrags zuständig. Im Rahmen der Auftragsausführung werden Informationen aus der Bestellung benötigt. Deshalb handelt es sich zwischen den Anwendungsfällen Bestellung und Auftragsausführung um eine *Include*-Beziehung (verwendet).
- ✓ Die Bezahlung der bestellten Waren wird von der Rolle Inkasso abgewickelt. Auch sie benötigt dazu Informationen aus der Bestellung.
- ✓ Die Rolle Lieferservice ist schließlich für den Versand der bestellten Waren verantwortlich.

Damit ist der Geschäftsprozess im Groben beschrieben. Die Anwendungsfälle, die sich im Verlauf des Prozesses ergeben, sind benannt und die beteiligten Rollen sind identifiziert. Welche internen Abläufe sich daraus ergeben, wird später bei der Entwicklung des dynamischen Modells näher untersucht.

Abgrenzung des relevanten Problemausschnitts

Die Abgrenzung des relevanten Problemausschnitts dient dazu, für das Modell Wichtiges von Unwichtigem zu trennen. Mit Hilfe der verbalen Beschreibung wird im Rahmen eines evolutionären Prozesses der Systemanalyse der Kern der Problemstellung herausgearbeitet und ständig verfeinert.

Als Hauptkriterium sind die Ziele des Geschäftsvorfalles heranzuziehen. Ferner ist zu überlegen, auf welche Elemente in dem Objektmodell verzichtet werden kann, ohne das Abbild der realen Problemstellung zu beeinträchtigen.

Beziehungen zu Elementen außerhalb der betrachteten Problemdomäne, die dennoch eine wichtige Rolle spielen, stellen sich unter Umständen erst im weiteren Verlauf der Analyse heraus. Der Problemausschnitt wird daher immer wieder überarbeitet und angepasst.

Mit Hilfe der Checkliste aus dem achten Kapitel lässt sich der relevante Problemausschnitt eingrenzen.

Zielsetzung

Der erste Punkt bezieht sich auf die Zielsetzung der oben aufgeführten Anwendungsfälle. Sie ist bereits im Rahmen der groben Systemanalyse definiert worden.

Relevanzprüfung

Im nächsten Punkt geht es um die Relevanz der identifizierten Anwendungsfälle für den Geschäftsprozess.

- ✓ *Ist das betrachtete Element für die Ziele und Aufgaben des Anwendungsfalls oder Geschäftsvorfalles von Bedeutung?*
 - ✓ Die Anwendungsfälle Bestellung, Auftragsbearbeitung, Bezahlung und Versand sind zwingend erforderlich, um den Online-Kauf abzuwickeln.
 - ✓ Die Notwendigkeit der Registrierung ergibt sich eher mittelbar, denn ohne die Aufnahme der Kundendaten und deren Überprüfung ist die Bezahlung nicht gewährleistet.
- ✓ *Welche Auswirkungen hat der Verzicht auf die Integration des betrachteten Elements für die Erfüllung der Aufgabe oder des Ziels?*
 - ✓ Bestellung, Auftragsbearbeitung und Bezahlung gehören zum Kern des Geschäftsprozesses. Ohne diese Elemente ist der Geschäftsprozess nicht durchführbar.
 - ✓ Der Versand ist zwar ebenfalls erforderlich, um die vom Kunden gewünschte Leistung zu erbringen und somit das Ziel des Prozesses zu realisieren. Andererseits kann die Lieferung der Artikel aber auch als ein nachgelagerter Prozess angesehen werden. Ändert man den Blickwinkel auf die Problemstellung in dieser Weise, dann könnte der Anwendungsfall Versand in unserem Objektmodell vernachlässigt werden.

- ✓ Auch die Registrierung könnte als externer Prozess angesehen werden. Der Geschäftsprozess Online-Kauf wäre dann über eine Schnittstelle mit dem Prozess Registrierung verbunden.

Die Bearbeitung des Punkts Relevanzprüfung verdeutlicht, wie wichtig die Sichtweise auf die Problemdomäne für die Gestalt des Objektmodells ist. Je nachdem, welche Elemente als relevant erachtet werden, verändert sich der Aufbau des Objektmodells schon in dieser noch recht groben Analyse.

Ermittlung von Alternativen

Bei der Ermittlung der Alternativen geht es darum, andere Ansätze für die Aufgabestruktur zu finden, die der bestehenden Option möglicherweise überlegen sind.

- ✓ *Welche Rollen (Stellen) sind an der Aufgabe beteiligt?*
Diese Frage wurde bereits im Rahmen der groben Systemanalyse geklärt.
- ✓ *Welche anderen Rollen sind von der Ausführung der Aufgaben betroffen?*
Auch diese Frage wurde oben bereits geklärt.
- ✓ *Kann die Erfüllung von Aufgaben durch eine Umgestaltung in aufbau- oder ablauforganisatorischer Hinsicht optimiert werden?*
 - ✓ Die Ablauforganisation des Geschäftsprozesses ist weitgehend vorgegeben. Welche Aufgaben in welcher Reihenfolge durchzuführen sind, ist – zumindest in dieser Phase einer relativ groben Betrachtung – eindeutig bestimmt.
 - ✓ Die Aufbauorganisation könnte überdacht werden. Dazu würden einzelne Teilaufgaben aus dem Aufgabenbereich eines Akteurs herausgelöst und einem anderen Akteur übertragen. Auch hier ist es jedoch sinnvoll, die Aufgaben zunächst im weiteren Verlauf der Analyse einer detaillierteren Untersuchung zu unterziehen.

- ✓ Können komplexe Wechselwirkungen durch die Einbeziehung weiterer Systeme (Systemvergrößerung) internalisiert werden?
Komplexe Wechselwirkungen sind nach dem derzeitigen Kenntnisstand nicht vorhanden.
- ✓ Können komplexe Wechselwirkungen durch eine reduzierte System-sicht (Systemverkleinerung) vernachlässigt werden?
Hier gilt ebenfalls: Wechselwirkungen sind nicht vorhanden.

Der Problembereich ist damit vorläufig abgegrenzt. Die oben genannten Alternativen, die Registrierung oder den Versand auszugliedern, werden hier nicht weiter verfolgt. Zur Vereinfachung wird das System als weitgehend geschlossen angesehen. Falls sich später herausstellen sollte, dass Beziehungen zu anderen Systemen bzw. Prozessen dringend benötigt werden, ist das Modell entsprechend zu ergänzen.

Abgrenzung von Teilsystemen (Paketen)

Die Bildung von Paketen schafft überschaubare Teilsysteme, die sich gegenseitig ergänzen und in einem Gesamtsystem zusammenarbeiten. Die bisherigen Ergebnisse der Beschreibung des Geschäftsprozesses und der Abgrenzung des relevanten Problemausschnitts legen eine Gliederung des Systems Web-Shop in Pakete nahe.

- ✓ Das erste Paket fasst alle Elemente der Problemdomäne zusammen, die mit der eigentlichen *Bestellung* zusammenhängen. Hierzu zählen im Wesentlichen alle Aktivitäten, in die der Kunde unmittelbar eingebunden ist.
- ✓ Ein zweites Paket bilden alle Elemente, die sich auf die *Abwicklung* eines Auftrags durch die Unternehmung beziehen. Dazu zählen etwa die Abteilungen bzw. Rollen, die an der Auftragsbearbeitung beteiligt sind. Aber auch unternehmensinterne Abläufe (Teilprozesse) gehören in dieses Paket.

Daraus ergibt sich eine prozessorientierte Gliederung der Pakete, die sich bei der Modellierung von Geschäftsprozessen meist als die am besten geeignete Alternative erweist. Die folgende Abbildung zeigt das entsprechende Paketdiagramm.

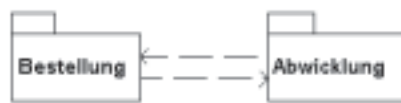


Abbildung 9.2: Pakete des Geschäftsprozesses

Zwischen den beiden Paketen besteht, wie an den beiden gestrichelten Pfeilen zu erkennen ist, eine gegenseitige Abhängigkeit. Änderungen in den Elementen des einen Pakets können Änderungen in dem anderen Paket zur Folge haben.



Die Abhängigkeitsbeziehung zwischen zwei Paketen bedeutet, dass Änderungen in dem unabhängigen Paket *möglicherweise* zu Änderungen in dem abhängigen Paket führen; sie sind also nicht zwangsläufig.

Anhand der Checkliste zu den Teilsystemen im achten Kapitel kann die hier vorgenommene Paketbildung überprüft werden.

Ermittlung von Gliederungskriterien

Zunächst wird nach Gliederungskriterien gesucht.

- ✓ *Ist das betrachtete System in der vorliegenden Form überhaupt teilbar?*
Die Teilbarkeit des Systems ist durchaus gegeben. Der Bereich der Bestellung lässt sich eindeutig von den Abläufen bei der Abwicklung des Auftrags trennen.

- ✓ *Ergeben sich bereits aus der Aufbau- oder Ablaufstruktur des Geschäftsprozesses oder Anwendungsfalls Kriterien für die Paketbildung?*
Die Abläufe legen die gewählte Paketstruktur nahe. Die Bestellung kann ebenso wie die Abwicklung als eigenständiger Bestandteil des Geschäftsprozesses angesehen werden.
- ✓ *Erlaubt die Gesamtaufgabe eine inhaltliche, prozessorientierte, lokale oder funktionale Gliederung?*
Diese Frage wurde bereits beantwortet. Eine prozessorientierte Gliederung liegt nahe.

Definition von Paketen

Im nächsten Schritt werden die Pakete definiert.

- ✓ *Handelt es sich bei dem betrachteten Element um ein isolierbares Subsystem?*
Die Bestellung und die Auftragsabwicklung lassen sich beide als isolierbare Subsysteme des Online-Kaufs betrachten.
- ✓ *Bildet das betrachtete Paket einen Themenbereich ab, der weitgehend für sich alleine gesehen werden kann?*
Auch diese Frage ist zu bejahen. Bestellung und Abwicklung sind aus der Sicht des Auftrags inhaltlich unterscheidbare Themen.
- ✓ *Liegen die Vererbungsstrukturen der beteiligten Klassen weitgehend innerhalb des Pakets?*
In diesem Stadium der Analyse – wir befinden uns immer noch in der groben Systemanalyse – sind die Vererbungsstrukturen der beteiligten Klassen noch nicht bekannt. Das bedeutet für das Objektmodell, dass sich die hier gewählte Paketstruktur u.U. später im Rahmen der Klassenbildung als ungeeignet erweist. In einem evolutionären Entwicklungsprozess werden die Strukturen ständig verfeinert und an neue Erkenntnisse angepasst. Dies kann in einigen Fällen auch dazu führen, dass ein zu Beginn als sinnvoll erachteter Ansatz später komplett über

Bord geworfen wird. Solange jedoch keine gegenteiligen Anzeichen erkennbar sind, kann an der derzeitigen Struktur festgehalten werden.

- ✓ *Können umfangreiche Subsysteme in weitere kleinere Pakete zerlegt werden?*

Die beiden Teilsysteme Bestellung und Abwicklung sind nach dem bisherigen Informationsstand nicht allzu umfangreich. Eine weitere Zerlegung ist daher nicht notwendig.

- ✓ *Können durch die Umgestaltung von Aufgaben, Prozessen und Zuständigkeiten Pakete gebildet werden, ohne die Ziele zu gefährden?*

Diese Frage der Checkliste zielt auf Situationen ab, in denen es schwierig ist, Pakete zu bilden. In solchen Fällen ist zu überlegen, ob eine Umgestaltung des Prozesses zu klareren Strukturen führt. In unserem Fallbeispiel ist dies aber nicht erforderlich.

Berücksichtigung von Interdependenzen

Nachdem die Pakete gebildet wurden, müssen nun Abhängigkeiten zwischen den Paketen untersucht werden.

- ✓ *Welche Beziehungen und Abhängigkeiten besitzt das betrachtete Element zu anderen Elementen innerhalb des Systems?*

Zwischen den beiden Paketen Bestellung und Abwicklung besteht eine gegenseitige Abhängigkeit. Änderungen im Zusammenhang mit der Bestellung führen möglicherweise auch zu Änderungen in der Auftragsabwicklung, da Letztere auf Informationen aus Ersterer zugreift. Umgekehrt hat aber auch die Umgestaltung der Auftragsabwicklung u.U. Folgen für die Art und Weise, wie eine Bestellung durchgeführt wird bzw. welche Informationen dort zu erheben sind.

- ✓ *Sind von der Integration des betrachteten Elements in ein (Teil-) System weitere Elemente in anderen (Teil-) Systemen betroffen, die ihrerseits wiederum Abhängigkeiten aufweisen?*

Diese Frage ist erst zu beantworten, wenn bei der detaillierteren Analyse Klassen gebildet und den Paketen zugeordnet werden.

Definition von Schnittstellen zwischen den Paketen

Die Schnittstellen zwischen den Paketen ergeben sich u.a. durch Beziehungen zwischen Klassen unterschiedlicher Pakete. Sie können daher erst nach der Identifikation der Klassen und ihrer Beziehungen definiert werden. Gleichwohl handelt es sich um einen Vorgang innerhalb des objektorientierten Analyseprozesses, der auf relativ abstrakter Ebene (also in der groben Systemanalyse) anzusiedeln ist.



An dieser Stelle zeigt sich erneut, dass die objektorientierte Analyse nicht in eine lineare Vorgehensweise eingebettet werden kann. Damit scheiden für die Objektorientierung auch die in der prozeduralen Programmierung weit verbreiteten Ansätze des Top-Down- und des Bottom-up-Vorgehens aus. Hier ist vielmehr ein zyklischer Prozess angezeigt, der beide Ansätze zu einem Gegenstromverfahren mit zyklischem Charakter verbindet. Die ständige Anpassung der erzielten (Teil-)Ergebnisse ist unabdingbar.

Die Bearbeitung der Checkliste für die Schnittstellen zwischen den Paketen ist an dieser Stelle wenig sinnvoll. Die Beantwortung der dort aufgeführten Fragen erfordert bereits nähere Informationen über die erst noch zu bildenden Klassen. Die Checkliste sollte daher erst in einem späteren Zyklus des Analyseprozesses bearbeitet werden.

Dennoch ergibt es durchaus einen Sinn, die Definition der Schnittstellen zwischen den Paketen (grundsätzlich) an diese Stelle des Vorgehens zu setzen, wenn man sich den evolutionären Charakter der objektorientierten Analyse vergegenwärtigt. Im Laufe späterer Zyklen ergibt sich ein ständiger Informationszuwachs, was die Art der Beziehungen zwischen Klassen und Paketen sowie deren innere Struktur angeht. Die Anpassung der Pakete an den neuen Informationsstand gehört dann genau an diese Stelle: Vor die Bearbeitung des statischen Objektmodells.

Statisches Modell

Die Entwicklung des statischen Modells hängt eng mit der Bildung von Paketen mit ihren Schnittstellen zusammen. Beziehungen zwischen Klassen unterschiedlicher Pakete führen zu Abhängigkeiten zwischen Paketen. Im Rahmen der Definition der Klassen und ihrer Beziehungen stellt sich zudem heraus, ob die bisherige Gliederung der Pakete sinnvoll ist oder eventuell angepasst werden muss.

Die Entwicklung des statischen Modells gliedert sich in die Aufgaben

- ✓ Identifikation von Klassen, Attributen und Methoden,
- ✓ Beschreibung von Klassen und Attributen, Ermittlung von Vererbungsstrukturen und
- ✓ Identifikation von Assoziationen.

Hier tritt einmal mehr der evolutionäre Charakter der objektorientierten Analyse zu Tage. Das Modell unterliegt einer ständigen Anpassung und Erweiterung im Rahmen des Analyseprozesses. Die gliederungstechnische Differenzierung der folgenden Abschnitte ist daher keineswegs als lineare Abfolge von Analysephasen zu betrachten. Sie bildet lediglich eine Struktur der bei der Entwicklung des statischen Modells anfallenden Aufgaben.

Identifikation von Elementen und Strukturen

In diesem Fallbeispiel dient allein die verbale Beschreibung des Geschäftsprozesses im vorangegangenen Abschnitt als Informationsgrundlage für das statische Modell. Bei einer realen Problemstellung werden darüber hinaus die im achten Kapitel näher erläuterten Quellen ausgewertet:

- ✓ Formulare, Listen, Verträge und andere Dokumente
- ✓ Dokumentationen von Geschäftsprozessen, Projekten und Software

- ✓ Datenmodelle (Datenbanken), Bildschirmmasken etc.
- ✓ Organigramme
- ✓ Stellenbeschreibungen
- ✓ Interviews mit Akteuren
- ✓ Dokumentation von Modellen vergleichbarer Problemdomänen (andere Abteilungen, Fachliteratur, Erfahrungsberichte)

Die Auswertung erfolgt mit Hilfe der Checkliste zur Klassenbildung. Im Einzelnen sind damit folgende Aktivitäten verbunden:

- ✓ Identifikation von Klassen
- ✓ Identifikation von Attributen
- ✓ Identifikation von Operationen
- ✓ Validierung

Identifikation von Klassen

Bei der Identifikation von Klassen geht es darum, Objekte mit gleichen oder ähnlichen Eigenschaften zu finden bzw. typische Muster innerhalb des Geschäftsprozesses zu ermitteln.

- ✓ *Welche Klassen ergeben sich aus der Beschreibung der Geschäftsprozesse bzw. Anwendungsfälle?*
 - ✓ Warenkorb
 - ✓ Auftrag
- ✓ *Welche Akteure (Personen, Abteilungen, Organisationen) sind an dem Geschäftsvorfall aktiv oder passiv beteiligt?*
 - ✓ Kunde
 - ✓ Inkasso
 - ✓ Auftragsabwicklung
 - ✓ Lieferservice

- ✓ *Welche Rollen übernehmen Personen in dem Geschäftsvorfall?*
Die Rollen entsprechen den Akteuren aus dem vorhergehenden Punkt.
- ✓ *Welche Gegenstände werden für die zu beschreibenden Aufgabe benötigt?*
 - ✓ Bücher
 - ✓ Musik-CD
 - ✓ Artikel
- ✓ *Welche Hardware und Software wird eingesetzt?*
 - ✓ Datenbank mit Kundendaten
 - ✓ Datenbank mit Artikeldaten
 - ✓ Software für den Web-Auftritt inklusive Benutzeroberfläche
- ✓ *Welche Informationen werden von den Akteuren verarbeitet?*
 - ✓ Kunde: Titel, Autoren und Preise der Bücher; Titel, Autoren und einzelne Stücke der Musik-CDs; Warenkorb, eigene persönliche Daten; Daten der Bestellung
 - ✓ Auftragsabwicklung: Artikelnummern der Bücher und Musik-CDs; persönliche Daten des Kunden; Daten der Bestellung
 - ✓ Inkasso: Bankverbindung des Kunden, Daten der Bestellung; persönliche Daten des Kunden (insbesondere Rechnungsanschrift und Bankverbindung)
 - ✓ Lieferservice: Persönliche Daten des Kunden (insbesondere Lieferanschrift)
- ✓ *An welchen Orten werden die mit den Aufgaben verbundenen Aktivitäten durchgeführt?*
 - ✓ innerhalb der Unternehmung
 - ✓ am Aufenthaltsort des Kunden (z.B. zu Hause)

- ✓ Können Aktionen zu Funktionsbereichen zusammengefasst werden?
Die Anwendungsfälle Registrierung, Bestellung, Auftragsausführung, Bezahlung und Versand ergeben jeweils eigene Funktionsbereiche.
- ✓ Lassen sich strukturelle Ähnlichkeiten zwischen einzelnen Elementen feststellen?
Strukturelle Ähnlichkeiten sind vorerst nicht erkennbar.
- ✓ Welche (internen und externen) Ereignisse treten in dem Geschäftsvorfall auf?
 - ✓ Der Kunde sendet eine Bestellung.
 - ✓ Ein Artikel ist nicht verfügbar.
 - ✓ Die bestellten Artikel sind versandfertig.
 - ✓ Der Kunde ist noch nicht registriert.
 - ✓ Der Kunde widerruft seine Bestellung.
 - ✓ Das Konto des Kunden weist keine ausreichende Deckung auf.

Als Ergebnis der Bearbeitung dieser Fragen der Checkliste lassen sich folgende potenziellen Klassen festhalten:

- ✓ Warenkorb
- ✓ Auftrag
- ✓ Kunde
- ✓ Inkasso
- ✓ Auftragsabwicklung
- ✓ Lieferservice
- ✓ Buch
- ✓ Musik-CD

- ✓ Artikel
- ✓ Datenbank (Kunden und Artikel)

Die weitere Bearbeitung der Checkliste wird zeigen, welche dieser Klassen tatsächlich benötigt werden und welche weiteren Klassen u.U. hinzu kommen.

Identifikation von Attributen

Im nächsten Schritt geht es darum, Attribute, also zustandsbeschreibende Eigenschaften, der identifizierten Klassen zu finden.

- ✓ *Wodurch ist die betrachtete Klasse eindeutig gekennzeichnet?*
 - ✓ Warenkorb: ID-Nummer des Warenkorbs
 - ✓ Auftrag: Auftragsnummer
 - ✓ Kunde: Kundennummer
 - ✓ Inkasso: Personalnummer
 - ✓ Auftragsabwicklung: Personalnummer
 - ✓ Lieferservice: Personalnummer
 - ✓ Buch: Artikelnummer
 - ✓ Musik-CD: Artikelnummer
 - ✓ Artikel: Artikelnummer
- ✓ *Welche Daten beschreiben die betrachtete Klasse?*
 - ✓ Warenkorb: Kunde, Artikelliste, Gesamtpreis
 - ✓ Auftrag: Kunde, Warenkorb, Status, Frist
 - ✓ Kunde: Name, Lieferanschrift, Rechnungsanschrift, Bankverbindung, Registrierung
 - ✓ Inkasso: Name des zuständigen Mitarbeiters (Inkasso selbst ist eine Rolle), zu bearbeitende Aufträge

- ✓ Auftragsabwicklung: Name des zuständigen Mitarbeiters (Auftragsabwicklung selbst ist eine Rolle), abzuwickelnde Aufträge
- ✓ Lieferservice: Name des zuständigen Mitarbeiters (Lieferservice selbst ist eine Rolle), abzuwickelnde Aufträge
- ✓ Buch: Titel, Autoren, Inhaltsverzeichnis, Klappentext, Lagerort, Verfügbarkeit, Preis
- ✓ Musik-CD: Titel, Autoren, Liste der einzelnen Stücke, Lagerort, Verfügbarkeit, Preis
- ✓ Artikel: Lagerort, Verfügbarkeit, Preis
- ✓ *Ergeben sich Attribute aus der Vererbungsstruktur?*
 - ✓ Bücher und Musik-CDs sind beides Artikel. Da diese drei Klassen ähnliche Eigenschaften aufweisen, bietet sich hier eine Vererbungsstruktur mit `Artikel` als Oberklasse sowie `Buch` und `Musik-CD` als Unterklassen an.
 - ✓ Auch die auf Seiten der Unternehmung beteiligten Rollen haben ähnliche Eigenschaften. Eventuell sollte daher auch hier eine Vererbungsstruktur eingesetzt werden.
- ✓ *Was unterscheidet die betrachtete von den übrigen identifizierten Klassen?*

Die Unterschiede bezüglich der Attribute ergeben sich aus den Antworten auf die beiden vorhergehenden Fragen.
- ✓ *Welche Zustände können die Exemplare einer Klasse annehmen?*

Die Zustände ergeben sich durch die konkreten Attributwerte der Objekte. Wichtige Zustände sind später u.a. der Status eines Auftrags, die (nicht) vorliegende Registrierung eines Kunden oder der Zahlungseingang.
- ✓ *Ist das betrachtete Attribut Gegenstand von Operationen?*

Hier sind die einzelnen Attribute zu prüfen. Dieser Aspekt wird später im Zusammenhang mit dem dynamischen Modell näher untersucht.

- ✓ *Handelt es sich um ein Klassenattribut oder besitzt jedes Objekt individuelle Werte für dieses Attribut?*
Unter den bisher identifizierten statischen Eigenschaften ist noch kein Klassenattribut zu erkennen.
- ✓ *Ist das Attribut so komplex, dass sich eine nähere Beschreibung in einer eigenen Klasse anbietet?*
Einige Attribute finden sich in der obigen Auflistung gleichzeitig als identifizierte Klassen wieder. In diesen Fällen ist anzunehmen, dass es sich weniger um Attribute als vielmehr um Klassenbeziehungen handelt. Dies gilt für folgende Elemente:
 - ✓ Registrierung
 - ✓ Warenkorb
 - ✓ Kunde
- ✓ *Verlangen die zugreifenden Operationen die Zusicherung eines bestimmten Ergebnistyps?*
Diese Frage wird im Rahmen der Definition von Operationen beantwortet. In einem evolutionären Analyseprozess wird dieser Aspekt erst in einem fortgeschrittenen Stadium behandelt.
- ✓ *Welche Informationen muss die betrachtete Klasse für andere Klassen zur Verfügung stellen?*
Die Informationen ergeben sich aus den bisherigen Antworten.

Identifikation von Operationen

Nach der Ermittlung der statischen Eigenschaften folgt nun die Identifikation der dynamischen Eigenschaften, die durch Operationen repräsentiert werden.

- ✓ *Welche (Teil-) Prozesse wurden in der Beschreibung der Geschäftsprozesse bzw. Anwendungsfälle ermittelt?*
 - ✓ Registrierung
 - ✓ Bestellung

- ✓ Auftragsausführung
- ✓ Bezahlung
- ✓ Versand
- ✓ *Welche Verhaltensweisen zeigen die beteiligten Akteure zur Bewältigung ihrer Aufgaben?*
 - ✓ Kunde: Artikel auswählen, Bestellung abgeben, Auftrag stornieren, registrieren
 - ✓ Auftragsabwicklung: Bestelldaten übernehmen, Artikel zusammenstellen, Gesamtpreis berechnen, Status melden
 - ✓ Lieferservice: Lieferadresse ermitteln, liefern, Status melden
 - ✓ Inkasso: Rechnungsadresse ermitteln, Bonität prüfen, abbuchen, mahnen
- ✓ *Welche Aktionen werden von den beteiligten Akteuren durchgeführt?*

Die Aktionen sind hier weitgehend identisch mit den Verhaltensweisen der Akteure. In manchen Problemstellungen kann es jedoch sinnvoll sein, hier noch einmal zu differenzieren, um leichter Operationen zu finden bzw. diese klarer voneinander trennen zu können.
- ✓ *Welche Prozesse laufen in der eingesetzten Hardware und Software ab?*

Bei der Bestellung eines Warenkorbs wird der Status der Registrierung geprüft und die Ausführung des Auftrags angestoßen.
- ✓ *Wer gibt Informationen weiter?*
 - ✓ Kunde
 - ✓ Auftragsabwicklung
 - ✓ Lieferservice
 - ✓ Inkasso

- ✓ *Wer erhält Informationen?*
 - ✓ Kunde
 - ✓ Auftragsabwicklung
 - ✓ Lieferservice
 - ✓ Inkasso
- ✓ *Über welche Kanäle werden Informationen weitergegeben?*
 - ✓ Webseite
 - ✓ E-Mail
 - ✓ Persönliche Benachrichtigung (schriftlich, telefonisch)
 - ✓ Software-Schnittstellen
- ✓ *Welche anderen Klassen (bzw. Objekte) benötigen Informationen?*
 - ✓ Registrierung
 - ✓ Auftragsausführung
 - ✓ Bestellung
 - ✓ Bezahlung
 - ✓ Versand
 - ✓ Warenkorb (Status der Registrierung)
- ✓ *In welcher Form benötigen andere Klassen oder Objekte diese Informationen?*

Dieser Aspekt wird später bei der konkreten Beschreibung der Operationen untersucht.
- ✓ *Durch welche Ereignisse oder Aktionen ändern die Exemplare einer Klasse ihren Zustand?*

Die konkreten Ereignisse, die Zustandsänderungen auslösen, werden bei der Entwicklung des dynamischen Modells analysiert.

Bis hierher ergeben sich aus den Checklisten die folgenden Klassen.

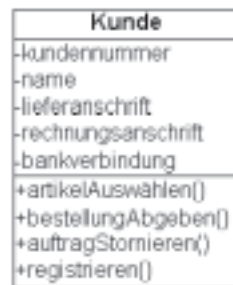


Abbildung 9.3: Die Klasse Kunde

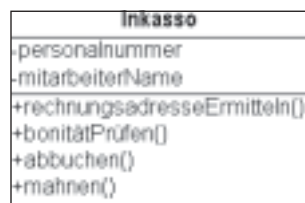
Für die Klasse Kunde wurden die Attribute `kundennummer`, `name`, `lieferanschrift`, `rechnungsanschrift` und `bankverbindung` ermittelt. Als Operationen wurden `artikelAuswählen`, `bestellungAbgeben`, `auftragStornieren` und `registrieren` identifiziert.



Abbildung 9.4: Die Klasse Warenkorb

Die Klasse Warenkorb besitzt die Attribute `id-Nummer` und `gesamtpreis`. Die Bestellung eines Warenkorbs durch den Kunden wird über die gleichnamige Operation ausgeführt (siehe Abbildung 9.5).

Aufträge haben die statischen Eigenschaften `auftragsnummer`, `status` und `frist`. Die Operation `ausführen` führt zur internen Abwicklung des Auftrags (siehe Abbildung 9.6).

**Abbildung 9.5:** Die Klasse Auftrag**Abbildung 9.6:** Die Klasse Inkasso

Für die Klasse Inkasso sind die Attribute `personalnummer` und `mitarbeiterName` zustandsbeschreibend. Das Verhalten der Objekte dieser Klasse äußert sich in den Operationen `rechnungsadresseErmitteln`, `bonitätPrüfen`, `abbuchen` und `mahnen`.

**Abbildung 9.7:** Die Klasse Auftragsabwicklung

Der Zustand der Klasse Auftragsabwicklung wird durch die gleichen Attribute wie bei der Klasse Inkasso beschrieben: personalnummer und mitarbeiterName. Als Operationen wurden bestelldatenübernehmen, artikelZusammenstellen, gesamtpreisBerechnen und statusMelden identifiziert.



Abbildung 9.8: Die Klasse Lieferservice

Das Klassendiagramm für den Lieferservice zeigt die Attribute `personalnummer` und `mitarbeiterName` sowie die Operationen `lieferadresseErmitteln`, `liefern` und `statusMelden`.

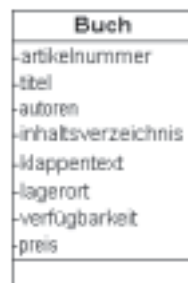


Abbildung 9.9: Die Klasse Buch

Ein Buch wird beschrieben durch die Attribute `artikelnummer`, `titel`, `autoren`, `inhaltsverzeichnis`, `klappentext`, `lagerort`, `verfügbarkeit` und `preis`. Operationen sind vorerst nicht vorhanden.



Abbildung 9.10: Die Klasse Musik-CD

Die Klasse Musik-CD ähnelt der Klasse Buch in ihren Eigenschaften sehr. Auch hier gibt es vorerst keine Operationen. Statt des Inhaltsverzeichnisses und des Klappentextes hat diese Klasse das Attribut `listeDerStücke`, in dem die einzelnen Musikstücke aufgeführt sind.



Abbildung 9.11: Die Klasse Artikel

Wie bereits bei der Bearbeitung der Checklisten angesprochen, bietet es sich an, die Klassen Buch und Musik-CD zu der Klasse Artikel zu generalisieren, da die beiden Klassen sehr ähnliche Eigenschaften haben. Zudem ergibt sich auch eine semantische Übereinstimmung, da es sich bei den Objekten beider Klassen um spezielle Artikel handelt.

Es fehlt nun noch die Klasse Registrierung. Ein Kunde muss zunächst registriert sein, bevor er eine Bestellung in Auftrag geben kann. Bei der Registrierung wird dem Kunde ein Benutzername und ein Passwort für seine zukünftigen Bestellungen zugewiesen. Außerdem muss der Status der Registrierung vor der Ausführung eines Auftrags geprüft werden (Operation `getStatus()`). Das zugehörige Klassendiagramm auf dem derzeitigen Informationsstand zeigt Abbildung 9.12.

Damit sind die bisher ermittelten Klassen mit ihren statischen und dynamischen Eigenschaften ermittelt. Diese müssen nun noch den beiden Paketen zugeordnet werden, die bereits weiter oben definiert wurden.

Dem Paket Bestellung werden die Klassen Kunde, Warenkorb, Registrierung, Buch, Musik-CD und Artikel zugeordnet. Diese drei Klassen sind unmittelbar an dem Bestellungsprozess beteiligt.



Abbildung 9.12: Die Klasse Registrierung

Das zweite Paket mit dem Namen Abwicklung bilden die Klassen Auftrag, Inkasso, Auftragsabwicklung und Lieferservice. Die folgende Abbildung zeigt die beiden Pakete im Überblick.

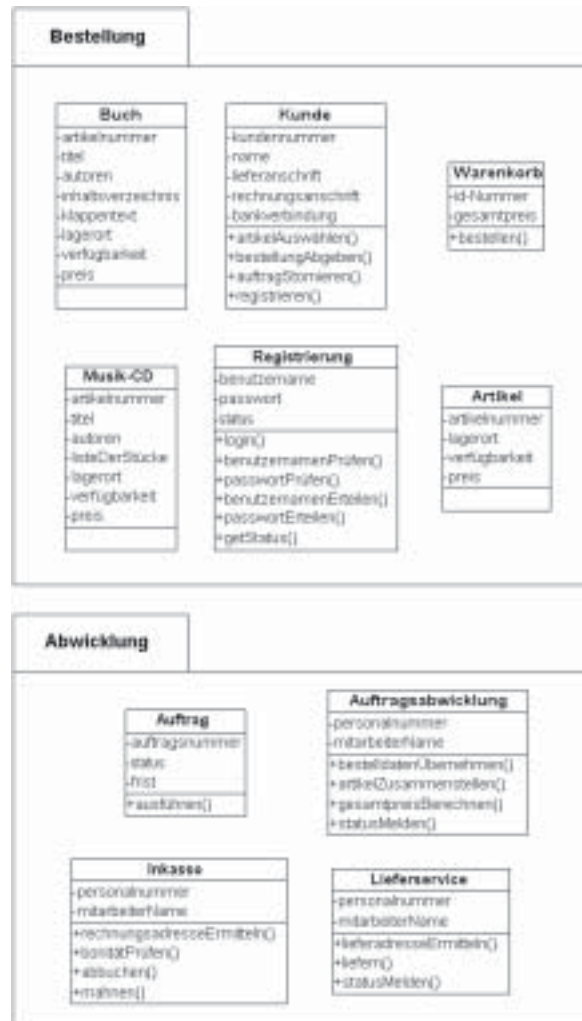


Abbildung 9.13: Die Pakete Bestellung und Abwicklung

Validierung

Die Validierung kann an dieser Stelle noch nicht vollständig durchgeführt werden, da sich einige der zu prüfenden Aspekte erst aus späteren Aktivitäten des Analyseprozesses ergeben (z.B. die Vererbungsstruktur). Sie wird daher zunächst zurückgestellt.

Beschreibung von Klassen und Attributen

Nachdem die beteiligten Klassen mit ihren Attributen und Operationen identifiziert wurden, sind diese im nächsten Schritt in einer verbalen Beschreibung zu dokumentieren. Die folgende Tabelle enthält beispielhaft die Beschreibungen einiger der oben genannten Klassen mit ihren Attributen und Operationen.

Klasse	Paket	Attribute	Operationen	Beschreibung
Buch (Artikel)	Bestel- lung	Geerbte Attribute: artikelnummer titel autoren lagerort verfügbarkeit preis Eigene Attribute: inhaltsver- zeichnis klappentext	Keine	Die Objekte der Klasse Buch repräsentieren die Bücher, die in dem Web-Shop angeboten werden. Jedes Objekt beschreibt ein Buch mit allen seinen Eigenschaften.
Registrierung	Bestel- lung	benutzername passwort status	Login benutzernamen- Prüfen passwortPrüfen benutzernamen- Erteilen passwortErteilen getStatus	Registrierung beschreibt sowohl den Vorgang der Registrierung, die ein Benutzer (Kunde) vornimmt, als

Klasse	Paket	Attribute	Operationen	Beschreibung
				auch die Eigenschaften, die dabei generiert werden. Jedes Objekt der Klasse gehört zu genau einem Kunden.
Lieferservice	Abwicklung	personalnummer mitarbeiter-Name	lieferadresse-Ermitteln liefern statusMelden	Lieferservice ist ein Akteur des Anwendungsfalls Versand. Objekte dieser Klasse wickeln den Versand der bestellten Artikel ab.

Tabelle 9.2: Beispiele für Klassenbeschreibungen

Wenn es nicht, wie in diesem Fallbeispiel, um die Modellierung von Geschäftsprozessen geht, sondern die UML im Rahmen des Software Engineering eingesetzt wird, werden über die hier aufgeführten Inhalte hinaus noch weitere Informationen in die Klassenbeschreibung aufgenommen. Dazu gehören u.a. die Datentypen und die Sichtbarkeit von Attributen und Operationen.

Der erläuternde Text in der letzten Spalte der obigen Tabelle ist in praktischen Problemstellungen meist wesentlich ausführlicher. Dies gilt insbesondere dann, wenn das Modell eine Vielzahl von Klassen aufweist, die darüber hinaus auch noch weniger eindeutige Abgrenzungen besitzen, als dies hier der Fall ist. Für unser Fallbeispiel soll die Kurzform zur Anschauung genügen.

Ermittlung von Vererbungsstrukturen

Vererbungsstrukturen können auf zwei Arten aus bestehenden Klassen abgeleitet werden. Bei der Generalisierung werden gleiche oder ähnliche Eigenschaften bereits identifizierter Klassen in Oberklassen zusammengefasst. Umgekehrt werden bei der Spezialisierung besondere Ausprägungen einer Klasse in Unterklassen konkretisiert.

Identifikation

Die Checkliste aus Kapitel 8 schlägt für die Ermittlung von Klassenstrukturen den folgenden Fragenkatalog vor.

- ✓ *Gibt es Gemeinsamkeiten zwischen bereits identifizierten Klassen?*
Die Klassen Buch und Musik-CD besitzen starke Ähnlichkeiten bezüglich ihrer Attribute. Beide können als Artikel durch den Kunden bestellt werden.
- ✓ *Besitzen bereits identifizierte Klassen spezielle Ausprägungen, die sich ihrerseits durch zusätzliche Eigenschaften auszeichnen?*
Diese Frage zielt auf die umgekehrte Vorgehensweise der Spezialisierung ab. Analog zur vorherigen Aussage besitzt die Klasse Artikel die speziellen Ausprägungen Buch und Musik-CD.
- ✓ *Sind in der Problemdomäne hierarchische Strukturen erkennbar?*
Hierarchische Strukturen sind nicht erkennbar.
- ✓ *Bestehen Ähnlichkeiten zwischen den Elementen der hierarchischen Struktur?*
Hierarchische Strukturen sind nicht erkennbar.
- ✓ *Lassen sich Beziehungen der Form »A ist ein B« bilden?*
Denkbar wäre es, die Akteure auf Seiten der Unternehmung in einer Klasse Mitarbeiter zu generalisieren. Für die Klasse Inkasso würde dann beispielsweise die Aussage gelten: »Ein Objekt der Klasse Inkasso ist ein Mitarbeiter.« Allerdings besteht dann (zumindest in Übereinstimmung mit dem Objektmodell) nicht mehr die Möglichkeit, die zugehörigen Aufgaben etwa zu

automatisieren oder durch mehrere Mitarbeiter durchführen zu lassen.

Die folgende Abbildung zeigt die einzige Vererbungshierarchie in dem hier zu entwickelnden Objektmodell.

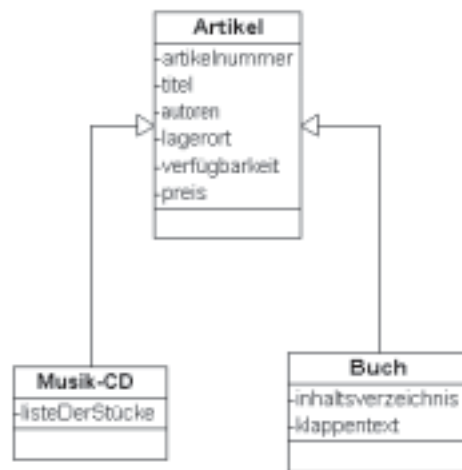


Abbildung 9.14: Die Klasse **Artikel** mit ihren speziellen Ausprägungen

Die Klasse **Artikel** ist in der Abbildung als Oberklasse zu den Klassen **Musik-CD** und **Buch** installiert.

Alle drei Klassen hatten bereits bisher die Attribute `artikelnummer`, `lagerort`, `verfügbarkeit` und `preis` gemeinsam. Diese werden nun aus den eigenen Zustandsbeschreibungen der beiden Unterklassen gestrichen. Durch die Vererbungsbeziehung erben **Musik-CD** und **Buch** diese Attribute nun von der Klasse **Artikel**.

Die weiteren Attribute `titel` und `autoren` existierten bisher nicht in der Klasse `Artikel`, sondern nur in den beiden Unterklassen. Auch sie wurden als gemeinsame Eigenschaften in die Oberklasse überführt. In dem stark eingegrenzten Fallbeispiel ist dies allerdings nur möglich, da außer Büchern und Musik-CDs keine weiteren Artikel vertrieben werden und auch keine Erweiterungen vorgesehen sind. Sobald einer neuer Artikel hinzukommt, der keinen Titel und/oder Autor besitzt (z.B. Elektrogeräte), kann dieser nicht mehr in die Vererbungsstruktur eingebaut werden.



Berücksichtigen Sie gerade beim Entwurf von Vererbungsstrukturen stets die Erweiterbarkeit Ihres Modells. Verzichten Sie vorsichtshalber auf eine zu starke Generalisierung von Eigenschaften, um die Flexibilität Ihres Modells zu erhöhen. Dies gilt in besonderem Maße im Rahmen des objektorientierten Software Engineering.

Validierung

Zur Überprüfung der ermittelten Vererbungsstruktur dienen die folgenden Fragen aus der Checkliste des achten Kapitels.

- ✓ *Werden alle vererbten Attribute zur Beschreibung des Zustands der Unterklasse benötigt?*
Alle vererbten Attribute werden für die Zustandsbeschreibung aller Unterklassen benötigt.
- ✓ *Repräsentieren alle vererbten Operationen das Verhalten der Unterklasse?*
Operationen werden in den drei beteiligten Klassen aus bisheriger Sicht nicht benötigt.
- ✓ *Unterscheiden sich die Unterklassen in ihren Attributen und Operationen?*
Diese Frage zielt darauf ab, ob tatsächlich alle Unterklassen benötigt werden, oder ob sich die entsprechenden Objekte auch

direkt durch die Oberklasse repräsentieren lassen. In diesem Fall rechtfertigen Musik-CDs und Bücher jeweils die Definition einer eigenen Klasse.

- ✓ *Fördert die Klassenstruktur das Verständnis des Modells?*
Das Modell ist aufgrund der eingegrenzten Problemstellung und der geringen Anzahl beteiligter Klassen sehr übersichtlich. Das Verständnis wird durch die gebildete Klassenstruktur zumindest nicht behindert.

Identifikation von Assoziationen

Für die Identifikation von Beziehungen zwischen Klassen können Sie wiederum die Checkliste aus dem achten Kapitel heranziehen. Mögliche Arten von Klassenbeziehungen sind die Assoziation, die Aggregation und die Komposition.

Die Vorgehensweise lässt sich in drei Schritte aufteilen:

- ✓ Ermittlung von Beziehungen,
- ✓ Bestimmung von Beziehungen und
- ✓ Validierung.

Ermittlung von Klassenbeziehungen

Bei der Ermittlung von Klassenbeziehungen helfen die folgenden Fragen.

- ✓ *Welche Elemente tauschen Nachrichten aus?*
 - ✓ Die Objekte der Klasse Kunde kommunizieren mit den verbundenen Objekten der Klasse Warenkorb, indem sie dort die gewünschten Artikel eintragen.
 - ✓ Außerdem ist ein Nachrichtenaustausch zwischen der Registrierung und dem Kunden erforderlich.

- ✓ Innerhalb des Pakets *Abwicklung* bestehen Nachrichtenbeziehungen zwischen der Klasse *Auftrag* und den drei Akteuren *Auftragsabwicklung*, *Inkasso* und *Lieferservice*.
- ✓ *Welche Elemente bieten Dienste für andere Elemente an?*
Die Objekte der Klasse *Registrierung* leisten Dienste bei der Identifikation des Kunden. Sie erteilen Passwörter und prüfen den korrekten Login.
- ✓ *Sind Elemente vorhanden, die bestimmte Rollen einnehmen?*
Bei den Akteuren handelt es sich um Rollen, die von Personen eingenommen werden.
- ✓ *Lassen sich zusammengehörige organisatorische Einheiten identifizieren?*
Organisatorische Einheiten bilden die drei Rollen *Auftragsabwicklung*, *Inkasso* und *Lieferservice*. Sie kommunizieren jedoch nicht direkt miteinander, sondern erhalten ihre Informationen durch die Objekte der Klasse *Auftrag*.
- ✓ *Welche Beziehungen lassen sich aus Datenbanken entnehmen (Primär- und Fremdschlüssel)?*
Dieser Aspekt ist hier nicht relevant.

Bestimmung von Klassenbeziehungen

Nachdem die Beziehungen grundsätzlich identifiziert sind, geht es nun darum, deren Art genauer zu bestimmen. Hierbei ist zu untersuchen, ob es sich um einfache Assoziationen, Aggregationen oder Kompositionen handelt.

- ✓ *Setzt sich ein Element aus anderen Elementen zusammen (Aggregation)?*
 - ✓ Eine Aggregation besteht zwischen den Klassen *Warenkorb* und *Artikel*, da ein Warenkorb verschiedene Artikel enthält.
 - ✓ Jedes Kundenobjekt hat einen Warenkorb. Daher besteht auch hier eine Aggregation.

- ✓ Außerdem gehört zu einem Kundenobjekt auch ein Objekt der Klasse Registrierung.
- ✓ *Ist die Existenz eines Elements abhängig von der Existenz eines anderen Elements (Komposition)?*
 - ✓ Warenkorb und Artikel bilden keine Komposition, da ein Artikel stets auch ohne Warenkorb existieren kann.
 - ✓ Ein Objekt der Klasse Warenkorb kann nur als Teil eines Objekts der Klasse Kunde existieren. Daher handelt es sich hier um eine Komposition.
 - ✓ Auch zwischen den Klassen Kunde und Registrierung besteht eine Komposition, da ein Registrierungsobjekt nur in Beziehung zu einem bestimmten Kunden existieren kann.
- ✓ *Ist mit der Assoziation eine Verarbeitung verbunden (Assoziationsklasse)?*

Mit den Assoziationen ist keine Verarbeitung verbunden.
- ✓ *Wie viele Objekte sind auf beiden Seiten mindestens an der Beziehung beteiligt?*
 - ✓ Zu einem Warenkorb gehören kein oder mehrere Artikel. Umgekehrt kann auch jeder Artikel zu keinem oder mehreren Warenkörben gehören. Artikel ist hier eine Gattungsbezeichnung, z.B. für einen bestimmten Buchtitel. Ein konkretes Buchexemplar wird nicht definiert.
 - ✓ Ein Kunde hat keinen oder maximal einen Warenkorb. Umgekehrt gehört jeder Warenkorb zu genau einem Kunden (Komposition).
 - ✓ Ein Objekt der Klasse Registrierung gehört zu genau einem Objekt der Klasse Kunde (Komposition). Zu jedem Kunden gehört keine oder maximal eine Registrierung.

- ✓ Jeder Auftrag wird von jeweils genau einem Objekt der Klassen Auftragsabwicklung, Inkasso und Lieferservice bearbeitet. Umgekehrt kann jedes Objekt der drei Rollen mehrere Aufträge bearbeiten.
- ✓ *Wie viele Objekte sind auf beiden Seiten höchstens an der Beziehung beteiligt?*
(siehe Beantwortung der vorhergehenden Frage)
- ✓ *Ist die Zahl der beteiligten Objekte auf einer der beiden Seiten stets gleich?*
(siehe Beantwortung der vorhergehenden Frage)
- ✓ *Unterliegt die Beziehung irgendwelchen besonderen Zusicherungen?*
Die Beziehungen selbst unterliegen keinerlei weiteren Zusicherungen.

Validierung

Zum Abschluss dieser Checkliste sind die Beziehungen noch einmal zu überprüfen.

- ✓ *Ist die identifizierte Assoziation für das Modell relevant?*
Alle Assoziationen sind für das Modell wichtig, damit die erforderlichen Informationen ausgetauscht werden können.
- ✓ *Handelt es sich um eine Assoziation oder um eine Vererbungsbeziehung?*
Die einzige Vererbungsbeziehung besteht zwischen den Klassen Artikel, Musik-CD und Buch.

Nach diesen Vorüberlegungen ergibt sich das in der folgenden Abbildung dargestellte Klassendiagramm für das Paket Bestellung.

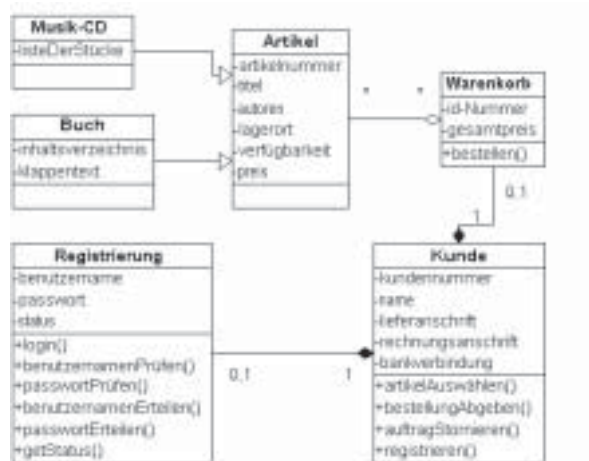


Abbildung 9.15: Klassenbeziehungen im Paket Bestellung

Für das Paket Abwicklung ergibt sich das folgende statische Modell.

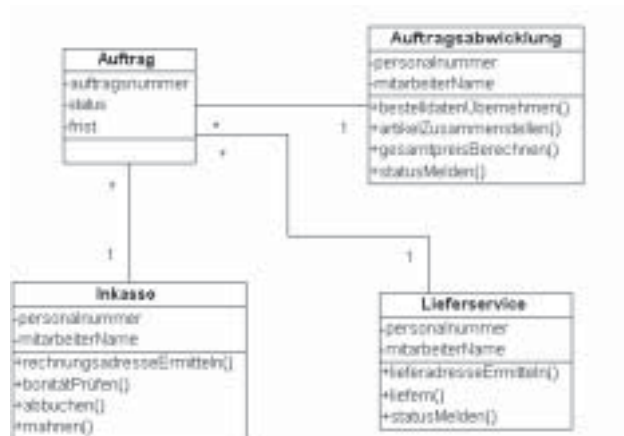


Abbildung 9.16: Klassenbeziehungen im Paket Abwicklung

Dynamisches Modell

Nachdem im statischen Modell die Klassenstruktur erstellt wurde, wird mit dem dynamischen Modell nun das Verhalten der Objekte näher untersucht. Die einzelnen Schritte bestehen dabei in

- ✓ der Identifikation von Interaktionen,
- ✓ der Definition von Zuständen und Zustandsübergängen,
- ✓ der Identifikation von Ereignissen,
- ✓ der Identifikation von Aktivitäten und
- ✓ der konkreten Beschreibung der im Zusammenhang mit der Klassenbeschreibung ermittelten Operationen.

Identifikation von Interaktionen

Interaktionen werden in der UML durch Sequenz- und Kollaborationsdiagramme abgebildet. Als Informationsquelle zur Identifikation von Interaktionen dient auch hier die verbale Beschreibung der Problemdomäne zu Beginn dieses Kapitels.



Beachten Sie, dass es nicht sinnvoll ist, für sämtliche potenziellen Interaktionsbeziehungen in einer Problemdomäne Diagramme zu entwickeln. Lediglich die Kernprozesse oder relativ komplizierte Interaktionen sollten durch Sequenz- oder Kollaborationsdiagramme veranschaulicht werden, um Abläufe und Zusammenhänge zu verdeutlichen.

Auch die Interaktionen werden wieder anhand der Checkliste des achten Kapitels analysiert.

Erfassung von Interaktionen

- ✓ *Welche Assoziationen enthält das statische Modell?*
 - ✓ Die Klasse Kunde bildet jeweils eine Komposition mit den Klassen Warenkorb und Registrierung.
 - ✓ Warenkorb und Artikel bilden eine Aggregation.
 - ✓ Die Klasse Auftrag ist durch Assoziationen mit den Klassen Auftragsabwicklung, Inkasso und Lieferservice verbunden.
- ✓ *Welche Objekte arbeiten an einer gemeinsamen Aufgabe?*
 - ✓ Die drei Rollen des Pakets Abwicklung bearbeiten gemeinsam einen Auftrag.
 - ✓ Objekte der Klassen Kunde, Registrierung und Warenkorb sind an der Bestellung beteiligt.
- ✓ *Welche Objekte sind Dienstleister für andere Objekte?*

Im Rahmen der Bestellung ist die Registrierung Dienstleister.
- ✓ *Welche Objekte benötigen Informationen von anderen Objekten?*
 - ✓ Der Informationsaustausch erfolgt im Paket Abwicklung zwischen den Objekten der Klasse Auftrag auf der einen Seite und den drei Rollen Auftragsabwicklung, Inkasso und Lieferservice auf der anderen Seite.
 - ✓ Die Objekte der Klasse Registrierung liefern bei der Bestellung Informationen über den Login.
- ✓ *Welche Objekte kommunizieren miteinander?*

Die Kommunikationsbeziehungen entsprechen dem Informationsaustausch.

Beschreibung von Interaktionen

Nachdem die Interaktionen bestimmt wurden, folgt nun im Rahmen der Beschreibung eine nähere Analyse.

- ✓ *Durch welche Ereignisse wird eine Interaktion ausgelöst?*
 - ✓ Sobald der Kunde eine Bestellung aufgibt, ist zu prüfen, ob er bereits registriert ist. Durch dieses Ereignis wird also eine Interaktion zwischen der Klasse Kunde und der Klasse Registrierung (bzw. zwischen deren Objekten) ausgelöst.
 - ✓ Ist die Bestellung aufgegeben, dann wird die Abwicklung des Auftrags ausgelöst.
- ✓ *Über welche Schnittstellen tauschen die Objekte Informationen aus?*
 - ✓ Schnittstellen zwischen Kunden-Objekten und Registrierungs-Objekten sind die Operationen `bestellungAbgeben`, `registrieren` und `login`.
- ✓ *Über welche Kommunikationswege werden Informationen ausgetauscht?*

Kommunikationswege sind die Bestellsoftware, mit deren Benutzungsoberfläche der Kunde arbeitet, die Datenbank, in der die Aufträge gespeichert werden, sowie persönliche Gespräche zwischen den Mitarbeitern der Auftragsabwicklung.
- ✓ *Welche Informationen werden ausgetauscht?*
 - ✓ Auftragsdaten
 - ✓ Kundendaten
 - ✓ Artikeldaten
- ✓ *Wann finden Interaktionen statt?*
 - ✓ Während der Bestellung durch den Kunden
 - ✓ Nach der Bestellung durch den Kunden
 - ✓ Während der Auftragsabwicklung
 - ✓ Nach der Auftragsabwicklung
- ✓ *Wo werden Informationen ausgetauscht?*

Dieser Aspekt ist hier nicht relevant.

- ✓ *In welcher Form werden Informationen übertragen?*
Informationen werden auf elektronischem Wege sowie schriftlich und mündlich übertragen.
- ✓ *In welchen Zuständen werden Informationen ausgetauscht?*
Vgl. dazu die Zustandsdiagramme weiter unten.
- ✓ *Unterliegt die Kommunikation zweier Objekte speziellen Bedingungen?*
Vgl. dazu die Verhaltensdiagramme weiter unten.
- ✓ *Wie ist der Lebenszyklus der beteiligten Objekte definiert?*
 - ✓ Die Objekte der Klassen Buch und Musik-CD existieren bereits vor Beginn des hier betrachteten Geschäftsprozesses und bleiben auch darüber hinaus bestehen.
 - ✓ Neue Kunden werden erst während der Bestellung angelegt.
 - ✓ Ein Warenkorb entsteht mit der Auswahl des ersten Artikels. Sobald der Kunde das System verlässt, endet der Lebenszyklus des Objekts vom Typ Warenkorb. Gibt der Kunde eine Bestellung auf, dann gehen die im Warenkorb gespeicherten Informationen in ein Objekt vom Typ Auftrag ein.
 - ✓ Die Rollen des Pakets Abwicklung bestehen vor und auch nach dem Ablauf des hier betrachteten Prozesses.
 - ✓ Objekte der Klasse Auftrag entstehen durch die Bestellung eines Kunden. Ihr Lebenszyklus endet, sobald der Auftrag vollständig abgewickelt ist.
- ✓ *Wann werden Objekte erzeugt oder gelöscht?*
Vgl. die Beantwortung der vorherigen Frage.
- ✓ *Welche Operationen kooperieren bei der Abbildung einer Interaktion?*
Vgl. dazu die folgenden Verhaltensdiagramme.
- ✓ *In welcher Reihenfolge sind die Operationen durchzuführen?*
Vgl. dazu die folgenden Verhaltensdiagramme.

Validierung

- ✓ *Was ist das Ziel der Kommunikation zweier Objekte?*
 - ✓ Das Ziel der Kommunikation zwischen den Objekten der Klassen Kunde und Registrierung ist die Prüfung, ob die persönlichen Daten des Kunden bereits erhoben wurden, und sie gegebenenfalls neu zu registrieren.
 - ✓ Die Kommunikation der drei Rollen des Pakets Auftragsabwicklung dienen der Durchführung des Auftrags.
- ✓ *Ist die Aufnahme der betrachteten Kommunikationsbeziehung in das Modell erforderlich?*

Alle beschriebenen Interaktionen sind Voraussetzungen für die korrekte Abwicklung des Geschäftsprozesses.
- ✓ *Sind alle Objekte nach dem derzeitigen Modell in der Lage, ihre Interaktionspartner zu erreichen (bestehen entsprechende Verbindungen im Modell)?*

Die erforderlichen Operationen sind im statischen Modell vorhanden.
- ✓ *Wurden alle benötigten Varianten einer Interaktion im Modell berücksichtigt?*

Auf dem derzeitigen Stand ist diese Frage noch nicht zu beantworten. Erst bei einer näheren Betrachtung der Verhaltensdiagramme ist hierzu eine Aussage möglich.
- ✓ *Lässt sich das Kommunikationsgeflecht durch eine Umstrukturierung von Aufgaben vereinfachen?*

Nach dem derzeitigen Informationsstand ist eine Umgestaltung nicht erforderlich.
- ✓ *Welche Auswirkungen hat die Umstrukturierung von Aufgaben auf die Zielerreichung des Systems?*

Nach dem derzeitigen Informationsstand ist eine Umgestaltung nicht erforderlich.

- ✓ Stimmen die Interaktionsdiagramme mit den Klassendiagrammen überein?

Die Übereinstimmung wird weiter unten im Text aufgegriffen.

Wie bereits erwähnt, ist es wenig sinnvoll, *sämtliche* Abläufe eines Geschäftsprozesses in einem objektorientierten Modell durch Verhaltensdiagramme abzubilden. Wir werden daher in diesem und den folgenden Abschnitten einige wesentliche Teilprozesse herausgreifen und sie mit Hilfe der dynamischen Darstellungsmittel der UML veranschaulichen.

Der Login-Prozess lässt sich mit einem Sequenzdiagramm sehr gut verdeutlichen.

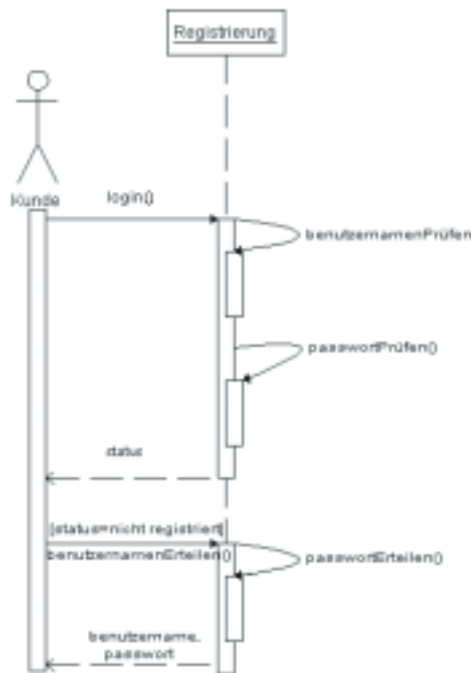


Abbildung 9.17: Sequenzdiagramm Login-Prozess

Das Sequenzdiagramm zeigt den Ablauf beim Login durch den Benutzer.

Der Kunde ist in dieser Sequenz Akteur. Er ruft die Operation login der Registrierung auf. Diese Operation prüft nun intern den Benutzernamen und das Passwort. Diese beiden Werte müssen dazu durch den Kunden als Parameter an die Operation login übergeben werden. D.h., der korrekte Aufruf der Operation lautet login(benutzername, password). Über den Wert von status erhält das Kunden-Objekt eine Rückmeldung über das Ergebnis seines Login.

Falls status den Wert nicht registriert hat, tritt das Objekt Kunde wiederum in Kontakt mit der Registrierung und ruft nun die Operation benutzernamenErteilen() auf. Intern wird diese ergänzt durch die Operation passwortErteilen. Beide Werte werden dem Kunden-Objekt als Antwort zurückgegeben und der Login-Prozess damit beendet.

Auch die Bestellung selbst lässt sich durch ein Verhaltensdiagramm darstellen. Zur Veranschaulichung dient in diesem Fall das folgende Kollaborationsdiagramm.

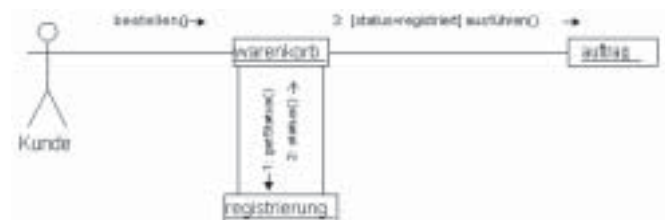


Abbildung 9.18: Kollaborationsdiagramm Bestellung

Der Akteur Kunde aktiviert die Operation bestellen des Objekts warenkorb. Bevor die Bestellung akzeptiert werden kann, wird der Status der Registrierung mit Hilfe der Operation getStatus ermit-

telt und an den Warenkorb zurückgegeben. Entspricht der Status dem Wert registriert, wird die Operation ausführen des Objekts auftrag ausgeführt, um die Bestellung abzuwickeln.

Definition von Zuständen und Zustandsübergängen

Auch für Zustandsdiagramme gilt der Grundsatz: Weniger ist mehr. Versuchen Sie nicht, ein umfassendes Modell des kompletten Problemausschnitts zu erstellen. Wählen Sie stattdessen Teilbereiche aus, die einer näheren Erläuterung oder Untersuchung bedürfen.

Für das Fallbeispiel des Web-Shops greifen wir hier das Zustandsdiagramm der Registrierung heraus. Ein Zustandsdiagramm bezieht sich immer nur auf *ein einziges* Objekt (bzw. auf eine einzige Klasse von Objekten). D.h., alle anderen Objekte, die möglicherweise an der Durchführung einer Aufgabe mit beteiligt sind, werden ausgeblendet. Zustandsdiagramme beschreiben weniger die Aktionen, die mit bestimmten Vorgängen verbunden sind, sondern in erster Linie das Verhalten eines einzelnen Objekts und seinen Wechsel von einem Zustand in den nächsten.

Die Checkliste für Zustände und Zustandsübergänge aus dem achten Kapitel enthält die folgenden Fragen:

Ermittlung von Zuständen

- ✓ *Wann wird das Objekt erzeugt?*
Das Objekt wird bei der ersten Registrierung des Kunden erzeugt und in einer Datenbank abgelegt.
- ✓ *Welche Initialwerte besitzt das Objekt?*
 - ✓ Benutzername
 - ✓ Passwort
 - ✓ Status (zunächst false für »nicht registriert«)

- ✓ *Wie ist der Anfangszustand des Objekts?*
Vgl. vorherige Frage.
- ✓ *Welche Werte können die Attribute des Objekts annehmen?*
Entscheidend ist hier das Attribut `status`, das die Werte `true` (registriert) und `false` (nicht registriert) annehmen kann.
- ✓ *Führt das Objekt beim Eintritt in den betrachteten Zustand Aktionen aus (Entry-Aktionen)?*
Benutzername und Passwort werden beim Eintritt in den Zustand prüfend anhand der Kundendaten geprüft.
- ✓ *Führt das Objekt in dem betrachteten Zustand Aktionen aus (Do-Aktionen)?*
Ist der Kunde noch nicht registriert, dann werden die Aktionen `benutzernamenErteilen` und `passwortErteilen` im Zustand `registrierend` ausgeführt.
- ✓ *Führt das Objekt beim Austritt aus dem betrachteten Zustand Aktionen aus (Exit-Aktionen)?*
Austrittsaktionen sind hier nicht vorgesehen.
- ✓ *Rufen die mit dem betrachteten Zustand verbundenen Aktionen externe Subroutinen auf (Include-Aktionen)?*
Externe Subroutinen liegen hier nicht vor.
- ✓ *Welche Endzustände besitzt das Objekt?*
Als Endzustände können `Account erteilt` (nach Registrierung eines neuen Kunden) und `geprüft` (nach erfolgreicher Prüfung der Anmeldung) unterschieden werden.
- ✓ *Gibt es anonyme Zustände?*
Anonyme Zustände sind hier nicht vorhanden.
- ✓ *Endet die Existenz des Objekts mit seinem Endzustand?*
Nein, der Lebenszyklus des Objekts endet nicht mit seinem Endzustand.

Ermittlung von Zustandsübergängen

- ✓ *Welche Schnittstellen besitzt das Objekt?*
Es existiert eine Schnittstelle zur Klasse Kunde.
- ✓ *Welche eigenen Operationen manipulieren die Attributwerte (= Zustand) des Objekts?*
 - ✓ benutzernamenErteilen
 - ✓ passwortErteilen
- ✓ *Wodurch werden diese Operationen aktiviert?*
Beide Operationen werden durch ein Objekt der Klasse Kunde aktiviert, falls die Überprüfung ergeben hat, dass der Kunde noch nicht registriert ist.
- ✓ *Welche konkreten Ereignisse führen zu einem Zustandsübergang?*
 - ✓ Der Zustand prüfend wird durch die Bestellung des Kunden ausgelöst.
 - ✓ Fällt die Prüfung positiv aus, folgt der Zustand geprüft.
 - ✓ Fällt die Prüfung negativ aus, wird der Zustand registrierend erreicht.
 - ✓ Nach der Registrierung ergibt sich der Zustand Account erteilt durch ein implizites Ereignis.
- ✓ *Unter welchen Bedingungen führt ein Ereignis zu einem Zustandsübergang?*
Vgl. vorherige Frage.
- ✓ *Welche impliziten Ereignisse können auftreten?*
Ein implizites Ereignis sorgt für den Übergang von registrierend zu Account erteilt.
- ✓ *Kann ein und dasselbe Ereignis unterschiedliche Reaktionen und Zustandsübergänge auslösen?*
Dieser Fall tritt hier nicht auf.

Validierung

- ✓ Sind die identifizierten Zustände für das Systemverständnis erforderlich?
Alle genannten Zustände werden für die Registrierung benötigt.
- ✓ Sind alle im Diagramm definierten Zustände vom Anfangszustand aus durch Ereignisse und Zustandsübergänge erreichbar?
Ja (vgl. folgende Abbildung).
- ✓ Können alle Zustände (abgesehen von den Endzuständen) durch Ereignisse verlassen werden?
Ja (vgl. folgende Abbildung).
- ✓ Sind alle Folgezustände eindeutig?
Alle Zustände und ihre Übergänge sind eindeutig definiert.
- ✓ Gibt es zu allen identifizierten Aktionen eine entsprechende Operation im Klassendiagramm?
Alle Aktionen sind als Operationen der Klasse Registrierung definiert.

Für die Registrierung ergibt sich aus der Bearbeitung der Checkliste das folgende Zustandsdiagramm.

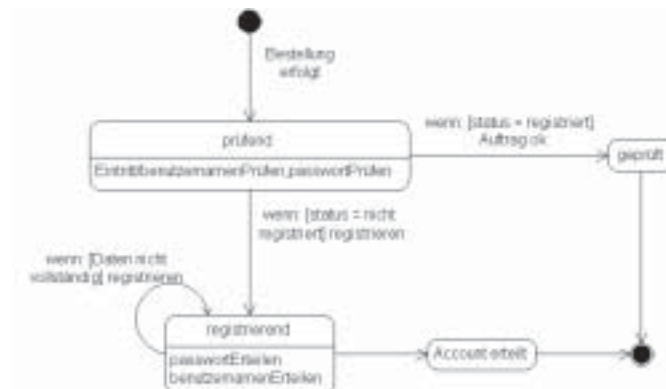


Abbildung 9.19: Zustandsdiagramm Registrierung

Durch die Aktion eines Kunden, der eine Bestellung aufgeben möchte, wird die Registrierung aktiviert. Das Objekt nimmt zunächst den Zustand prüfend an, in dem es die korrekte Eingabe des Benutzernamens und des zugehörigen Passworts durch den Kunden überprüft. Diese beiden Aktionen finden sich in der Abbildung als Eintrittsaktionen des Zustands prüfend.

In welchen Zustand das Objekt anschließend übergeht, hängt von einer Bedingung ab. Ist der Kunde bereits registriert, dann nimmt das Attribut status den Wert true an und das Registrierungsobjekt wechselt in den Zustand geprüft. In diesem Fall steht der Ausführung des Auftrags nichts mehr im Wege.

Ist der Kunde jedoch noch nicht registriert, so lautet der Folgezustand von prüfend registrierend. Das Objekt erteilt dem Kunden dann einen Benutzernamen und ein Passwort (Eintrittsaktionen). In diesem Zustand verharret das Objekt so lange, bis der Kunde die erforderlichen Daten vollständig eingegeben hat. Erst dann wird durch ein implizites Ereignis der Zustand Account erteilt erreicht.

Die beiden Zustände geprüft und Account erteilt müssen hier nicht unbedingt Endzustände sein. Geht man davon aus, dass ein Kunde weitere Bestellungen vornimmt, dann könnten beide Zustände wieder in den Anfangszustand prüfend übergehen, sobald eine weitere Bestellung eingeht (Ereignis).

Spätestens bei der Definition der Zustände und Zustandsübergänge werden Sie feststellen, dass Sie im Verlauf des objektorientierten Analyseprozesses die Problemstellung aus ständig wechselnden Blickwinkeln betrachten. Zunächst gehen Sie von der allgemeinen Problembeschreibung aus und teilen die Domäne in Pakete auf. Anschließend dringen Sie immer tiefer in die statischen und dynamischen Strukturen dieser Pakete ein. Bei der Beschreibung der Zustände betrachten Sie dann einzelne Objekte in ihrem Lebenszyklus.

Es erfolgt also eine ständige Verschiebung der Betrachtungsweise vom Abstrakten zum Konkreten, von einer Gesamtschau zur Detailbetrachtung und vom einzelnen Objekt zu kompletten Strukturen oder Abläufen. Diese Phasen sind eingebettet in einen evolutionären Prozess der ständigen Verfeinerung und Anpassung.

Bedenken Sie, dass es sich bei der objektorientierten Analyse um einen evolutionären Prozess handelt. Jeder neue Blickwinkel, den Sie im Laufe des Modellierungsprozesses einnehmen, ergibt auch neue Erkenntnisse für vorangegangene Phasen. So hat das dynamische Modell Auswirkungen auf die Struktur des statischen Modells und umgekehrt.

Die zahlreichen Interdependenzen zwischen den unterschiedlichen Blickwinkeln, vertreten durch die verschiedenen UML-Diagramme, führen zu ständigen Anpassungen und Erweiterungen, bis ein Objektmodell für eine gegebene Problemstellung vollständig und in sich konsistent ist.

Identifikation von Ereignissen

Auch bei den Ereignissen wollen wir uns auf das Beispiel der Registrierung beschränken.

Ermittlung von Ereignissen

- ✓ Welche Beziehungen besitzt die Klasse bzw. das Objekt laut statischem Objektmodell?

Die Klasse Registrierung besitzt eine Assoziation zu der Klasse Kunde.

- ✓ Welche Zustände sind bereits ausgemacht worden?

- ✓ prüfend

- ✓ geprüft

- ✓ registrierend
- ✓ Account erteilt
- ✓ *Was unterscheidet diese Zustände?*
 - ✓ Attributwerte
 - ✓ Zeitpunkt und Fortschritt der Durchführung von Operationen
- ✓ *Welche externen Operationen greifen auf das Objekt zu?*

Der Zugriff erfolgt durch ein Objekt vom Typ Kunde aus seiner Operation *bestellungAbgeben* heraus.
- ✓ *Welche internen Operationen führen zu Zustandsänderungen und wodurch werden diese Operationen ausgelöst?*
 - ✓ *benutzernamenErteilen*
 - ✓ *passwortErteilen*
 - ✓ Die Operationen werden je nach Status der Überprüfung des Kunden (registriert, nicht registriert) ausgelöst.
- ✓ *Welche Ereignisse werden von Objekten ausgelöst?*

Die Registrierung selbst löst beim Kunden-Objekt den Eingang einer Statusmeldung aus (vgl. obiges Sequenzdiagramm).
- ✓ *Welche Ereignisse werden von Personen (z.B. Anwender der Software) ausgelöst?*

Der Benutzer (Kunde) löst mit seiner Bestellung die Überprüfung der Registrierung aus.
- ✓ *Welche impliziten Ereignisse können auftreten?*

Ein implizites Ereignis führt zum Übergang von *registrierend* nach *Account erteilt*.
- ✓ *Welche zeitlichen Ereignisse (Zeitpunkt, Zeitspanne) können auftreten?*

Zeitliche Ereignisse sind nicht vorgesehen. Allerdings ist die Registrierung häufig mit einem Zeitlimit verbunden, das auch hier eingebaut werden könnte.

- ✓ *Welche Bedingungen sind selbst Ereignisse?*
Die Bedingung Daten nicht vollständig könnte durchaus auch als Ereignis betrachtet werden.

Beschreibung von Ereignissen

- ✓ *Unter welchen Bedingungen werden die identifizierten Ereignisse ausgelöst?*
Vgl. die Checkliste zu den Zuständen und Zustandsübergängen.
- ✓ *Führt ein und dasselbe Ereignis bei verschiedenen Zuständen zu unterschiedlichen Reaktionen bzw. Ergebnissen?*
Dieser Fall tritt hier nicht auf.
- ✓ *Betrifft ein Ereignis mehrere Objekte?*
 - ✓ Von den Ereignissen der Registrierung ist auch das Kunden-Objekt betroffen.
 - ✓ Alle übrigen Objekte werden erst dann an dem Geschäftsprozess beteiligt, wenn die Registrierung erfolgreich abgeschlossen wurde.

Validierung

- ✓ *Ist das betrachtete Ereignis für das Modell von grundsätzlicher Bedeutung?*
Die Zustandsübergänge der Objekte vom Typ Registrierung und ihre auslösenden Ereignisse sind entscheidend für den weiteren Verlauf des Geschäftsprozesses. Ohne erfolgreiche Registrierung wird ein Auftrag gar nicht erst ausgeführt.
- ✓ *Welche Konsequenzen hat der Verzicht auf die Abbildung des Ereignisses für das Modell?*
Vgl. die Beantwortung der vorhergehenden Frage.
- ✓ *Ist das betrachtete Ereignis bereits in anderer Form oder unter einer anderen Bezeichnung berücksichtigt worden?*
Dieser Fall ist hier nicht gegeben.

- ✓ *Kann das betrachtete Ereignis als Spezialfall vernachlässigt werden?*
Dieser Aspekt ist hier nicht relevant.
- ✓ *Kann das betrachtete Ereignis als Spezialfall in ein eigenes (Teil-)Modell ausgelagert werden?*
Auch diese Frage spielt hier keine Rolle.

Die Identifikation von Ereignissen ist weitgehend in die Beschreibung der Zustände und Zustandsübergänge eingebunden. Im Rahmen dieser Überlegungen ist zu prüfen, aufgrund welcher Ereignisse ein Zustandsübergang erfolgt. Zudem müssen Sie sich bei der Beschreibung der Zustände Gedanken darüber machen, welche Ursachen ein Ereignis hat und welche Wirkungen es entfaltet. Diese äußern sich dann in einem Ausgangs- und in einem Zielzustand.

Identifikation von Aktivitäten

Aktivitäten konkretisieren die internen Aktionen von Objekten. Sie dienen der Beschreibung einzelner Verarbeitungsschritte innerhalb eines Ablaufs. Beispielhaft werden im Folgenden die Aktivitäten im Rahmen der Bestellung analysiert.

Ermittlung von Aktivitäten

- ✓ *Welche Aktivitäten ergeben sich aus den Zustandsdiagrammen?*
Aus dem Zustandsdiagramm für die Registrierung ergeben sich die folgenden Aktivitäten:
 - ✓ Benutzernamen prüfen
 - ✓ Passwort prüfen
 - ✓ Benutzernamen erteilen
 - ✓ Passwort erteilen

- ✓ *Welche internen Prozesse laufen in einem Objekt ab?*
Neben den gerade genannten Aktivitäten innerhalb der Registrierungs-Objekte ergeben sich auch Aktivitäten der Kunden-Objekte:
 - ✓ Artikel auswählen
 - ✓ Bestellung abgeben
 - ✓ Registrieren
 - ✓ Auftrag ausführen
- ✓ *Von welchen Bedingungen hängt die Ausführung der Prozesse ab?*
Die Bedingungen wurden bereits im Zustandsdiagramm der Registrierung ermittelt.
- ✓ *Welchen Regeln unterliegt ein Prozess?*
Die Bestellung kann nur dann abgegeben und als Auftrag weiterverfolgt werden, wenn zuvor eine Registrierung erfolgt ist.
- ✓ *Wer ist für die Durchführung der Aktivitäten verantwortlich (swimlanes)?*
 - ✓ Kunde
 - ✓ Registrierung
 - ✓ Auftrag
- ✓ *Welche Ergebnisse erzeugen die Prozesse?*
Ergebnis der Bestellung ist ein Auftrag.
- ✓ *Welche alternativen Zustände sind mit einer Aktivität verknüpft?*
Vgl. das obige Zustandsdiagramm.
- ✓ *Wie lauten jeweils die Bedingungen (Entscheidungen) für das Feuern der alternativen Zustandsübergänge?*
Vgl. das obige Zustandsdiagramm.
- ✓ *Führt ein Zustandsübergang zu mehreren verschiedenen Aktivitäten (Splitting)?*
Ein Splitting liegt in diesem Fall nicht vor.

- ✓ *Haben mehrere parallel laufende Aktivitäten eine gemeinsame Folgeaktivität (Synchronisation)?*
Eine Synchronisation ergibt sich nach Überprüfung der vorhandenen Registrierung, wenn der Auftrag ausgeführt wird (vgl. das Aktivitätsdiagramm weiter unten).
- ✓ *Ist es sinnvoll, einzelne Objektzustände mit in das Aktivitätsdiagramm aufzunehmen, um Zusammenhänge zu verdeutlichen?*
Die relevanten Zustände können als Bedingungen in das Aktivitätsdiagramm aufgenommen werden.
- ✓ *Werden Signale an Objekte gesendet?*
 - ✓ Es werden Signale an die Auftragsabwicklung gesendet, falls eine Bestellung erfolgreich abgegeben wird.
 - ✓ Auch mit der Kundendatenbank und mit der Artikeldatenbank gibt es einen Signalaustausch.
- ✓ *Werden Signale von Objekten empfangen?*
Es werden Signale von der Kundendatenbank und von der Artikeldatenbank empfangen.
- ✓ *Mit welchen Objekten werden diese Signale ausgetauscht?*
Vgl. die Beantwortung der beiden vorhergehenden Fragen.
- ✓ *Wann werden die Signale ausgetauscht?*
 - ✓ Der Signalaustausch mit der Auftragsabwicklung erfolgt nach erfolgreicher Bestellung.
 - ✓ Mit der Kundendatenbank und der Artikeldatenbank werden während des Bestellvorgangs Signale ausgetauscht.
- ✓ *Unter welchen Bedingungen werden die Signale ausgetauscht?*
 - ✓ Ein Signalaustausch mit der Auftragsabwicklung erfolgt nur dann, wenn die Bestellung erfolgreich ist.
 - ✓ Der Signalaustausch mit der Kundendatenbank und mit der Artikeldatenbank ist an keine besonderen Bedingungen geknüpft.

Validierung

- ✓ *Ist die betrachtete Aktivität wichtig für das Verständnis der Abläufe in dem System?*
Alle identifizierten Aktivitäten sind für die Bestellung von Bedeutung.
- ✓ *Wurden alle relevanten Aktivitäten berücksichtigt?*
Alle für die Bestellung wesentlichen Aspekte wurden berücksichtigt.
- ✓ *Wurden alle vorhergehenden Aktivitäten erfasst?*
Dieser Aspekt ist hier nicht relevant.
- ✓ *Wurden alle Folgeaktivitäten erfasst?*
Dieser Aspekt ist hier nicht relevant.
- ✓ *Sind alle Aktivitäten als Operationen im Klassenmodell definiert?*
Alle Aktivitäten tauchen als Operationen der Klassen Kunde, Warenkorb und Registrierung im Klassenmodell auf.

Aus der Checkliste ergibt sich das in Abbildung 9.20 dargestellte Aktivitätsdiagramm.

Für die Aktivitäten, die im Zusammenhang mit einer Bestellung anfallen, wurden drei Zuständigkeitsbereiche identifiziert: Kunde, Registrierung und Auftrag.

Der Kunde wählt so lange Artikel aus, bis er seine Bestellung komplettiert hat. Anschließend gibt er seine Bestellung ab. Die Registrierung prüft nun, ob der Kunde bereits geführt wird. Dazu werden Benutzername und Passwort geprüft. Ist diese Prüfung erfolgreich, kann der Auftrag ausgeführt werden. Falls der Kunde noch nicht registriert ist, werden ihm ein Benutzername und ein Passwort erteilt und der Auftrag erst danach ausgeführt.

Das Aktivitätsdiagramm der Registrierung verdeutlicht erneut die Bedeutung des Blickwinkels auf die Problemstellung für das Objektmodell. Das obige Modell gibt den Prozess der Bestellung noch auf einem relativ abstrakten Niveau wieder.

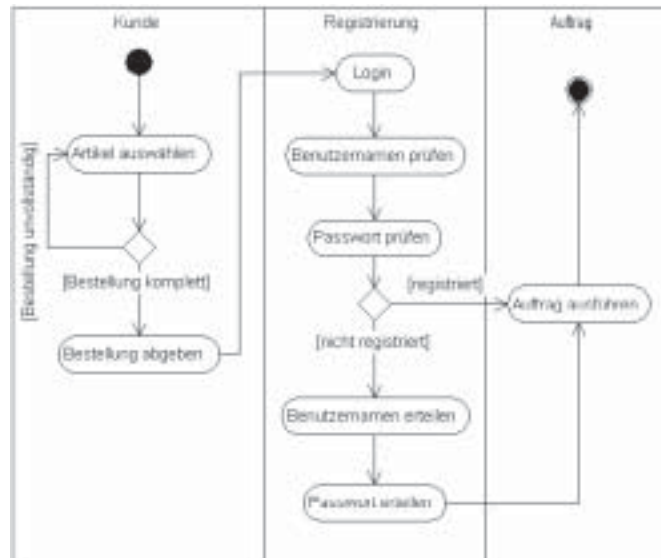


Abbildung 9.20: Aktivitätsdiagramm Bestellung

Jede einzelne der hier aufgeführten Aktivitäten lässt sich bei Bedarf noch einer tiefer gehenden Analyse unterziehen, in der weitere Teilaktivitäten ermittelt werden können. Es hängt stets von den Anforderungen des Modellierers ab, wie tief er einen Geschäftsprozess analysiert.

Beschreibung von Operationen

Der letzte Schritt bei der Erstellung des dynamischen Modells ist die Beschreibung der Operationen. Bisher wurden die Operationen lediglich mit ihrem Namen und ihren groben Aufgaben identifiziert. Sie sind nun noch in ihrer Struktur und in ihrem internen Ablauf zu spezifizieren.

Wichtige Informationen zur Definition einer Operation sind neben dem Namen ihre Parameter, Zweck, Datentyp, Vorgehensweise und Ergebnis. Am Beispiel der Operation `Auftrag ausführen`, die zum Verhalten der Objekte vom Typ `Auftrag` gehört, wird die Beschreibung von Operationen verdeutlicht.

- ✓ Name der Operation: `Auftrag ausführen`
- ✓ Klasse: `Auftrag`
- ✓ Paket: `Abwicklung`
- ✓ Parameter: `Kundendaten`, `Bestelldaten`
- ✓ Datentyp: Es handelt sich um eine typlose Operation, die keine Werte zurückliefert.
- ✓ Verwendete Operationen anderer Klassen: `bestelldatenübernehmen` der Klasse `Auftragsabwicklung`, `liefern` der Klasse `Lieferservice`, `abbuchen` der Klasse `Inkasso`.
- ✓ Vorgehensweise: Die Ausführung des Auftrags erfolgt durch die Rollen `Auftragsabwicklung`, `Lieferservice` und `Inkasso`. Die Klasse `Auftrag` (bzw. deren Objekte) führt den Auftrag nicht selbst aus.
- ✓ Ergebnis: Auftrag ist ausgeführt.

Die Beschreibung der Operation `Auftrag ausführen` wird durch die Beantwortung folgender Fragen mit Inhalt gefüllt.

- ✓ *Welchem Zweck dient die Operation?*
Der Zweck ist die Steuerung der Auftragsdurchführung.
- ✓ *Wodurch wird die Operation ausgelöst?*
Die Operation wird nach vorheriger Prüfung der Registrierung durch die Bestellung des Kunden ausgelöst.
- ✓ *Welche Daten benötigt die Operation zur Durchführung ihrer Aufgabe?*
 - ✓ `Bestelldaten`
 - ✓ `Kundendaten`

- ✓ *Wie führt die Operation ihre Aufgabe aus?*
Es werden Operationen der drei Rollen Auftragsabwicklung, Lieferservice und Inkasso aufgerufen, die für die eigentliche Durchführung des Auftrags sorgen.
- ✓ *In welcher Reihenfolge werden bestimmte Teilaktivitäten ausgeführt?*
Die Reihenfolge spielt hier keine Rolle.
- ✓ *Unter welchen Bedingungen werden bestimmte Teilaktivitäten ausgeführt?*
Auch von Bedingungen soll in diesem einfachen Beispiel abgesehen werden. Denkbar ist aber beispielsweise die Einführung einer Restriktion, die zunächst den Zahlungseingang verlangt, bevor die bestellten Artikel ausgeliefert werden.
- ✓ *Auf welche internen Attribute wird bei der Verarbeitung zugegriffen?*
 - ✓ auftragsnummer
 - ✓ status
 - ✓ frist
- ✓ *Welche internen Attribute werden manipuliert?*
In der Regel ändert sich während der Bearbeitung der Operation lediglich der Status des Auftrags.
- ✓ *Welche Dienste anderer Objekte benötigt die Operation?*
Siehe oben.
- ✓ *Welche Informationen benötigt die Operation von anderen Objekten?*
 - ✓ Kundendaten
 - ✓ Bestelldaten
- ✓ *Welche Informationen benötigen andere Objekte von dieser Operation?*
 - ✓ auftragsnummer
 - ✓ status
 - ✓ frist

- ✓ *Welche Ergebnisse erzielt die Operation?*
Ergebnis der Operation ist die Erledigung der Kundenbestellung.
- ✓ *Sendet die Operation eine Antwort an das aufrufende Objekt?*
Es werden keine Antworten gesendet.
- ✓ *Übermittelt die Operation ihre (Zwischen-) Ergebnisse an das aufrufende Objekt?*
Auch die Übermittlung von Zwischenergebnissen ist nicht vorgesehen.
- ✓ *In welchem Bereich ist die Operation sichtbar (public, protected, private)?*
Die Operation muss sowohl innerhalb des eigenen Pakets als auch im Paket Abwicklung sichtbar sein, da sie von dort aus der Klasse Registrierung heraus aufgerufen wird.

Zur exakten Definition von Operationen kann auch eine formale Sprache, wie beispielsweise eine Programmiersprache, verwendet werden.

Zur Validierung der ermittelten Eigenschaften einer Operation dienen die folgenden Fragen aus der Checkliste des achten Kapitels.

- ✓ *Ist der in der Klassenbeschreibung gewählte Name der Operation aussagekräftig?*
Die Aufgabe der Operation ergibt sich aus ihrem Namen.
- ✓ *Ist der Name der Operation in dem jeweiligen Namensraum (Paket) eindeutig?*
Der Name ist eindeutig.
- ✓ *Wird die Operation in allen möglichen Fällen ordnungsgemäß beendet?*
Diese Frage lässt sich bei der hier vorgenommenen groben Betrachtungsweise nicht beantworten.
- ✓ *Kann die Operation auf alle erforderlichen Daten (Attribute, Parameter) zugreifen?*

Die Beziehungen und Schnittstellen sind so gestaltet, dass alle erforderlichen Daten erreichbar sind.

- ✓ *Hat die Operation Zugriff auf alle benötigten externen Operationen (Sichtbarkeit)?*
Auch der Zugriff auf die benötigten anderen Operationen ist im Modell gewährleistet.
- ✓ *Ist die Operation für alle anderen Objekte, die auf sie zugreifen müssen, sichtbar?*
Die Operation wird aus der Klasse Registrierung aufgerufen. Dort ist sie sichtbar.
- ✓ *Ist die Parameterliste vollständig?*
Die Parameterliste ist auf dem vorliegenden Betrachtungsniveau vollständig.
- ✓ *Haben alle Parameter den korrekten Datentyp?*
Diese Frage ist hier nicht relevant.
- ✓ *Gibt die Operation die richtigen Werte zurück?*
Die Operation gibt gar keine Werte zurück.
- ✓ *Ist der Datentyp der Operation für alle möglichen Aufrufe korrekt?*
Diese Frage ist hier nicht relevant.



KAPITEL

Fallbeispiel: »Passagierabfertigung«

Die vorgestellten Konzepte der UML werden im Folgenden an dem Fallbeispiel der Passagierabfertigung an einem Flughafen verdeutlicht. Die Vorgänge von der Gepäckaufgabe bis hin zur Einnahme des Sitzplatzes sollen in Auszügen in einem objektorientierten Modell abgebildet werden.

10



Fallbeispiel: »Passagierabfertigung«

Als Fallbeispiel soll die Passagierabfertigung an einem Flughafen dienen. Die Vorgänge von der Gepäckaufgabe bis hin zur Einnahme des Sitzplatzes im Flugzeug sollen in Auszügen in einem objekt-orientierten Modell abgebildet werden.

Grobe Systemanalyse

Im Rahmen der groben Systemanalyse sind folgende Schritte durchzuführen, die im achten Kapitel allgemein beschrieben wurden:

- ✓ Beschreibung des Anwendungsfalls
- ✓ Abgrenzung des relevanten Problemausschnitts
- ✓ Abgrenzung von Teilsystemen (Paketen)
- ✓ Definition von Schnittstellen zwischen Paketen

Beschreibung des Anwendungsfalls

Zunächst ist der Anwendungsfall in groben Zügen verbal zu beschreiben. Daraus werden die beteiligten Akteure, ihre Aufgaben und auftretende Ereignisse abgeleitet. Im Laufe der weiteren Ermittlung von Elementen und ihrer Beschreibung wird der Anwendungsfall ständig verfeinert, eingeschränkt oder erweitert.

Eine grobe verbale Beschreibung des Anwendungsfalls »Passagierabfertigung« könnte wie folgt aussehen:

Auf dem Weg zu seinem Sitzplatz muss der Passagier zunächst zum Schalter der Fluggesellschaft im Flughafen-Terminal, um sich und sein Gepäck einzuchecken. Dazu legt er sein Flugticket vor und stellt sein Ge-

päck auf das Laufband. Das Gepäck wird gewogen und mit einem Label versehen, das mit einer Identifikationsnummer, der Flugnummer sowie Abflug- und Zielflughafen gekennzeichnet ist. Anschließend transportiert das Laufband die Gepäckstücke zur Gepäckabfertigung, damit sie in das Flugzeug geladen werden können. Überschreitet das Gepäck das zulässige Gewicht, muss der Fluggast eine zusätzliche Gebühr entrichten.

Die Angaben auf dem Ticket werden von dem Mitarbeiter des Check-In in den Computer eingegeben. Dabei bekommt der Fluggast einen Sitzplatz zugewiesen, falls er eine Festbuchung besitzt. Hat er lediglich ein Stand-by-Ticket erworben, wird er auf die Warteliste gesetzt. In diesem Fall wird auch sein Gepäck noch nicht entgegen genommen. Bei Auslandsflügen wird der Reisepass des Fluggastes bereits am Check-In kontrolliert. Passagiere mit Festbuchung erhalten ihre Bordkarte, die u.a. Informationen über den Flugsteig (Gate), die Sitzplatznummer und den voraussichtlichen Zeitpunkt für den Einsteigevorgang (Boarding) enthält.

Sobald das Flugzeug für den Einsteigevorgang bereit ist, werden die Passagiere mit Festbuchung aufgerufen und begeben sich zum Gate. Sie geben den dortigen Mitarbeitern der Fluggesellschaft ihre Bordkarte und ihren Flugschein. In einem Scan-Vorgang werden die Informationen der Bordkarte in einen Computer eingelesen, der daraus die so genannte Passenger Information List (PIL) zusammenstellt. Der Passagier erhält seine Bordkarte zurück, während der Flugschein für die spätere Abrechnung der Fluggesellschaft mit dem Reisebüro dient. Etwa 15 Minuten vor dem geplanten Abflug werden die Passagiere der Warteliste aufgerufen, falls noch Plätze im Flugzeug frei sind.

Sobald die Passagiere das Flugzeug betreten haben, weisen die Flugbegleiter ihnen ihren Platz an. Die Passagiere verstauen ihr Handgepäck und nehmen ihren Sitzplatz ein.

Währenddessen ist das Gepäck verladen worden. Am Flugzeug werden die Label der Gepäckstücke noch einmal gescannt, um Irrläufer herauszufiltern.

Der Stationsmanager vergleicht die Anzahl der eingesammelten Flugscheine mit der Zahl der Passagiere an Bord des Flugzeugs und übergibt die PIL dem Purser an Bord der Maschine. Außerdem überprüft er, ob

alle Passagiere, deren Gepäck verladen wurde, auch an Bord gegangen sind. Falls nicht, müssen die entsprechenden Gepäckstücke aus Sicherheitsgründen wieder ausgeladen werden.

Wenn alle Passagiere das Flugzeug betreten haben und die Maschine komplett beladen und betankt ist, werden die Türen geschlossen. Die Passagierabfertigung ist damit abgeschlossen.

Aus den Informationen dieser groben verbalen Beschreibung lässt sich das Geschäftsprozessdiagramm der folgenden Abbildung ableiten.

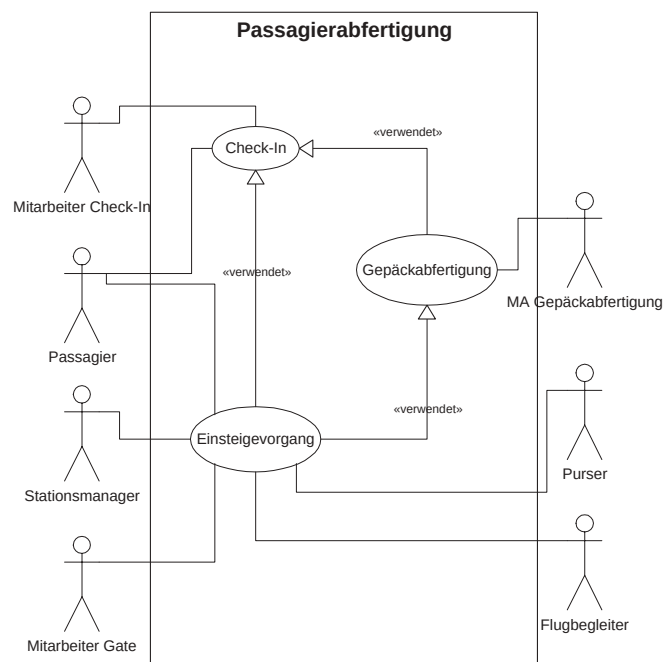


Abbildung 10.1: Geschäftsprozessdiagramm Passagierabfertigung

Beteiligte Rollen des Geschäftsprozesses Passagierabfertigung sind:

- ✓ Der Passagier selbst führt Aktionen während des Check-In und des Einsteigevorgangs aus.
- ✓ Der Mitarbeiter Check-In wickelt für die Fluggesellschaft den Check-In ab.
- ✓ Die Rollen Stationsmanager und Mitarbeiter Gate sind außerhalb des Flugzeugs verantwortlich für den Einsteigevorgang. Flugbegleiter und Purser nehmen die mit diesem Anwendungsfall verbundenen Aufgaben an Bord des Flugzeugs wahr.
- ✓ Der Mitarbeiter der Gepäckabfertigung (Rolle MA Gepäckabfertigung) sorgt dafür, dass das Gepäck in den Laderaum des Flugzeugs gelangt. Er greift dabei auf die Informationen aus dem Check-In zurück.
- ✓ Der Anwendungsfall Einsteigevorgang verwendet Check-In (u.a. Sitzplatzvergabe) und Gepäckabfertigung (u.a. Prüfung, ob zu jedem verladenen Koffer ein Passagier an Bord ist).

Abgrenzung des relevanten Problemausschnitts

Die Abgrenzung des relevanten Problembereichs dient dazu, für das Modell Wichtiges von Unwichtigem zu trennen. Mit Hilfe der verbalen Problembeschreibung wird im Rahmen eines evolutionären Prozesses der Systemanalyse der Kern der Problemstellung herausgearbeitet und ständig verfeinert.

Als Hauptkriterium sind die Ziele des Geschäftsvorfalles heranzuziehen. Ferner ist zu überlegen, auf welche Elemente in dem Objektmodell verzichtet werden kann, ohne das Abbild der realen Problemstellung zu beeinträchtigen. Beziehungen zu Elementen außerhalb der betrachteten Problemdomäne, die dennoch eine wichtige Rolle spielen, stellen sich u.U. erst im weiteren Verlauf der Analyse heraus. Der Problemausschnitt wird daher immer wieder überarbeitet und angepasst.

Anhand der in den Checklisten aufgeführten Fragen lässt sich der relevante Problemausschnitt wie folgt eingrenzen:

- ✓ *Welche Ziele werden mit dem Geschäftsprozess verfolgt?*
 - ✓ Abbildung des Problembereichs »Passagierabfertigung«
 - ✓ Identifikation der beteiligten Rollen, ihrer Aufgaben und Aktivitäten
 - ✓ Analyse der wesentlichen Teilprozesse der Passagierabfertigung
 - ✓ Aufzeigen von Zusammenhängen zwischen den Teilprozessen
- ✓ *Ist das betrachtete Element für die Ziele und Aufgaben des Geschäftsvorfalls von Bedeutung?*
 - ✓ Einige Bereiche, die nicht unmittelbar mit der Passagierabfertigung zusammenhängen, werden an dieser Stelle ausgegrenzt.
 - ✓ Dazu zählen u.a. der Sicherheitscheck, die Überbuchung, die Beladung des Flugzeugs mit Fracht (außer Gepäck), das Catering sowie das Reinigen und Betanken der Maschine.
- ✓ *Welche Auswirkungen hat der Verzicht auf die Integration der betrachteten Elemente für die Erfüllung der Aufgabe oder des Ziels?*
 - ✓ Die ausgegrenzten Elemente haben keinen großen Einfluss auf den Check-In, die Gepäckabfertigung und den Einsteigevorgang. Sie führen lediglich zu Variationen im Ablauf, die sich später mühelos ergänzen lassen.
 - ✓ Eine Beeinträchtigung der mit der Analyse verfolgten Ziele ist nicht erkennbar.
- ✓ *Welche Rollen sind an der Aufgabe beteiligt?*
 - ✓ Beteiligte Rollen sind: Mitarbeiter Check-In, Passagier, Stationsmanager, Mitarbeiter Gate, Mitarbeiter Gepäckabfertigung, Purser und Flugbegleiter.

- ✓ Mit den genannten Akteuren sind, soweit bisher erkennbar, alle beteiligten Rollen erfasst.
- ✓ Sollten sich in der weiteren Analyse zusätzliche Rollen ergeben, werden sie später hinzugefügt.
- ✓ *Kann die Erfüllung von Aufgaben durch eine Umgestaltung in aufbau- oder ablauforganisatorischer Hinsicht optimiert werden?*
 - ✓ Die Reorganisation ist in diesem Fallbeispiel nicht vorgesehen. Bei der Analyse Ihrer eigenen betrieblichen Geschäftsprozesse sollten Sie jedoch die Gelegenheit nutzen, Schwachstellen in der Aufbau- oder Ablauforganisation aufzudecken und gegebenenfalls zu beseitigen.
- ✓ *Welche Beziehungen und Abhängigkeiten besitzt das betrachtete Element zu anderen Elementen innerhalb des Systems?*
 - ✓ Die gegenseitigen Beziehungen der drei Anwendungsfälle sind der obigen Abbildung zu entnehmen. Die Gepäckabfertigung verwendet den Check-In. Der Einsteigevorgang verwendet den Check-In und die Gepäckabfertigung.
- ✓ *Welche Beziehungen und Abhängigkeiten besitzt das betrachtete Element zu anderen Elementen außerhalb des Systems?*
 - ✓ Die Beziehungen zu Elementen außerhalb des Systems sind recht vielseitig. Die genannten Rollen werden von Personen bekleidet, die größtenteils Mitarbeiter des Flughafens oder einer Fluggesellschaft sind. Als solche unterliegen sie gewissen Beschränkungen, die außerhalb des Systems definiert werden (z.B. Tarifverträge, gesetzliche Bestimmungen).
 - ✓ Auch einige der ausgeblendeten Bereiche wirken in das System hinein. So gehören die den Fluggästen zugewiesenen Sitzplätze zu einem Flugzeug. Das Flugzeug ist Teil der Flotte einer Fluggesellschaft. Es ist nicht sinnvoll, das Flugzeug in dem System »Passagierabfertigung« zu modellieren. Daher erfolgen voraussichtlich Zugriffe auf Objekte, die in anderen

Systemen definiert sind. Wie diese Zugriffe konkret gestaltet werden und welche internen sowie externen Elemente betroffen sind, wird im weiteren Verlauf der Analyse ermittelt.

Abgrenzung von Teilsystemen (Paketen)

Die Bildung von Paketen schafft überschaubare Teilsysteme, die sich gegenseitig ergänzen und in einem Gesamtsystem zusammenarbeiten. Die bisherigen Ergebnisse der Beschreibung des Geschäftsvorfalles und der Abgrenzung des relevanten Problemausschnitts legen eine Gliederung des Systems »Passagierabfertigung« in drei Pakete nahe:

- ✓ Ein erstes Paket bildet die *Fluggesellschaft*. Sie ist eine organisatorische Einheit, die an dem Fallbeispiel durch ihre Mitarbeiter und ihre Flugzeuge beteiligt ist.
- ✓ Die Passagiere, die im Rahmen des Geschäftsvorfalles abzufertigen sind, sind *Kunden* der Fluggesellschaft. Alle Elemente im Einflussbereich der Kunden gehören in dieses zweite Paket. Eine spätere Ergänzung um Frachtkunden ist jederzeit möglich.
- ✓ Als drittes Paket lässt sich der *Flughafen* identifizieren. Er ist durch die Mitarbeiter der Gepäckabfertigung und die Bereitstellung der Räumlichkeiten (Abfertigungshalle, Gate etc.) an dem Geschäftsvorfall beteiligt. Die Mitarbeiter des Check-In werden in dem Fallbeispiel dem Paket Fluggesellschaft zugeordnet.

Daraus ergibt sich eine funktionale Gliederung der Pakete nach spezialisierten Rollen, die sich grafisch wie in der folgenden Abbildung darstellen lässt.

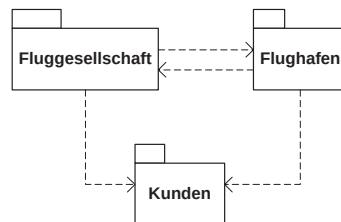


Abbildung 10.2: Pakete des Fallbeispiels »Passagierabfertigung«

Zwischen den Paketen bestehen gegenseitige Abhängigkeiten. Die Fluggesellschaft greift bei der Abfertigung auf Daten ihrer Kunden zurück. Sollte sich die Struktur dieser Daten verändern, dann sind auch notwendige Änderungen im Paket `Fluggesellschaft` zu erwarten. Gleiches gilt für den Flughafen, der u.a. für das Gepäck der Passagiere zuständig ist. Die Mitarbeiter der Fluggesellschaft kommunizieren mit den Mitarbeitern der Gepäckabfertigung. Da Interaktionen in beiden Richtungen stattfinden, handelt es sich hier um ein gegenseitiges Abhängigkeitsverhältnis.



Die Abhängigkeitsbeziehung zwischen zwei Paketen bedeutet, dass Änderungen in dem unabhängigen Paket *möglicherweise* zu Änderungen in dem abhängigen Paket führen; sie sind also nicht zwangsläufig.

Anhand der in den Checklisten genannten Fragen kann die hier vorgenommene Paketbildung überprüft werden.

- ✓ *Legt die Aufbau- oder Ablauforganisation des Geschäftsprozesses bereits Kriterien für eine Paketbildung nahe?*
- ✓ Die Beschreibung der Problemdomäne »Passagierabfertigung« lässt eine organisatorische Struktur erkennen, die zumindest die Pakete `Kunden` und `Fluggesellschaft` nahe legt.

- ✓ Inwiefern die Definition des Pakets Flughafen als dritte Komponente sinnvoll ist, wird sich im Laufe der weiteren Analyse zeigen.
- ✓ *Erlaubt die Aufgabe eine inhaltliche, prozessorientierte, lokale oder funktionale Gliederung?*
- ✓ Die vorgenommene Gliederung ist zumindest inhaltlich möglich. Ob sie auch aus Sicht der objektorientierten Analyse sinnvoll ist, muss im weiteren Verlauf geprüft werden.
- ✓ *Ist die betrachtete Aufgabe teilbar?*
- ✓ Die Aufgabe ist teilbar. Es sind mehrere Rollen aus verschiedenen organisatorischen Bereichen mit der Passagierabfertigung befasst. Die zahlreichen Aktivitäten im Rahmen des Prozesses können somit aufgeteilt werden, da in der realen Problemstellung bereits eine funktionsfähige Aufteilung existiert.
- ✓ *Handelt es sich bei dem betrachteten Element um ein isolierbares Subsystem?*
- ✓ Vor dem Hintergrund des hier eingenommenen Blickwinkels handelt es sich bei allen drei Elementen (Fluggesellschaft, Flughafen und Kunden) um isolierbare Subsysteme.



Beachten Sie, dass eine andere Sichtweise zu anderen Ergebnissen führt. Aus der Sicht des Flughafens ergibt sich eine völlig andere Struktur. Eine objektorientierte Analyse der Problemdomäne »Flughafen« würde beispielsweise u.a. die Subsysteme »Sicherheitskontrolle«, »Passagierabfertigung« und »Flugsicherung« identifizieren. Die Perspektive des Modellierers spielt also bereits bei der groben Gliederung der Problemstellung eine entscheidende Rolle.

- ✓ Können durch die Umgestaltung von Aufgaben, Prozessen und Zuständigkeiten Pakete gebildet werden, ohne die Ziele zu gefährden?
 - ✓ Diese Möglichkeit ist nicht Gegenstand der Fallstudie. Allerdings ist das Aufzeigen von Reorganisationspotenzialen eine große Stärke der objektorientierten Analyse. Häufig können so Schwächen aufgedeckt werden, deren Beseitigung auch zu einer Steigerung der organisatorischen Effizienz führt.
- ✓ Welche Beziehungen und Abhängigkeiten besitzt das betrachtete Element zu anderen Elementen innerhalb des Systems?
 - ✓ Es bestehen wechselseitige Abhängigkeitsbeziehungen zwischen den identifizierten Paketen (vgl. obige Abbildung).
- ✓ Welche Beziehungen und Abhängigkeiten besitzt das betrachtete Element zu anderen Elementen außerhalb des Systems?
 - ✓ In dieser Fallstudie wird zur Vereinfachung ein geschlossenes System unterstellt. Alle benötigten Elemente werden in die hier identifizierten Pakete integriert.
- ✓ Sind von der Integration des betrachteten Elements in ein (Teil-) System weitere Elemente in anderen (Teil-) Systemen betroffen, die ihrerseits wiederum Abhängigkeiten aufweisen?
 - ✓ Die Interdependenzen zwischen Elementen der drei Pakete werden im weiteren Verlauf der Analyse untersucht.
- ✓ Können umfangreiche Subsysteme in weitere kleine Pakete zerlegt werden?
 - ✓ Aufgrund des begrenzten Inhalts dieser Fallstudie ist eine weitere Unterteilung der Pakete zunächst nicht angebracht.

Definition von Schnittstellen zwischen Paketen

Die Schnittstellen zwischen den Paketen ergeben sich u.a. durch Beziehungen zwischen Klassen unterschiedlicher Pakete. Sie können daher erst nach der Identifikation der Klassen und ihrer Bezie-

hungen definiert werden. Gleichwohl handelt es sich um einen Vorgang innerhalb des objektorientierten Analyseprozesses, der auf relativ abstrakter Ebene anzusiedeln ist.



Auch hier zeigt sich wieder, dass die objektorientierte Analyse nicht in eine lineare Vorgehensweise eingebettet werden kann. Damit scheiden für die Objektorientierung auch die in der prozeduralen Programmierung weit verbreiteten Ansätze des Top-down- und des Bottom-up-Vorgehens aus. Hier ist vielmehr ein zyklischer Prozess angezeigt, der beide Ansätze zu einem Gegenstromverfahren mit zyklischem Charakter verbindet. Die ständige Anpassung der erzielten (Teil-) Ergebnisse ist unabdingbar.

Anhand der Fragen aus den Checklisten können an dieser Stelle einige grundsätzliche Aussagen zu den Schnittstellen der Pakete getroffen werden, die im Folgenden zu konkretisieren sind.

- ✓ Welche Kommunikationsbeziehungen bestehen zwischen den Paketen?
 - ✓ Die Verflechtungen auf Ebene der Pakete bestehen vor allem aus Ergebnissen von Prozessen.
 - ✓ Das Paket Flughafen ist, wie weiter unten noch ausgeführt wird, an dem Geschäftsprozess »Passagierabfertigung« durch die Gepäckabfertigung vertreten.
 - ✓ Nach dem Einchecken des Fluggastes ist es Aufgabe der Akteure des Pakets Flughafen, das Gepäck zum Flugzeug zu transportieren.
 - ✓ Die Pakete Kunden und Fluggesellschaft sind mit ihren Klassen an den Prozessen des Check-In und des Einsteigevorgangs beteiligt.

- ✓ *Welche Kommunikationsbeziehungen bestehen zwischen den Elementen der verschiedenen Pakete?*
 - ✓ Die Kommunikationsbeziehungen zwischen den drei Paketen sind sehr vielseitig. Insbesondere die Pakete Kunden und Fluggesellschaft tauschen über ihre Klassen eine Vielzahl von Botschaften aus. Während des Check-In benötigt der entsprechende Mitarbeiter als Akteur der Klasse Fluggesellschaft zahlreiche Informationen von Klassen des Pakets Kunden. Hierzu zählen u.a. das Gepäck, die Person des Fluggastes selbst, sein Ausweis, sein Ticket usw.
 - ✓ Beim anschließenden Einsteigevorgang kommuniziert der Kunde mit weiteren Akteuren der Klasse Fluggesellschaft (Mitarbeiter am Gate, Flugbegleiter). Hier werden erneut Informationen über die Bordkarten und die Flugscheine ausgetauscht.
 - ✓ Das Paket Flughafen ist ebenfalls beteiligt, indem es Informationen über das Gepäck des Kunden bzw. dessen Verarbeitung an den Stationsmanager übermittelt.
- ✓ *Welche Abhängigkeitsbeziehungen bestehen zwischen den Paketen?*
 - ✓ Aufgrund der zahlreichen Interdependenzen bestehen wechselseitige Abhängigkeiten zwischen den Paketen. Sie wurden bereits in der obigen Abbildung veranschaulicht.
- ✓ *Welche Abhängigkeitsbeziehungen bestehen zwischen den Klassen verschiedener Pakete?*
 - ✓ Die Abhängigkeiten zwischen den Klassen der verschiedenen Pakete ergeben sich aus den nachfolgenden Ausführungen.
- ✓ *Welche Daten werden über die identifizierten Schnittstellen ausgetauscht?*
 - ✓ Die ausgetauschten Daten wurden bereits kurz erwähnt. Eine genauere Untersuchung findet sich bei der Analyse der Sequenzen, der Zustände und Zustandsübergänge sowie der Aktivitäten des dynamischen Modells.

- ✓ *Ist durch die erforderlichen Schnittstellen eine Umstrukturierung der Pakete angezeigt?*
- ✓ Die Notwendigkeit einer Umstrukturierung ergibt sich, falls sich die Abgrenzung der Pakete als nicht sinnvoll erweist. Auch diese Frage kann erst zu einem späteren Zeitpunkt beantwortet werden.

Die Beantwortung der vorstehenden Fragen mag Ihnen an dieser Stelle ein wenig unbefriedigend vorkommen. Dies liegt daran, dass in diesem recht frühen Stadium des Analyseprozesses noch nicht genügend Erkenntnisse zur Verfügung stehen.

Dennoch ergibt es einen Sinn, die Definition der Schnittstellen zwischen den Paketen an diese Stelle des Vorgehens zu setzen, wenn man sich den evolutionären Charakter der objektorientierten Analyse vergegenwärtigt. Im Laufe späterer Zyklen ergibt sich ein ständiger Informationszuwachs, was die Art der Beziehungen zwischen Klassen und Paketen sowie deren innere Struktur angeht. Die Anpassung der Pakete gehört dann genau an diese Stelle: vor die Bearbeitung des statischen Objektmodells.

Statisches Modell

Die Entwicklung des statischen Objektmodells hängt eng mit der Bildung von Paketen mit ihren Schnittstellen zusammen. Beziehungen zwischen Klassen unterschiedlicher Pakete führen zu Abhängigkeiten zwischen Paketen. Im Rahmen der Definition der Klassen und ihrer Beziehungen stellt sich zudem heraus, ob die bisherige Gliederung der Pakete sinnvoll ist oder eventuell angepasst werden muss.

Die Entwicklung des statischen Modells gliedert sich in die Aufgaben

- ✓ Identifikation von Klassen, Attributen und Methoden,
- ✓ Beschreibung von Klassen und Attributen,

- ✓ Ermittlung von Vererbungsstrukturen und
- ✓ Identifikation von Assoziationen.

Hier zeigt sich einmal mehr der evolutionäre Charakter der objekt-orientierten Analyse. Das Modell unterliegt einer ständigen Anpassung und Erweiterung im Rahmen des Analyseprozesses. Die gliederungstechnische Differenzierung der folgenden Abschnitte ist daher keineswegs als lineare Abfolge von Analysephasen zu betrachten. Sie bildet lediglich eine Struktur der bei der Entwicklung des statischen Modells anfallenden Aufgaben.

Identifikation von Elementen und Strukturen

Als Informationsgrundlage für das statische Modell dient in diesem Fallbeispiel allein die verbale Beschreibung des Geschäftsvorfalles. Bei einer realen Problemstellung werden darüber hinaus die bereits genannten weiteren Quellen ausgewertet.

Zieht man die Checklisten zur Analyse heran, dann lassen sich folgende Elemente identifizieren, die zunächst in Form eines Brainstorming gesammelt werden:

- ✓ *Welche Akteure (Personen, Abteilungen, Unternehmungen) sind an dem Geschäftsvorfall beteiligt?*
 - ✓ Fluggast, Fluggesellschaft, Flughafen, Mitarbeiter Check-In, Gepäckabfertigung, Mitarbeiter Gate, Stationsmanager, Pilot, Flugbegleiter, Purser.
- ✓ *Welche Gegenstände werden für die zu beschreibenden Aufgaben benötigt?*
 - ✓ Ticket, Gepäck, Gepäck-Label, Ausweis, Flugzeugtür, Sitzplatz, Bordkarte, Passenger Information List (PIL), Flugschein, Warteliste.

- ✓ Welche Hardware und Software wird eingesetzt?
 - ✓ Terminal für Eingabe der Daten am Check-In, Scanner am Gate, Scanner für Gepäck-Label am Flugzeug, Datenbank zur Speicherung der Daten.
 - ✓ In dieser Fallstudie werden Hardware und Software nicht weiter berücksichtigt.
- ✓ Welche Informationen werden von den Akteuren verarbeitet?
 - ✓ Name und Status (Vielflieger, VIP) des Passagiers, Buchungsstatus (Festbuchung, Stand-by), Maße und Gewicht des Gepäcks, Abflugort, Zielort, Flugnummer, Fluggesellschaft, reservierte Klasse, Abflugzeit, Klasse und Sitznummer der Sitzplätze, Raucher- oder Nichtraucherbereich.
- ✓ An welchen Orten werden die mit den Aufgaben verbundenen Aktivitäten durchgeführt?
 - ✓ Abfertigungshalle (Check-In), Gate, Flugzeug, Parkposition (Gepäck verladen).
- ✓ Lassen sich strukturelle Abhängigkeiten zwischen einzelnen Aufgaben feststellen?
 - ✓ Abhängigkeiten bestehen zwischen den Teilprozessen des Geschäftsvorfalls (Check-In, Gepäckabfertigung, Einsteigevorgang). Die Prozesse bauen teilweise aufeinander auf, sind also abhängig von den Ergebnissen anderer Prozesse. Die Gepäckabfertigung und der Einsteigevorgang hängen vom Check-In ab. Der Einsteigevorgang ist wiederum abhängig von der Gepäckabfertigung.
- ✓ Welche (internen und externen) Ereignisse treten in dem Geschäftsmodell auf?
 - ✓ Gepäck überschreitet zulässiges Gewicht, Gepäckstück ohne Passagier an Bord, Aufruf Passagiere, Flugzeug für Einsteigevorgang bereit, Gepäck Irrläufer.

Im nächsten Schritt sind die gesammelten Informationen zu strukturieren. Als Orientierung dient dabei die Gliederung in die Pakete Kunden, Fluggesellschaft und Flughafen.

Der Fluggast bietet sich als Klasse des Pakets Kunde an. In diesen Zusammenhang gehören außerdem die Elemente Ticket, Gepäck und Ausweis. Der Fluggast besitzt einen Namen und einen Status (z.B. Vielflieger). Er führt einen Ausweis mit sich, der jedoch im Folgenden keine weitere Berücksichtigung findet.

Für das Gepäck sind Maße und Gewicht relevante Informationen bei der Passagierabfertigung. Außerdem erhält das Gepäck am Check-In ein Label. Das Gepäck wird gewogen, transportiert, gelagert, verladen, ausgeladen und überprüft. Das Label informiert über Abflug- und Zielort sowie über die Flugnummer. Diese Daten werden bei der Gepäckabfertigung über einen Scanner eingelesen.

Das Ticket enthält wichtige Informationen zum Flug. Dazu zählen der Name des Passagiers, Abflug- und Zielort, Fluggesellschaft, Flugnummer, reservierte Klasse, Abflugzeit, Status (z.B. Stand-by) und Preis. Zu dem Ticket gehört ferner ein Flugschein, der während des Einsteigevorgangs von der Fluggesellschaft eingezogen wird, um mit dem Reisebüro abzurechnen.

Alle diese Elemente, die mit Hilfe der Fragen ermittelt wurden, gehören in das Paket Kunden, für das sich nach einer ersten Analyse die folgenden Klassen ergeben.

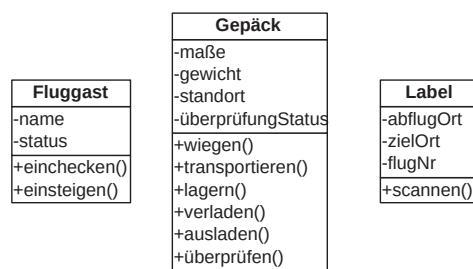


Abbildung 10.3: Klassen des Pakets Kunden

Die Klasse `Fluggast` wird beschrieben durch die Attribute `name` und `status` sowie durch die Methoden `einchecken` und `einsteigen`. Sie repräsentiert die Passagiere als Akteure des Geschäftsvorfalls. Das Attribut `status` enthält Informationen über den Fluggast und gibt beispielsweise an, ob es sich um einen Vielflieger oder VIP handelt, dem eine bevorzugte Behandlung im Rahmen der Passagierabfertigung bzw. während des Fluges gewährt wird.

Das Gepäck ist für die Fluggesellschaft vor allem durch die Eigenschaften Maße und Gewicht von Bedeutung. Sie werden durch die gleichnamigen Attribute repräsentiert. Die Methoden `wiegen`, `transportieren`, `lagern`, `verladen`, `ausladen` und `überprüfen` bezeichnen Operationen, die zur Verarbeitung des Gepäcks während der Passagierabfertigung durchzuführen sind. Dabei spielen die weiteren Attribute `standort` und `überprüfungStatus` eine Rolle. Der Standort enthält jeweils aktuelle Informationen darüber, wo sich das Gepäck auf seinem Weg vom Schalter der Fluggesellschaft zum Flugzeug gerade befindet. Das Attribut `überprüfungStatus` hält das Ergebnis der Methode `überprüfen` fest, auf das später noch näher eingegangen wird.

Jedes beim Check-In aufgegebene Gepäckstück wird laut Problembeschreibung mit einem Label versehen, das Informationen über den Abflugort, den Zielort und die Flugnummer enthält. Die benötigten Daten werden mit Hilfe der Methode `scannen` abgefragt.

Als weitere Klassen des Pakets `Kunden` können `Ticket`, `Flugschein` und `Bordkarte` identifiziert werden, die in der folgenden Abbildung dargestellt sind.

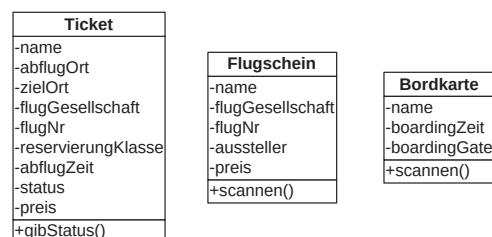


Abbildung 10.4: Weitere Klassen des Pakets `Kunden`

Die Klasse `Ticket` speichert alle Daten, die für die Reise des Fluggastes benötigt werden. Neben dem Namen des Passagiers gehören dazu der Abflugort, der Zielort, die befördernde Fluggesellschaft, die Flugnummer und die Abflugzeit. Ferner gibt das Ticket Auskunft über die gebuchte Klasse (Economy Class, Business Class, First Class), den Status sowie den Preis des Tickets. Die Methode `gibStatus` dient als Schnittstelle für entsprechende Anfragen anderer Objekte.

Ein weiteres Element des Pakets `Kunden` ist die Klasse `Flugschein`. Sie enthält ähnliche Informationen wie die Klasse `Ticket`. Sie repräsentiert die Flugscheine, die während des Einsteigevorgangs von den Mitarbeitern der Fluggesellschaft eingesammelt werden und der späteren Abrechnung mit dem Reisebüro dienen.

Die Bordkarte wird während des Einsteigevorgangs gescannt, um den zugehörigen Passagier auf die `Passenger Information List` zu setzen. Sie informiert mit ihren Attributen `boardingZeit` und `boardingGate` über Zeit und Ort des Einsteigevorgangs.

In dem Paket `Flughafen` wird für die hier zu untersuchende Problem-domäne nach der ersten Analyse lediglich eine Klasse benötigt, wie die folgende Abbildung zeigt.

Gepäckabfertigung
+gepäckVerladen() +gepäckLabelScannen()

Abbildung 10.5: Klasse des Pakets `Flughafen`

Die Klasse `Gepäckabfertigung` besitzt die Methoden `gepäckVerladen`, und `gepäckLabelScannen`. Sie repräsentiert die Mitarbeiter des Flughafens bzw. die entsprechende maschinelle Vorrichtung, die das Gepäck zum Flugzeug transportiert.

Es ist offensichtlich, dass `Gepäckabfertigung` nicht die einzige Klasse des Pakets `Flughafen` ist. Darüber hinaus können noch zahl-

reiche weitere Klassen identifiziert werden. Für das hier betrachtete Beispiel ist sie jedoch die einzig relevante Klasse.

Für das dritte Paket, die Fluggesellschaft, sind zunächst alle Objekte rund um das einzusetzende Flugzeug von Bedeutung. Dazu zählen das Flugzeug selbst, das einen Typ und eine Flugnummer besitzt, die Sitzplätze in der Maschine, die den Passagieren zugewiesen werden, die Flugzeugtüren und die Passenger Information List.

Außerdem sind die Mitarbeiter abzubilden. Dazu zählen die Flugbegleiter, die Piloten, der Purser, der Stationsmanager sowie die Mitarbeiter am Gate und am Check-In.

Es ergeben sich die in den nächsten drei Abbildungen dargestellten Klassen für das Paket `Fluggesellschaft`.

Zunächst lassen sich die folgenden Klassen zur Repräsentation des Bodenpersonals identifizieren.

Mitarbeiter Check-In	Mitarbeiter Gate	Stationsmanager
-arbeitszeitTarif -überStd -grundGehalt +gepäckWiegen() +labelAusstellen() +ausweisKontrollieren()	-arbeitszeitTarif -überStd -grundGehalt +passagiereAufrufen() +flugscheineVerarbeiten() +bordkartenVerarbeiten()	-arbeitszeitTarif -überStd -grundGehalt +überprüfenPIL() +aushändigenPIL()

Abbildung 10.6: Bodenpersonal als Klassen des Pakets `Fluggesellschaft`

Die Attribute der Klassen `Mitarbeiter Check-In`, `Mitarbeiter Gate` und `Stationsmanager` sind gleich. Sie werden im weiteren Verlauf der Analyse im Zusammenhang mit den Vererbungsstrukturen noch eingehend erläutert. Die in der obigen Abbildung enthaltenen Methoden repräsentieren das Verhalten der Mitarbeiter im Rahmen der Passagierabfertigung. Sie zeigen die Aktivitäten, die von den Mitarbeitern der Fluggesellschaft während des Check-In und des Einsteigevorgangs durchzuführen sind.

Neben dem Bodenpersonal ist auch das Flugpersonal an der Passagierabfertigung beteiligt. Während des Einsteigevorgangs treten Purser und Flugbegleiter als Akteure in Erscheinung. Inwieweit die Klasse *Pilot* für die Analyse der Problemstellung relevant ist, muss sich im Laufe der weiteren Analyse noch zeigen.

Die folgende Abbildung enthält eine grafische Darstellung der Klassen *Purser*, *Flugbegleiter* und *Pilot* in der UML-Notation.

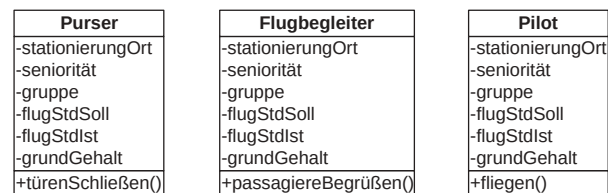


Abbildung 10.7: Flugpersonal als Klassen des Pakets Fluggesellschaft

Der obigen Abbildung ist zu entnehmen, dass die Klassen *Purser*, *Flugbegleiter* und *Pilot* weitgehend gleiche Eigenschaften haben. Hier bietet sich bereits eine Zusammenfassung in einer Oberklasse an. Die Attribute dieser Klassen haben im einzelnen folgende Bedeutung:

- ✓ **stationierungOrt:** Einsatzort des regional gegliederten Flugpersonals (»Heimatflughafen«).
- ✓ **seniorität:** Viele organisatorische Angelegenheiten des Flugpersonals werden über die Seniorität geregelt. Dazu gehören z.B. die Positionsvergabe an Bord und die Personaleinsatzplanung. Die Seniorität wird im Wesentlichen von der Dauer der Firmenzugehörigkeit und einigen weiteren persönlichen Faktoren des Mitarbeiters bestimmt.

- ✓ gruppe: Einige Fluggesellschaften gliedern ihr Flugpersonal regional nach bestimmten Flugzielen (beispielsweise Nordamerika, Asien etc.).
- ✓ flugStdSoll: Zahl der pro Monat zu leistenden Flugstunden.
- ✓ flugStdIst: Zahl der bisher geleisteten Flugstunden.

Der Stationsmanager und die Mitarbeiter am Gate und am Check-In sind gekennzeichnet durch die Attribute:

- ✓ arbeitszeitTarif: Arbeitszeit laut Tarifvertrag.
- ✓ überStd: Bisher über die tarifliche Arbeitszeit hinaus geleistete Überstunden.

Die in der Abbildung 10.7 aufgeführten Methoden ergeben sich aus dem Verhalten der beschriebenen Elemente. Sie werden später noch eingehend erläutert.

Den dritten Bereich des Pakets Fluggesellschaft bilden die Klassen, die direkt mit dem eingesetzten Flugzeug zusammenhängen. Hierzu gehören neben dem Flugzeug selbst Türen, Sitzplätze und die Passenger Information List. Die nachstehende Abbildung zeigt die entsprechenden Klassen des statischen Objektmodells.

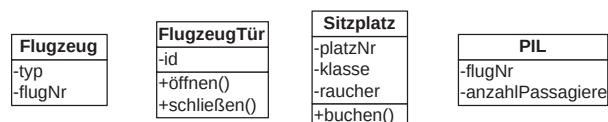


Abbildung 10.8: Flugzeugelemente als Klassen des Pakets Fluggesellschaft

Die Klasse Flugzeug besitzt entsprechend der obigen Abbildung die Attribute typ und flugNr. Weitere Eigenschaften sind für das Fallbeispiel nicht relevant. Jede Flugzeugtür besitzt eine eigene Kennzeichnung, die durch das Attribut id repräsentiert wird. Die Sitz-

plätze werden mit den Attributen `platzNr`, `klasse` und `raucher` beschrieben. Das Attribut `raucher` besitzt den Datentyp `Boolean`. Für den Wert `true` handelt es sich um einen Platz im Raucherbereich, der Wert `false` kennzeichnet einen Platz im Nichtraucherbereich. Die Klasse `PIL` repräsentiert die `Passenger Information List`. Sie enthält Informationen über die Flugnummer (Attribut `flugNr`) und die Anzahl der an Bord befindlichen Passagiere (Attribut `anzahl-Passagiere`). Auf weitere Daten der `PIL`, wie beispielsweise die Namen der Passagiere, wird hier verzichtet.

Beschreibung von Klassen und Attributen

Nachdem die beteiligten Klassen mit ihren Attributen und Methoden identifiziert wurden, sind diese im nächsten Schritt in einer verbalen Beschreibung zu dokumentieren. Die folgende Tabelle enthält beispielhaft die Beschreibung einiger der identifizierten Klassen mit ihren Attributen und Methoden.

Klasse (Ober- klassen)	Paket	Attribute (Name, Typ, Sichtbarkeit)	Methoden	Beschreibung
Fluggast	Kunden	name (String, public) status (String, public)	einchecken einsteigen	Repräsentiert die Rolle des Passagiers; Status gibt an, ob es sich z.B. um einen Vielflieger oder VIP handelt; Akteur der Prozesse Check-In und Einsteigevorgang
Label	Kunden	abflugort (String, public) zielort (String, public)	scannen	Komponente der Klasse Gepäck; enthält Informationen

Klasse (Ober- klassen)	Paket	Attribute (Name, Typ, Sichtbarkeit)	Methoden	Beschreibung
		flugnummer (String, public)		für die Beförderung des Gepäcks und seine Zuordnung zu einem Passagier
Personal	Flug- gesell- schaft	name (String, public) personalNr (String, protected) grundgehalt (float, protected)		Abstrakte Klasse; definiert ein Muster für die verschiedenen Arten von Mitarbeitern der Fluggesellschaft; Unterklassen: Bodenpersonal und Flugpersonal
Flugbe- gleiter (Flugperso- nal, Personal)	Flug- gesell- schaft	aus <i>Flugpersonal</i> : stationierungOrt (String, protected) seniorität (String, protected) gruppe (String, protected) flugstundenSoll (float, protected) flugstundenIst (float, protected) aus <i>Personal</i> : name (String, public) personalNr (String, protected) grundgehalt (float, protected)	passagiere- Begrüßen	Spezielle Ausprägung der Klasse Flugpersonal; repräsentiert die entsprechenden Mitarbeiter an Bord des Flugzeugs

Tabelle 10.1: Ausgewählte Klassenbeschreibungen

Die Inhalte der Tabelle nehmen teilweise Ergebnisse späterer Phasen des Analyseprozesses vorweg. So erfolgt z.B. die Bildung von Klassenhierarchien mit Ober- und Unterklassen, die Definition der Datentypen von Attributen und die Spezifikation der Methoden erst im weiteren Verlauf der Untersuchung.



Die Klassenbeschreibung entwickelt sich mit dem Fortschritt der Analyse immer weiter. Sie wird in einem evolutionären Prozess ständig erweitert und dem jeweiligen Kenntnisstand angepasst.

Insofern bildet die obige Tabelle das Endergebnis eines ständigen Anpassungs- und Erweiterungsprozesses.

Die Informationen der Klassenbeschreibung sind teilweise redundant. Sie finden die Klassen und ihre Strukturen auch in den Objektdiagrammen der folgenden Abschnitte.

Ermittlung von Vererbungsstrukturen

Vererbungsstrukturen können auf zwei Arten aus bestehenden Klassen abgeleitet werden. Bei der Generalisierung werden gleiche oder ähnliche Eigenschaften bereits identifizierter Klassen in Oberklassen zusammengefasst. Umgekehrt werden bei der Spezialisierung besondere Ausprägungen einer Klasse in Unterklassen konkretisiert.

Im Laufe der Identifikation von Klassen wurden oben bereits Gemeinsamkeiten im Paket `Fluggesellschaft` entdeckt. Die Klassen `Stationsmanager`, `Mitarbeiter`, `Gate` und `Mitarbeiter`, `Check-In` sowie `Flugbegleiter`, `Purser` und `Pilot` sind gekennzeichnet durch weitgehend ähnliche Attribute.

Eine mögliche Zusammenfassung dieser Eigenschaften durch Generalisierung zeigt die folgende Abbildung. Sie enthält auch bereits die Datentypen der verschiedenen Attribute.

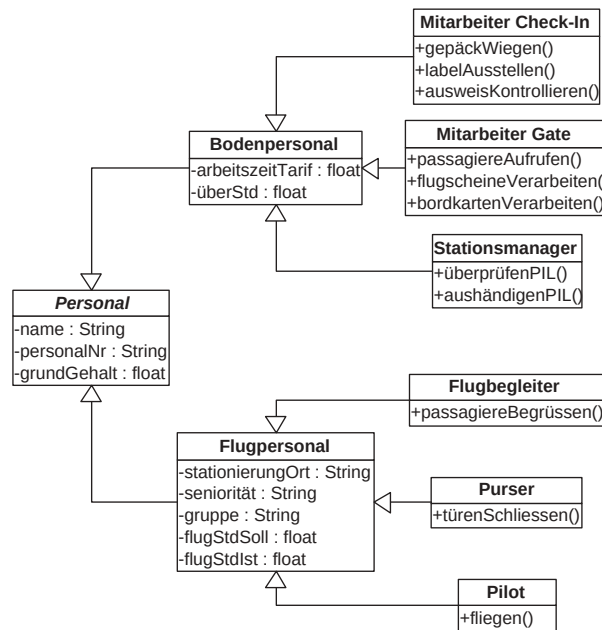


Abbildung 10.9: Personal und ihre Subklassen im Paket Fluggesellschaft

Das Personal lässt sich in Bodenpersonal und Flugpersonal gliedern. Zum Bodenpersonal gehören die Mitarbeiter von Check-In und Gate sowie der Stationsmanager. Das Flugpersonal bilden Flugbegleiter, Purser und Piloten.

Die Oberklasse Personal ist eine abstrakte Klasse, d.h. sie bildet selbst keine Objekte. Sie ist gekennzeichnet durch die Attribute name, personalNr und grundGehalt. Aus ihr werden die beiden Klassen Bodenpersonal und Flugpersonal abgeleitet.

Die Klasse `Bodenpersonal` spezialisiert ihre Oberklasse durch die Attribute `arbeitszeitTarif` (tariflich festgelegte Arbeitszeit) und `überStd`. Die Attribute der Klasse `Flugpersonal` lauten `stationierungOrt`, `seniorität`, `gruppe`, `flugStdSoll` und `flugStdIst`. `Bodenpersonal` und `Flugpersonal` werden im Hinblick auf die Erweiterungsfähigkeit des Objektmodells als konkrete Klassen definiert. Es ist durchaus vorstellbar, dass diese Klassen im Gegensatz zu `Personal` eigene Objekte instanzieren.

Aus der Klasse `Bodenpersonal` werden wiederum die drei Unterklassen `Mitarbeiter Check-In`, `Mitarbeiter Gate` und `Stationsmanager` abgeleitet. Spezielle Ausprägungen des `Flugpersonals` sind die Klassen `Flugbegleiter`, `Purser` und `Pilot`. Diese Unterklassen unterscheiden sich von ihren Oberklassen in dem Fallbeispiel durch die Methoden. Die Instanzen der Klassen zeigen verschiedene Verhaltensweisen.

Weitere Vererbungsstrukturen sind bei der gegebenen Problemstellung nicht zu erkennen.

Auch mit Hilfe der Fragen aus den Checklisten lassen sich Vererbungsstrukturen identifizieren oder vermutete Vererbungsstrukturen überprüfen. Angewandt auf die soeben beschriebene Klassenhierarchie des Personals der Fluggesellschaft ergeben sich die folgenden Überlegungen:

- ✓ *Gibt es Gemeinsamkeiten zwischen bereits identifizierten Klassen?*
 - ✓ Diese Frage zielt auf die Bildung von Generalisierungen ab. Offensichtlich bestehen Gemeinsamkeiten zwischen den Eigenschaften verschiedener Mitarbeitergruppen sowie zwischen den Mitarbeitern insgesamt.
 - ✓ Sie bestehen in dem konkreten Beispiel in den zustandsbeschreibenden Attributen `name`, `personalNr` und `grundGehalt`.
 - ✓ Darüber hinaus bestehen Gemeinsamkeiten innerhalb der Gruppen des `Bodenpersonals` und des `Flugpersonals`.

- ✓ *Besitzen bereits identifizierte Klassen spezielle Ausprägungen, die sich ihrerseits durch zusätzliche Eigenschaften auszeichnen?*
- ✓ Mit der zweiten Frage wird der umgekehrte Weg beschritten und nach Spezialisierungen bereits identifizierter Klassen gesucht.
- ✓ In dem Fallbeispiel wurde von der verbalen Problembeschreibung zu Beginn dieses Kapitels ausgegangen. Insofern wurde die Klassenhierarchie der Mitarbeiter durch Generalisierung ermittelt.
- ✓ Eine Spezialisierung könnte sich aber beispielsweise dann ergeben, wenn etwa die Klasse Mitarbeiter Gate weiter nach spezielleren Aufgaben differenziert würde.
- ✓ *Existieren in der Problemdomäne Begriffspaare, die denselben oder ähnliche Sachverhalte auf unterschiedlichen Abstraktionsstufen ausdrücken?*
- ✓ An dieser Stelle des Analyseprozesses sind solche Zusammenhänge in dem Fallbeispiel nicht zu erkennen.
- ✓ *Sind in der Problemdomäne hierarchische Strukturen erkennbar?*
- ✓ Mit dem Begriff »hierarchische Strukturen« sind hier nicht etwa Organisationshierarchien gemeint. Die Frage zielt vielmehr auf die Über- und Unterordnung von Prozessen ab.
- ✓ Denkbar ist die Bildung einer Klassenhierarchie zur Beschreibung von Teilprozessen innerhalb des betrachteten Geschäftsvorfalles. Von dieser Sichtweise wird in dem hier betrachteten Fallbeispiel jedoch abgesehen.
- ✓ *Bestehen Ähnlichkeiten zwischen den Elementen der hierarchischen Struktur?*
- ✓ Der nächste Schritt bei der Bildung von Prozesshierarchien wäre die Untersuchung auf Ähnlichkeiten in der Vorgehensweise, in den zu bewältigenden Aufgaben und Aktivitäten oder auch in den beteiligten Personen bzw. Rollen.

- ✓ *Werden alle vererbten Attribute auf der Ebene der Unterklasse tatsächlich benötigt?*
 - ✓ Diese Frage schützt den Entwickler vor der Definition von Vererbungsstrukturen, die unnötig oder nicht sinnvoll sind. Häufig gerät man in Versuchung, aus Begriffspaaren, die irgendwie zusammenzugehören scheinen, eine Vererbungsstruktur zu bilden. Oft ist es jedoch wesentlich zweckdienlicher, einfache Assoziationen oder andere Beziehungen zwischen den Klassen zu etablieren. Wenn Sie feststellen, dass die Gemeinsamkeiten einer Klasse mit einer bestehenden Oberklasse eher gering sind oder mehrere Eigenschaften nicht benötigt werden, dann entfernen Sie die Unterklasse besser aus der Klassenhierarchie.
- ✓ *Repräsentieren alle vererbten Methoden tatsächlich das Verhalten der Instanzen der Unterklasse?*
 - ✓ Auf Basis der bis hierher vorliegenden Informationen sind die Klassen in ihrem Verhalten korrekt beschrieben. Unter Umständen ergibt sich jedoch im weiteren Verlauf der Analyse eine Revision.

Identifikation von Assoziationen

Auch um Beziehungen zwischen den Klassen der Problemdomäne zu ermitteln, können Sie wieder die Fragen aus den Checklisten heranziehen.

- ✓ *Welche Elemente der Problemstellung tauschen Nachrichten aus?*
 - ✓ Ein Nachrichtenaustausch besteht zunächst zwischen den beteiligten Akteuren.
 - ✓ Die Mitarbeiter des Check-In geben die bei der Erfassung der Passagiere erhobenen Informationen an ihre Kollegen am Gate und an die Gepäckabfertigung weiter.
 - ✓ Aber auch zwischen den Klassen, die verschiedene Doku-

mente repräsentieren, und den menschlichen Mitarbeitern besteht ein Informationsaustausch.

- ✓ Bei der Suche nach Botschaften zwischen Klassen bzw. ihren Objekten werden Sie zudem in der eingesetzten Software fündig.
- ✓ *Welche Elemente bieten Dienste für andere Elemente an?*
 - ✓ So genannte Client-Server-Beziehungen bestehen in dem Fallbeispiel der Passagierabfertigung u.a. in dem Computernetzwerk der Fluggesellschaft. Die Daten der Passagiere und des eingesetzten Flugzeugs sind in der Datenbank der Fluggesellschaft gespeichert. Von ihren Terminals greifen die Mitarbeiter am Check-In und am Gate auf die Datenbank zu und können so beispielsweise ermitteln, ob noch ein Fensterplatz verfügbar ist. Die eingesetzte Hard- und Software wird in der Analyse dieses Fallbeispiels jedoch nicht weiter berücksichtigt.
- ✓ *Setzt sich ein Element aus anderen Elementen zusammen?*
 - ✓ Ein Beispiel für ein zusammengesetztes Element ist das Flugzeug. Es besteht u.a. aus Sitzplätzen, die für die Platzierung der Passagiere benötigt werden, und aus Flugzeugtüren.
 - ✓ Auch die Klasse Gepäck bietet sich für eine Aggregation oder Komposition an: Jedes Gepäckstück hat ein Label, das die Informationen über seinen Abflug- und Zielort sowie die Flugnummer enthält.
- ✓ *Lassen sich zusammengehörige organisatorische Einheiten identifizieren?*
 - ✓ Zusammengehörige organisatorische Einheiten sind z.B. die Mitarbeiter am Gate und der Stationsmanager. Sie arbeiten im Rahmen des Einsteigevorgangs zusammen und tauschen dabei ebenfalls Nachrichten aus.

- ✓ *Sind Elemente vorhanden, die bestimmte Rollen einnehmen?*
 - ✓ Ein und derselbe Mitarbeiter kann bei der Passagierabfertigung verschiedene Rollen einnehmen.
 - ✓ Denkbar ist beispielsweise, dass ein Mitarbeiter zunächst den Check-In der Passagiere vornimmt, um anschließend am Gate auch beim Einsteigevorgang der Fluggäste mitzuwirken. Ein und dieselbe Person schlüpft dann nacheinander in die Rollen Mitarbeiter Check-In und Mitarbeiter Gate.
- ✓ *Sind Beziehungen der Form hat ein, ist ein, besitzt, gehört zu, benutzt, verwaltet, plant, führt aus, kontrolliert oder beschreibt vorhanden?*
 - ✓ Ein Flugzeug hat Sitzplätze und Türen.
 - ✓ Jedes Gepäckstück hat ein Label und gehört zu einem Passagier.
 - ✓ Jeder Fluggast besitzt ein Ticket.

Im Folgenden werden beispielhaft einige Assoziationen, Aggregationen und Kompositionen herausgegriffen. Darüber hinaus existieren weitere Beziehungen, die vor allem die beteiligten Rollen betreffen.

Die Beziehungen zwischen den identifizierten Klassen sind in der Problemdomäne »Passagierabfertigung« äußerst vielfältig. Aufgabe des Modellierers ist es, die relevanten Klassenbeziehungen zu finden und zu definieren. Dabei ist auch zu entscheiden, ob es sich um einfache Assoziationen, Aggregationen oder Kompositionen handelt.

Ein erstes Beziehungsgeflecht lässt sich zwischen den Klassen Fluggast, Gepäck, Label und Ticket ausmachen. Die folgende Abbildung veranschaulicht diese Beziehungen in dem Klassendiagramm für das Paket Kunden.

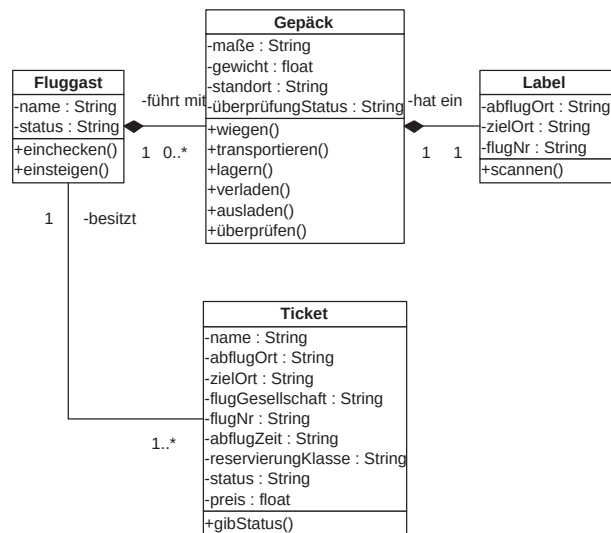


Abbildung 10.10: Klassendiagramm für das Paket Kunden

Die Eigenschaften der Klassen **Fluggast**, **Gepäck**, **Label** und **Ticket** werden als Attribute definiert. Zwischen den Klassen **Fluggast** und **Ticket** besteht eine (einfache) Assoziation, wobei jeder Kunde ein oder mehrere Tickets besitzt, jedes Ticket aber nur zu genau einem **Fluggast** gehören kann.

Die Beziehung zwischen den Klassen **Fluggast** und **Gepäck** ist eine Komposition. Ohne **Fluggast** gibt es kein **Gepäck**, umgekehrt kann aber ein **Fluggast** auch ohne **Gepäck** oder mit mehreren **Gepäck**-stücken verreisen. Jedes erfasste **Gepäck**stück wird mit einem **Label** versehen, was in dem Klassendiagramm durch die Komposition der beiden zugehörigen Klassen ausgedrückt wird. Es handelt sich um eine 1:1-Beziehung. Jedes **Label** gehört zu genau einem **Gepäck**.

stück und umgekehrt. Auch hier gilt: Ohne Gepäck existiert kein Label.

Zum Paket `Fluggesellschaft` gehören neben dem Personal, das bereits im vorangegangenen Abschnitt mit seiner Klassenhierarchie analysiert wurde, die Klassen `Flugzeug`, `Sitzplatz`, `Flugzeugtür` und die `Passenger Information List`, repräsentiert durch die Klasse `PIL`.

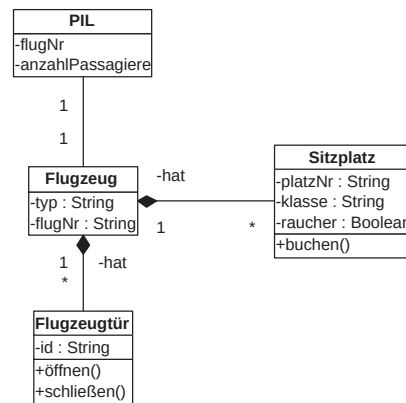


Abbildung 10.11: Die Klasse `Flugzeug` im Paket `Fluggesellschaft`

Die Klasse `Flugzeug` beschreibt die zu dem betrachteten Geschäftsvorfall gehörende Maschine mit ihren für das Fallbeispiel relevanten Bestandteilen. Hierzu gehören die Sitzplätze und die Flugzeugtüren.

Entsprechend besteht eine Komposition zwischen den Klassen `Flugzeug` und `Sitzplatz` sowie zwischen `Flugzeug` und `Flugzeugtür`. Die `Passenger Information List` bezieht sich stets auf einen bestimmten Flug, der hier zur Vereinfachung durch das `Flugzeug` repräsentiert wird.

Aus dem Paket Flughafen ist für die hier betrachtete Domäne »Passagierabfertigung« mit der Gepäckabfertigung nur eine einzige Klasse relevant.

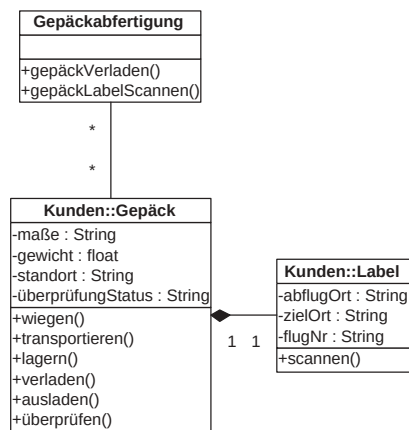


Abbildung 10.12: Die Klasse Gepäckabfertigung aus dem Paket Flughafen

Es besteht eine Assoziation zu der Klasse Gepäck aus dem Paket Kunden. Gepäckabfertigung besteht aus den Methoden `gepäckVerladen` und `gepäckLabelScannen`.

Dynamisches Modell

Nachdem im statischen Modell die Klassenstruktur erstellt wurde, wird mit dem dynamischen Modell nun das Verhalten der Objekte näher untersucht. Die einzelnen Schritte bestehen dabei in

- ✓ der Identifikation von Interaktionen,
- ✓ der Definition von Zuständen und Zustandsübergängen,
- ✓ der Identifikation von Ereignissen,
- ✓ der Identifikation von Aktivitäten und
- ✓ der konkreten Beschreibung der im Zusammenhang mit der Klassendefinition ermittelten Methoden.

Identifikation von Interaktionen

Interaktionen werden in der UML durch Sequenz- und Kollaborationsdiagramme abgebildet. Als Informationsquelle zur Identifikation von Interaktionen dient auch hier die verbale Beschreibung der Problemdomäne zu Beginn dieses Kapitels.

Im Wesentlichen besteht die Passagierabfertigung aus den drei Teilprozessen:

- ✓ Check-In
- ✓ Gepäckabfertigung
- ✓ Einsteigevorgang

Diese drei Teilprozesse sind auf Interaktionen zwischen den beteiligten Objekten zu untersuchen – und zwar sowohl innerhalb der Teilprozesse als auch prozessübergreifend.

Die bei der Erstellung des statischen Modells identifizierten Assoziationen dienen als eine erste Quelle für potenzielle Interaktionsbeziehungen. Benutzen Sie auch die übrigen Fragen, die in den Checklisten zur Ermittlung von Interaktionen aufgeführt wurden.

Stellen Sie fest, welche Objekte an einer gemeinsamen Aufgabe arbeiten. Die drei Teilprozesse der Passagierabfertigung stellen eine grobe Gliederung der Aufgaben dar, die in weitere Teilaufgaben differenziert werden können.

Eine ergiebige Quelle bei der Feststellung von Interaktionsbeziehungen ist auch die Frage, welche Objekte Informationen von anderen Objekten benötigen.

Zur näheren Beschreibung der identifizierten Interaktionsbeziehungen dienen dann die weiteren Fragen nach den Bedingungen des Informationsaustausches, den Zuständen, in denen eine Kommunikation erfolgt, sowie nach dem Zeitpunkt, wann Objekte erzeugt oder gelöscht werden.



Beachten Sie, dass es nicht sinnvoll ist, für sämtliche potenziellen Interaktionsbeziehungen in einer Problemdomäne Diagramme zu entwickeln. Lediglich die Kernprozesse oder relativ komplizierte Interaktionen sollten durch Sequenz- oder Kollaborationsdiagramme veranschaulicht werden, um Abläufe und Zusammenhänge zu verdeutlichen.

Für die Passagierabfertigung wird an dieser Stelle beispielhaft ein Sequenzdiagramm für den Check-In dargestellt. Am Check-In sind die Objekte `Mitarbeiter`, `Check-In`, `Fluggast`, `Ticket`, `Sitzplatz`, `Bordkarte`, `Label` und `Warteliste` beteiligt. Das entsprechende Sequenzdiagramm zeigt Abbildung 10.13.

Meldet sich ein Passagier beim Check-In, ermittelt der Mitarbeiter der Fluggesellschaft den Status des Fluggastes anhand des Tickets. Dazu dient die Methode `gibStatus` des Objekts `Ticket`. Das Objekt liefert als Antwort den Parameter `status`. In Abhängigkeit vom Wert dieses Parameters nimmt der Mitarbeiter bei einer Festbuchung die Buchung des Sitzplatzes vor, stellt eine Bordkarte und ein Gepäcklabel aus oder setzt Passagiere mit einem Stand-by-Ticket auf die Warteliste.

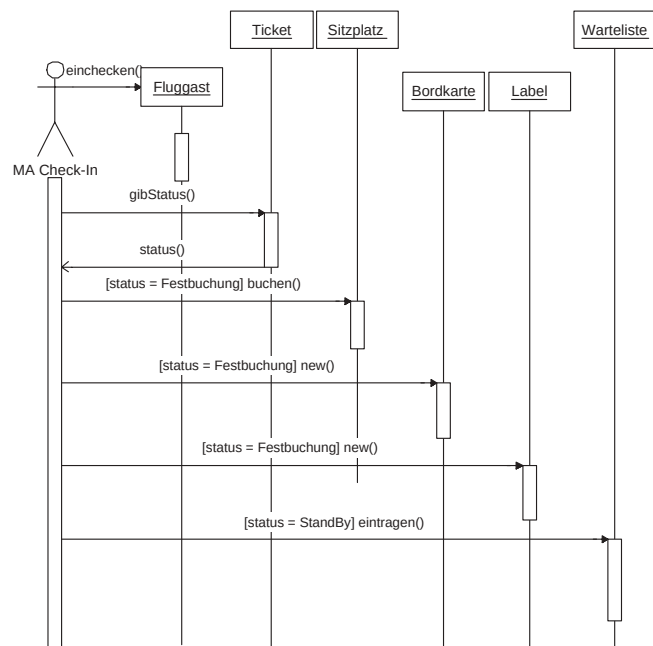


Abbildung 10.13: Sequenzdiagramm Check-In

Definition von Zuständen und Zustandsübergängen

Auch für die Zustandsdiagramme gilt der Grundsatz: Weniger ist mehr. Versuchen Sie nicht, ein umfassendes Modell des kompletten Problemausschnitts zu erstellen. Wählen Sie stattdessen die Teilbereiche aus, die einer näheren Erläuterung oder Untersuchung bedürfen.

In dem Fallbeispiel bietet sich ein Zustandsdiagramm der Gepäckabfertigung an, da hier einige Bedingungen existieren, die zu wechselnden Zuständen führen und die Abfertigung des Gepäcks auf den ersten Blick ein wenig kompliziert erscheinen lassen.

Die Fragen aus den Checklisten unterstützen bei der Entwicklung des Zustandsdiagramms der Gepäckabfertigung.

- ✓ *Wann wird ein Objekt der Klasse Gepäck erzeugt?*
 - ✓ Ein neues Objekt der Klasse Gepäck wird erzeugt, sobald ein Passagier am Check-In erscheint und ein Gepäckstück aufgibt.
- ✓ *Welche Initialwerte besitzt das Objekt?*
 - ✓ Einziger Initialwert ist der Standort des Gepäcks. Durch die Aufgabe am Check-In erhält das Attribut `standort` einen entsprechenden Wert.
 - ✓ Ansonsten wird das Objekt zunächst ohne konkrete Attributwerte initialisiert.
 - ✓ Maße und Gewicht werden erst bei der weiteren Erfassung des Gepäcks durch den Mitarbeiter der Fluggesellschaft bestimmt.
- ✓ *Wie ist der Anfangszustand des Objekts?*
 - ✓ Alle neu erzeugten Objekte der Klasse Gepäck haben zunächst denselben Zustand, der nur durch den Standort spezifiziert ist. Er lautet `aufgegeben`.
 - ✓ Erst durch das Wiegen beim Check-In und die Verknüpfung mit einem Label erhalten die Objekte weitere Informationen.
- ✓ *Welche eigenen Methoden manipulieren die Attributwerte des Objekts?*
 - ✓ In der hier vorgenommenen Abgrenzung findet eine Manipulation des Zustands zum einen durch die Methode `wiegen` statt. Nachdem das Gepäck gewogen wurde, enthält das Ob-

jekt die Information über das ermittelte Gewicht als Attributwert.

- ✓ Die Methoden transportieren, lagern, verladen und ausladen haben durch die Manipulation des Attributs Standort Einfluss auf den Zustand des Objekts vom Typ Gepäck.
- ✓ Die Methode überprüfen legt den Wert des Attributs überprüfungsStatus fest.
- ✓ *Wodurch werden diese Methoden aktiviert?*
 - ✓ Die Methode wiegen wird durch den Aufruf aus einem Objekt der Klasse Mitarbeiter Check-In ausgelöst.
 - ✓ überprüfen ist eine Methode, die durch eine entsprechende Anweisung des Stationsmanagers von den Mitarbeitern der Gepäckabfertigung ausgelöst wird.
- ✓ *Welche konkreten Ereignisse führen zu einem Zustandsübergang?*
 - ✓ Vgl. dazu die folgende Abbildung des Zustandsdiagramms.
- ✓ *Unter welchen Bedingungen führt ein Ereignis zu einem Zustandsübergang?*
 - ✓ Das Gepäck wird nur dann verladen, wenn es an seinem Bestimmungsort (das richtige Flugzeug) angekommen ist.
 - ✓ Falls zu dem bereits verladenen Gepäck kein Passagier an Bord der Maschine ist, wird das Gepäck wieder ausgeladen.
- ✓ *Kann ein und dasselbe Ereignis unterschiedliche Reaktionen und Zustandsübergänge auslösen?*
 - ✓ Liegt hier nicht vor.
- ✓ *Wie ist der Endzustand des Objekts?*
 - ✓ Ist der zu dem Gepäck gehörende Passagier an Bord der Maschine, dann lautet der Endzustand ist verladen. Ansonsten verbleibt das Gepäck nicht im Laderaum des Flugzeugs, sondern wird auf dem Flughafen gelagert. Der Endzustand lautet dann lagernd.

- ✓ *Wann endet der Lebenszyklus des Objekts?*
 - ✓ Der Lebenszyklus des Objekts endet nicht mit dem Abschluss des Geschäftsvorfalles Passagierabfertigung, da es am Ziel-flughafen weiter verarbeitet wird.
 - ✓ Erst wenn der Passagier sein Gepäck am Zielort entgegengenommen hat, wird es aus dem hier abgebildeten System entfernt.
- ✓ *Sind alle im Diagramm definierten Zustände vom Anfangszustand aus durch Ereignisse und Zustandsübergänge erreichbar?*
 - ✓ Ja. Vgl. die Abbildung des Zustandsdiagramms.

Die Ergebnisse fließen in Abbildung 10.14 des Zustandsdiagramms ein.

Zunächst wird das Gepäckstück erfasst und erreicht den Zustand `aufgegeben`. Mit diesem Zustand existiert das Objekt `Gepäck` überhaupt erst in der Problemdomäne. Anschließend wird mit der Eintrittsaktion `wiegen` der Zustand `gewogen` erreicht.

Durch ein implizites Ereignis geht der Zustand des Objekts nach in `Transport` über, der mit der Aktivität `transportieren` verbunden ist. Diese Aktivität wird ausgeführt, solange das Gepäck seinen Bestimmungsort noch nicht erreicht hat. Das Objekt verharrt in dem Zustand in `Transport`, bis das Ziel erreicht wird, und wechselt dann in den Zustand `auf Verladung wartend`.

Mit der Austrittsaktion `Label.scannen` wird der Zustandsübergang zu `gescannt` ausgelöst. Beachten Sie, dass die Methode `scannen` zu der über eine Komposition mit `Gepäck` verbundenen Klasse `Label` gehört. Falls das Gepäck fehlgeleitet wurde, geht der Zustand erneut zu in `Transport` über. Ist es am korrekten Flugzeug eingetroffen, wird über die Eintrittsaktion `verladen` der Zustand `ist verladen` erreicht. Mit Eintreten des Ereignisses `Passagier an Bord` ist das Objekt in seinem Endzustand.

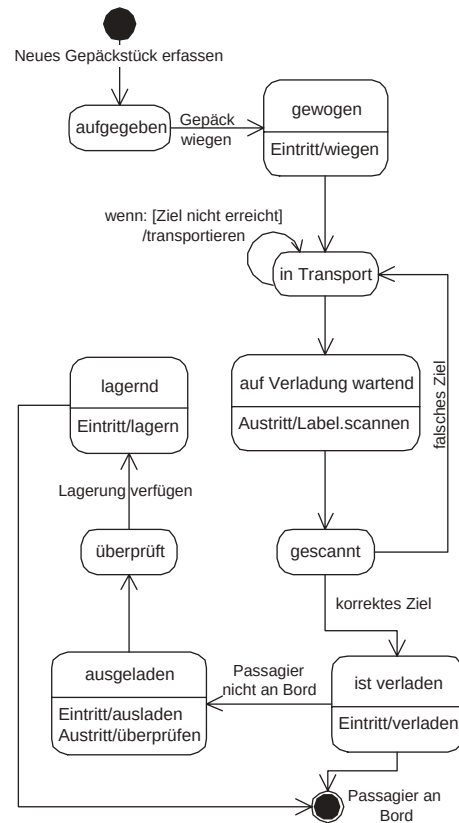


Abbildung 10.14: Zustandsdiagramm der Objekte vom Typ Gepäck

Lautet das Ereignis jedoch *Passagier nicht an Bord*, dann führt die Eintrittsaktion *ausladen* zum Zustand *ausgeladen* und anschließend die Austrittsaktion *überprüfen* zum Zustand *überprüft*. Daraufhin wird die Lagerung verfügt und das Objekt Gepäck erreicht mit dem Eintrittsereignis *lagern* den Zustand *lagernd*.

Im Rahmen der Erstellung von Zustandsdiagrammen stößt man häufig auf Methoden der betrachteten Objekte, die bisher nicht berücksichtigt wurden. Aus dem Zustandsdiagramm in der obigen Abbildung ergeben sich für die Objekte vom Typ Gepäck die Methoden wiegen, transportieren, verladen, ausladen, überprüfen und lagern. Die Methode scannen gehört zu der Klasse Label. Diese Methoden sind bereits weiter oben bei der Klassenbeschreibung aufgeführt.

Zur Ermittlung weiterer Zustände ziehen Sie wieder die Checklisten heran. Spätestens bei der Definition der Zustände und Zustandsübergänge werden Sie feststellen, dass Sie im Verlauf des objektorientierten Analyseprozesses die Problemstellung ständig aus einem anderen Fokus betrachten. Zunächst gehen Sie von der allgemeinen Problembeschreibung aus und teilen die Domäne in Pakete auf. Anschließend dringen Sie dann immer tiefer in die statischen und dynamischen Strukturen dieser Pakete ein. Bei der Beschreibung der Zustände betrachten Sie dann einzelne Objekte in ihrem Lebenszyklus.



Es erfolgt also eine ständige Verschiebung der Betrachtungsweise vom Abstrakten zum Konkreten, von einer Gesamtschau zur Detailbetrachtung und vom einzelnen Objekt zu kompletten Strukturen oder Abläufen. Diese Phasen sind eingebettet in einen evolutionären Prozess der ständigen Verfeinerung und Anpassung.

Bedenken Sie, dass es sich bei der objektorientierten Analyse um einen evolutionären Prozess handelt. Jeder neue Blickwinkel, den Sie im Laufe des Modellierungsprozesses einnehmen, ergibt auch neue Erkenntnisse für vorangegangene Phasen. So hat das dynamische Modell Auswirkungen auf die Struktur des statischen Modells und umgekehrt.

Die zahlreichen Interdependenzen zwischen den unterschiedlichen Blickwinkeln, vertreten durch die verschiedenen UML-Diagram-

me, führen zu ständigen Anpassungen und Erweiterungen, bis ein Objektmodell für eine gegebene Problemstellung vollständig und in sich konsistent ist.

Identifikation von Ereignissen

Einige Ereignisse wurden bereits bei der Ermittlung von Zuständen und Zustandsübergängen identifiziert. Hierzu zählen die Ereignisse falsches Ziel, korrektes Ziel, Passagier an Bord und Passagier nicht an Bord. Da Zustandsübergänge durch Ereignisse ausgelöst werden, liegt hier eine besonders ergiebige Quelle für die Ermittlung von Ereignissen.

Einige ausgewählte Ereignisse enthält die folgende Aufstellung.

Ereignis	Beschreibung
Gepäck überschreitet zulässiges Gewicht	Konsequenz: Passagier muss Gebühr entrichten
Flugzeug für Einsteigevorgang bereit	Aktion: Passagiere gehen zum Gate
Sitzplätze frei	Aktion: Passagiere auf Warteliste werden aufgerufen
Alle Passagiere an Bord	Aktion: Türen schließen

Tabelle 10.2: Ausgewählte Ereignisse

Mit Hilfe der Checklisten und der verbalen Beschreibung der Problemstellung zu Beginn dieses Kapitels können weitere Ereignisse gefunden werden.

Die Identifikation von Ereignissen ist weitgehend in die Beschreibung der Zustände und Zustandsübergänge eingebunden. Im Rahmen dieser Überlegungen ist zu prüfen, aufgrund welcher Ereignisse ein Zustandsübergang erfolgt. Zudem müssen Sie sich bei der

Beschreibung der Zustände Gedanken darüber machen, welche Ursachen ein Ereignis hat und welche Wirkungen es entfaltet. Diese äußern sich dann in einem Ausgangs- und einem Zielzustand.

Identifikation von Aktivitäten

Aktivitäten konkretisieren die internen Aktionen von Objekten. Sie dienen der Beschreibung einzelner Verarbeitungsschritte innerhalb eines Ablaufs.

Nachdem der Prozess der Gepäckabfertigung mit Hilfe eines Zustandsdiagramms analysiert wurde, soll in diesem Abschnitt der Einsteigevorgang beispielhaft auf Aktivitäten untersucht werden. Dazu werden die in den Checklisten genannten Fragen benutzt.

- ✓ Welche Verantwortlichkeiten (swimlanes) können unterschieden werden?
 - ✓ Am Einsteigevorgang sind die Klassen Mitarbeiter Gate und Stationsmanager beteiligt. Sie besitzen unterschiedliche Aufgaben und arbeiten gemeinsam an der Abfertigung der Passagiere. Es bietet sich daher an, diese beiden Verantwortlichkeitsbereiche zu unterscheiden.
 - ✓ Als dritter Akteur spielt auch die Gepäckabfertigung eine Rolle. Sie ist jedoch nicht an dem Kernprozess selbst beteiligt, sondern sorgt lediglich unter bestimmten Bedingungen für das Ausladen des Gepäcks (bei Abwesenheit des zugehörigen Passagiers).
- ✓ Von welchen Bedingungen hängt die Ausführung der Prozesse ab?
 - ✓ Einige der Aktivitäten innerhalb des Prozesses »Einsteigevorgang« sind von Bedingungen abhängig. So wird etwa die Passenger Information List erst dann an den Purser weitergegeben, wenn alle Passagiere an Bord sind und insbesondere zu jedem Gepäckstück ein Fluggast an Bord der Maschine ist.

- ✓ Welche alternativen Zustandsübergänge sind mit einer Aktivität verknüpft?
 - ✓ Besondere Bedeutung hat für den Einsteigevorgang der Zustand des Gepäcks. Der Übergang von *ist verladen* nach *ausgeladen* wird durch das Fehlen des zu dem jeweiligen Gepäckstück gehörenden Passagiers ausgelöst.
- ✓ Führt ein Übergang zu mehreren verschiedenen Aktivitäten (*Splitting*)?
 - ✓ Ein Splitting ist bei der gleichzeitigen Durchführung mehrerer Aktivitäten erforderlich. Während des Einsteigevorgangs tritt dies bei der Abgabe der Bordkarten und der Flugscheine durch den Passagier auf.
- ✓ Werden mehrere Aktivitäten zur einer Folgeaktivität zusammengeführt (*Synchronisation*)?
 - ✓ Die Aktivitäten zur Bearbeitung von Bordkarten und Flugscheinen werden nach der Ablage der Flugscheine und der Aushändigung der Bordkarten an die Passagiere wieder zusammengeführt.
- ✓ Ist es sinnvoll, einzelne Objektzustände in das Aktivitätsdiagramm aufzunehmen, um Zusammenhänge zu verdeutlichen?
 - ✓ In diesem relativ gut überschaubaren Teilprozess ist die Aufnahme von Zuständen in das Aktivitätsdiagramm nicht notwendig.
- ✓ Welche eingehenden und ausgehenden Signale lassen sich identifizieren?
 - ✓ Signale werden mit den Instanzen der Klasse *Gepäck* ausgetauscht, falls ein Passagier, der sein Gepäck aufgegeben hat, nicht erscheint. Der Einsteigevorgang ist erst dann beendet, wenn dessen Gepäck wieder ausgeladen wurde.

Das folgende Aktivitätsdiagramm beschreibt die Arbeitsschritte des Einsteigevorgangs.

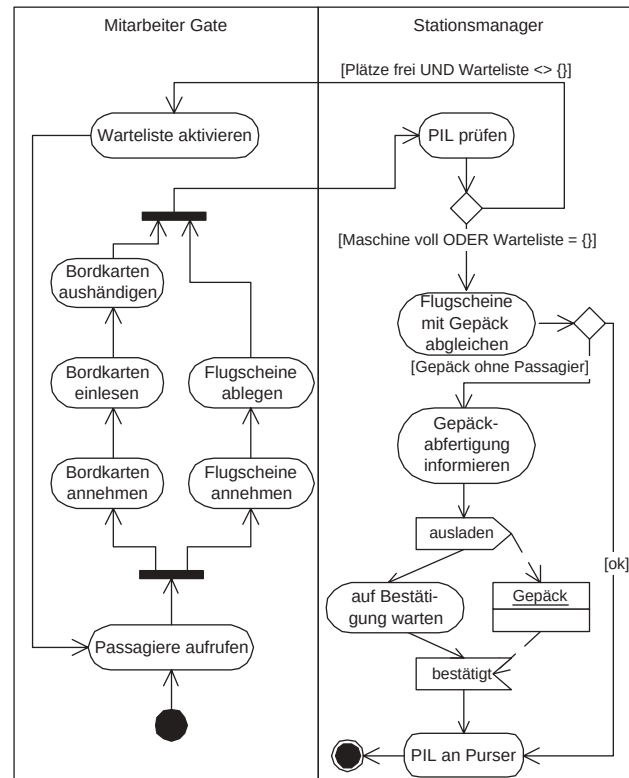


Abbildung 10.15: Aktivitätsdiagramm Einsteigevorgang

Die Abbildung teilt sich in die beiden Verantwortungsbereiche *Mitarbeiter Gate* und *Stationsmanager*. Zu Beginn des Einsteigevorgangs werden die Passagiere aufgerufen. Es folgt ein Split in zwei Aktivitätsfolgen. In der einen Folge werden die Flugscheine angenommen und zur späteren Abrechnung mit den Reisebüros abge-

legt. Die zweite Aktivitätsfolge beschreibt die Verarbeitung der Bordkarten. Die Mitarbeiter nehmen die Bordkarten entgegen, lesen sie ein und händigen sie anschließend den Passagieren wieder aus.

Nach Abschluss dieser beiden Aktivitätsfolgen prüft der Stationsmanager die Passenger Information List. Sind alle Angaben in Ordnung, kann die PIL an den Purser weitergereicht werden und die Aktivitäten im Rahmen des Einsteigevorgangs sind beendet.

Sollte sich bei der Überprüfung jedoch herausstellen, dass Gepäck verladen wurde, zu dem kein Passagier an Bord der Maschine ist, dann erfolgt ein Signal an das Objekt Gepäck mit der Anweisung, das entsprechende Gepäckstück wieder auszuladen. Sobald das Signal zur Bestätigung des Ausladens eintrifft, geht dann die PIL an den Purser.

Stellt sich als drittes mögliches Ergebnis bei der Überprüfung der PIL heraus, dass alle Passagiere mit Festbuchung an Bord sind, die Maschine aber noch nicht vollständig besetzt ist, wird die Warteliste aktiviert. Die Aktivitäten vom Aufruf der Passagiere bis zur Weitergabe der PIL an den Purser wiederholen sich nun für die Warteliste.

Beschreibung von Methoden

Der letzte Schritt bei der Erstellung des dynamischen Modells ist die Beschreibung der Methoden. Bisher wurden die Methoden lediglich mit ihren Namen und ihren groben Aufgaben identifiziert. Sie sind nun noch in ihrer Struktur und in ihrem internen Ablauf zu spezifizieren.

Wichtige Informationen zur Definition einer Methode sind neben dem Namen ihre Parameter, Zweck, Datentyp, Vorgehensweise und Ergebnis. Am Beispiel der Methode `verladen`, die zum Verhalten des Objekts `Gepäck` gehört, wird die Beschreibung von Methoden verdeutlicht.

- ✓ Name der Methode: verladen
- ✓ Klasse: Gepäck
- ✓ Paket: Kunden
- ✓ Parameter: Lagerort
- ✓ Datentyp: Es handelt sich um eine typlose Methode. Es wird kein Wert an das aufrufende Objekt zurückgeliefert.
- ✓ Verwendete Methoden anderer Klassen: Methode `Ladeluke` öffnen der Klasse `Ladeluke`. Entsprechende Methode eines menschlichen Akteurs, der das Gepäck verlädt.
- ✓ Vorgehensweise: `Ladeluke` öffnen, Gepäckstück von Transportband nehmen, Gepäckstück in Laderaum heben, Gepäckstück an Lagerort ablegen.
- ✓ Ergebnis: Gepäck ist verladen; kein Rückgabewert.

Die Beschreibung der Methode `verladen` wird durch die Beantwortung folgender Fragen mit Inhalt gefüllt.

- ✓ *Welchen Zweck hat die Methode?*
 - ✓ Die Methode beschreibt den Vorgang des Verladens eines Gepäckstücks im Laderaum des Flugzeugs.
- ✓ *Ist der in der Klassenbeschreibung für die Methode gewählte Name aussagekräftig?*
 - ✓ Der gewählte Name drückt genau die Aufgabe der Methode aus.
- ✓ *Wodurch wird die Methode ausgelöst?*
 - ✓ Die Methode wird ausgelöst, sobald das Gepäck gescannt wurde und bestätigt ist, dass es an das richtige Flugzeug geliefert wurde (vgl. das obige Sequenzdiagramm).

- ✓ Welche Daten benötigt die Methode zur Durchführung ihrer Aufgabe?
 - ✓ Die Methode verladen benötigt den Lagerort innerhalb des Laderaums des Flugzeugs. Er wird beim Aufruf der Methode als Parameter übergeben.
- ✓ Wie führt die Methode ihre Aufgabe aus?
 - ✓ Da das Gepäck sich nicht selbst verladen kann, muss ein entsprechender Dienst eines menschlichen Akteurs in Form einer Assoziation eingerichtet werden.
- ✓ In welcher Reihenfolge und unter welchen Bedingungen werden bestimmte Teilaktivitäten (z.B. Berechnungen) durchgeführt?
 - ✓ Vgl. die Beschreibung der Vorgehensweise in der Methodenbeschreibung.
- ✓ Auf welche internen Attribute wird bei der Bearbeitung zugegriffen?
 - ✓ Die Attribute maße und gewicht des Gepäcks spielen eine Rolle bei der Wahl eines geeigneten Lagerplatzes.
- ✓ Welche internen Attribute werden manipuliert?
 - ✓ Eine Manipulation von Attributen findet nicht statt.
 - ✓ Denkbar ist jedoch, den Objekten der Klasse Gepäck ein Attribut mitzugeben, das ihren aktuellen Stand- oder Lagerort enthält. Dieses Attribut wird dann beim Verladen entsprechend geändert.
- ✓ Sendet die Methode eine Antwort an das aufrufende Objekt?
 - ✓ Nein.
- ✓ Übermittelt die Methode ihre (Zwischen-) Ergebnisse an andere Objekte?
 - ✓ Nein.
- ✓ In welchem Bereich ist die Methode sichtbar?
 - ✓ Die Methode verladen muss nur innerhalb des Pakets Flughafen sichtbar sein.

Zur exakten Definition von Methoden kann auch eine formale Sprache, wie beispielsweise eine Programmiersprache, verwendet werden. Allerdings sollte dies im Rahmen der objektorientierten Softwareentwicklung erst relativ spät geschehen, wenn bereits feststeht, welche Programmiersprache bei der Implementierung zum Einsatz kommt. In der Regel wird dies erst im Laufe des objektorientierten Design festgelegt.





KAPITEL

Werkzeuge der objekt-orientierten Modellierung

In diesem Kapitel stellen wir Ihnen einige Werkzeuge der objektorientierten Modellierung vor. Sie finden sie zum größten Teil auch als Evaluationsversionen oder Vollversion auf der beiliegenden CD-ROM.

11



Werkzeuge der objektorientierten Modellierung

Wir haben uns bemüht, Ihnen eine Auswahl guter und in der Praxis weit verbreiteter Werkzeuge zur Verfügung zu stellen. Mit Hilfe der Evaluationsversionen oder Vollversion erhalten Sie einen Eindruck von den jeweils verfügbaren Funktionen und der Arbeit mit der Software.

Es würde allerdings den Rahmen dieses Taschenbuchs sprengen, die Handhabung der Tools ausführlich zu erläutern. Wir beschränken uns daher auf Hinweise zur Installation der Software (vgl. Anhang B) und auf kurze Erläuterungen zu den Benutzungsoberflächen, die Sie in den folgenden Abschnitten finden.

Der letzte Abschnitt dieses Kapitels fasst die Eigenschaften der hier vorgestellten UML-Werkzeuge in einer Übersicht zusammen.

Wenn Sie sich intensiver mit einem Tool auseinander setzen möchten, sollten Sie das jeweilige Tutorial der Hersteller zu Rate ziehen. Dort erhalten Sie ausführliche Erklärungen der Programmfunktionen. Auf den Internetseiten der Hersteller, deren Adressen in Anhang C aufgelistet sind, finden Sie weitere nützliche Informationen.

Together

TogetherSoft bezeichnet sein Tool Together als *Model-Build-Deploy Platform*. Das Werkzeug unterstützt den Prozess der Software-Entwicklung von der Modellierung von Fachmodellen, Benutzungsschnittstellen und Datenhaltungsschnittstellen über die Programmerstellung von Benutzungsschnittstellen und Datenbankverbindungen bis hin zur Konfiguration verteilter Komponenten.

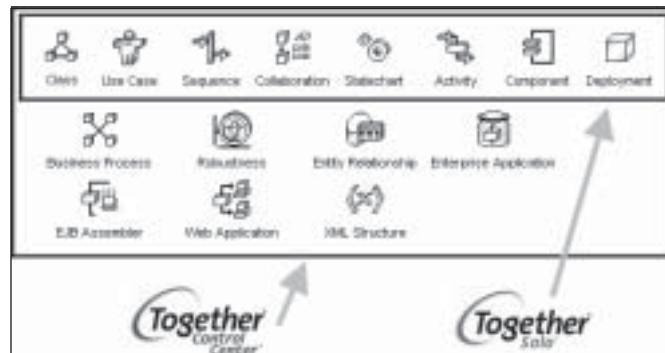
Together unterstützt alle UML-Diagramme der Version 1.3 sowie deren Umsetzung in verteilte, webbasierte Anwendungen.

Hervorstechendes Merkmal im Bereich der UML-Modellierung ist die Single-Source-Technologie, bei der UML-Modell und Programmcode zu einer Einheit verschmelzen. Ergebnis ist die simultane Aktualisierung von UML-Klassendiagrammen und Programmcode in beide Richtungen.

Der Hersteller hebt die folgenden Eigenschaften besonders hervor.

- ✓ Simultanes Roundtrip Engineering für Java, C#, C++, IDL, Visual Basic
- ✓ Unterstützung der .NET-Technologie
- ✓ GUI-Builder
- ✓ Debugging
- ✓ Datenbank-Modellierung mit Roundtrip Engineering
- ✓ Plattformunabhängiges Arbeiten
- ✓ Möglichkeit der Integration aller wichtigen Entwicklungsumgebungen
- ✓ Unterstützung der gängigen Versionskontrollsysteme
- ✓ UML 1.3 und Entity Relationship Diagramm
- ✓ Patterns und Components (inkl. EJBs)
- ✓ Offene Java-API
- ✓ Multi-Level Dokumentations-Generierung für HTML, RTF und Javadoc2
- ✓ Metriken und Audits
- ✓ View-Management und leistungsfähige Autolayout-Funktionen

TogetherSoft bietet zwei Versionen seines Werkzeugs an. *Together Solo* unterstützt die Modellierung aller UML-Diagramme und Programmiersprachen.

**Abbildung 11.1:** Together-Versionen und Diagramme

Die Ihnen zur Verfügung stehende Version *Together Control Center* unterstützt weiter gehende Aspekte wie die Geschäftsprozess-Modellierung, Datenbankmodellierung sowie die Entwicklung verteilter Anwendungen mit Entity Java Beans (EJB) im Zusammenspiel mit diversen Enterprise Application Servern.

Im Kontext dieses Buches stehen die grundlegenden Eigenschaften zur UML-Modellierung im Vordergrund. Daher können wir auf die fortgeschrittenen Eigenschaften hier nicht weiter eingehen.



Beachten Sie die Informationen im Anhang B zur Installation und Einrichtung des Lizenzschlüssels.

Einrichten eines Projekts

Starten Sie Together, nachdem Sie das Programm installiert und freigeschaltet haben, über das Windows-Startmenü *Start/Programme/Together 6.0/Together*.

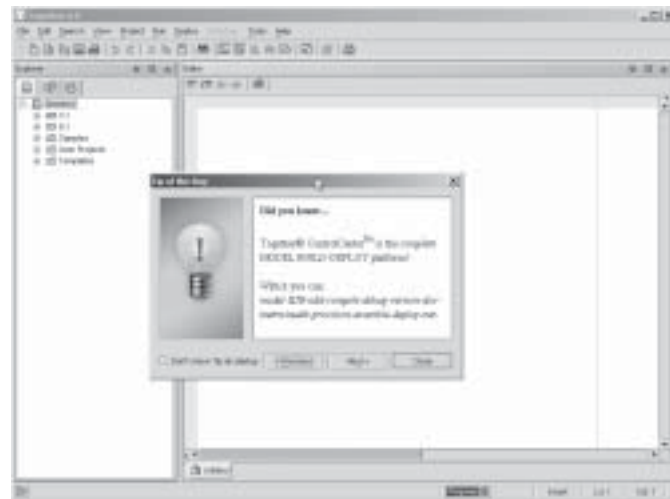


Abbildung 11.2: Hauptfenster

Nachdem Sie mit *Close* den *Tip of the Day* beendet haben, beginnt jede Modellierung in Together mit der Definition eines Projektes. Alle Einstellungen für die Benutzungsoberfläche, die UML-Diagramme und der erstellte Quellcode sind Bestandteil dieses Projektes.

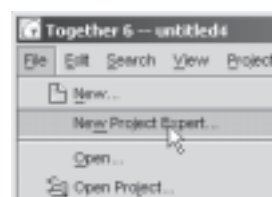


Abbildung 11.3: Projekt einrichten

Ein Assistent fragt Schritt für Schritt die notwendigen Einstellungen für ein Projekt ab. Geben Sie als Projektnamen *Anlage* und als Programmiersprache *Java* an.

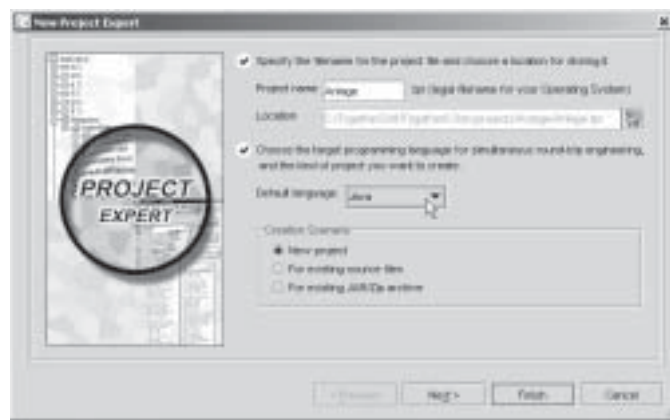


Abbildung 11.4: Projekt-Optionen

Über die *Next*-Schaltfläche gelangen Sie in einen Dialog, falls das gewählte Projektverzeichnis noch nicht existiert. Bestätigen Sie mit *Yes*, dass das Verzeichnis auf der Festplatte angelegt werden soll.



Abbildung 11.5: Projektverzeichnis anlegen

Nach erfolgreicher Beendigung des Dialogs nimmt der Assistent seine Arbeit wieder auf, um weitere Optionen zu erfragen. Über-

nehmen Sie alle weiteren Optionen mit den voreingestellten Werten, indem Sie den Assistenten durch einen Klick auf die Schaltfläche *Finish* beenden.

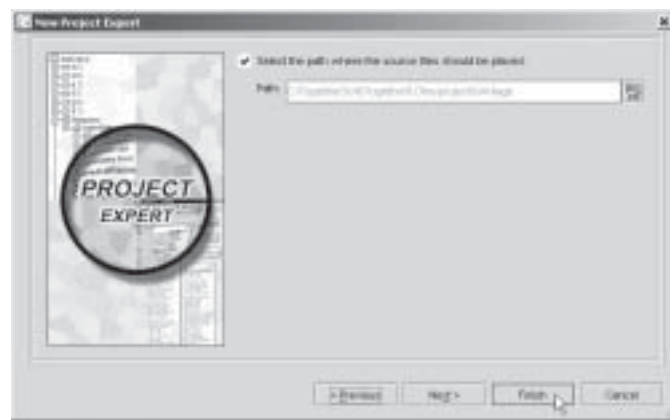


Abbildung 11.6: Projekt anlegen

Das soeben erzeugte Projekt präsentiert sich im Together-Hauptfenster mit drei unterteilten Bereichen. Diese für die bei der Installation eingestellte Developer-Sicht typische Einstellung kann durch eine Reihe von Tuning-Optionen individuell angepasst werden. Sollte Ihre Darstellung von der hier gezeigten abweichen, so liegt das an divergierenden Einstellungen.

Das Fenster zeigt die drei Bereiche *Explorer*, *Designer* und *Editor* (siehe Abbildung 11.7).

Der Designer ist für UML-Modellierer das zentrale Werkzeug. Hier werden alle UML-Diagramme grafisch erstellt. Das Zeichnen ähnelt von der Handhabung her dabei grundsätzlich einem gängigen Grafikprogramm.

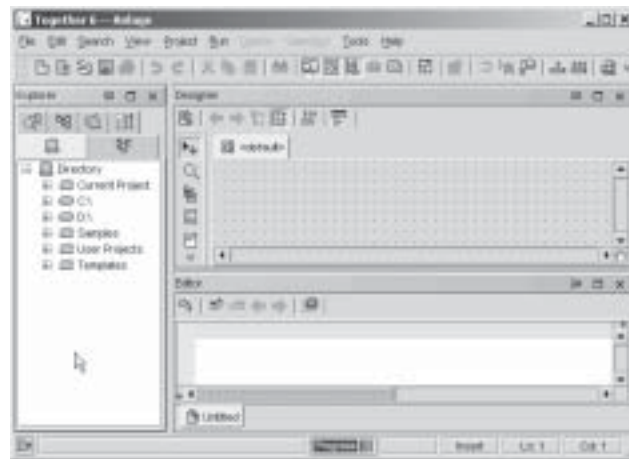


Abbildung 11.7: Hauptfenster in der Developer-Sicht

Der Explorer dient zur Navigation in verschiedenen Bereichen. Sie können in Verzeichnissystemen Dateien suchen, in der Hierarchie der von Ihnen erstellten Diagramme und Elemente navigieren oder durch Testfälle browsen, mit denen Sie Ihre Anwendung überprüfen.

Im Editor bearbeiten Sie den Quellcode in der Programmiersprache, die Sie für Ihr Projekt aus dem zur Verfügung stehenden Kanon ausgewählt haben.



Auch wenn Together mehrere Programmiersprachen unterstützt, so bietet sie für die Programmiersprache Java doch die umfassendsten Funktionen. Das zeigt sich auch bei der Umsetzung von Assoziationen, Aggregationen und Kompositionen mit Hilfe von Patterns, die entsprechende Attribute und Operationen vorgeben.

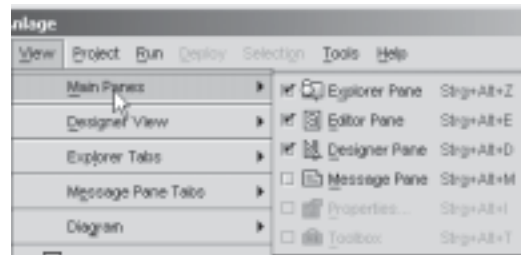


Abbildung 11.8: View konfigurieren

Im Menü *View/Main Panes* haben Sie die Möglichkeit, Bereiche komplett ein- oder auszublenden.

Erstellen von Klassendiagrammen

Im Folgenden wählen wir exemplarisch das als zentrales Diagramm in unserem Projekt festgelegte Klassendiagramm, um die simultane UML-Diagramm- und Code-Erzeugung zu betrachten, die Together so bekannt und beliebt gemacht hat (siehe Abbildung 11.9).

Links neben dem Zeichenfeld, in dem Sie Elemente des Klassendiagramms anordnen können, befinden sich die Zeichenwerkzeuge. Die Symbole repräsentieren UML-Modellelemente. *Paket*, *Klasse*, *Vererbung* und *Assoziation* sind als grundlegende Vertreter des Klassendiagramms in der Abbildung mit einem Pfeil gekennzeichnet. Über das Assoziations-Symbol können Sie sowohl einfache Assoziationen als auch Aggregationen und Kompositions-Beziehungen in Ihr Modell einfügen.

Um ein Element im Zeichenfeld zu positionieren, klicken Sie zunächst auf das entsprechende Symbol in der Werkzeugleiste. Damit haben Sie das Symbol ausgewählt und können mit einem weiteren Klick auf der Zeichenfläche das Element als Modellelement erzeugen und an einer beliebigen Position auf dem Zeichenfeld darstellen (siehe Abbildung 11.10).



Abbildung 11.9: Zeichenwerkzeuge im Klassendiagramm



Abbildung 11.10: Simultanes Roundtrip Engineering

Wählen Sie das Klassensymbol aus und klicken Sie anschließend in das Zeichenfeld, um eine neue Klasse zu erzeugen und im Klassendiagramm darzustellen.

Im nächsten Schritt klicken Sie in das erste Segment (Klassennamen) der grafisch dargestellten Klasse auf dem Zeichenfeld und geben Sie den Klassennamen `Konto` ein.

Drücken Sie die Tastenkombination `[Strg] + [A]`, fügen Sie der markierten Klasse ein neues Attribut zu. Geben Sie exemplarisch das Attribut `kontostand:double` ein.

Mit der Tastenkombination `[Strg] + [O]` erhält das markierte Objekt eine neue Operation. Vereinbaren Sie hier die Operation `einzahlen` mit `void` als Rückgabety (das heißt, es gibt keinen Rückgabewert).

Beim Einfügen von Attributen und Operationen erkennen Sie, dass Together gleichzeitig mit dem Einfügen von Klassen, Attributen und Operationen den Programmcode in der Sprache Java generiert und im Editor auch sofort anzeigt.



Abbildung 11.11: Attribute und Operationen vereinbaren

Die Abbildung zeigt für das Attribut `kontostand` und die Operation `einzahlen(...)` die beiden simultan erzeugten Symbole im Klassendiagramm und die korrespondierenden Quellcodezeilen.



Roundtrip-Engineering bedeutet, dass Änderungen im Quellcode sich genauso auf das UML-Modell und seine grafische Darstellung durchschlagen, wie auch eine Änderung im UML-Element umgekehrt auf den Quellcode. Simultan bedeutet, dass die Änderungen gleichzeitig stattfinden.

Editieren Sie Änderungen von UML-Elementen wie Klassen, Attribute und Operationen alternativ im Designer-View direkt im Klassensymbol oder aktivieren Sie bei markierter Klasse über **Alt** +  den Property-Inspektor. Die dritte Möglichkeit zum Editieren ist der Quellcode. Sie sehen, jede Änderung in einem der Bereiche schlägt sich auf die drei anderen Bereiche sofort nieder.



Abbildung 11.12: Property-Inspektor

Fügen Sie eine weitere Klasse mit dem Namen Sparkonto hinzu. Wählen Sie anschließend in der Werkzeugleiste das Vererbungssymbol aus und definieren Sie eine Vererbungsbeziehung zwischen dem Sparkonto (Unterklasse) und dem Konto (Oberklasse), indem Sie zunächst auf das Sparkonto und anschließend auf das Konto klicken (siehe Abbildung 11.13).

Auch die Vererbungsbeziehung, die Sie nach Anwahl des Vererbungssymbols in der Werkzeugleiste von der Klasse Sparkonto zur Klasse Konto gezeichnet haben, ist simultan im Editorfenster als extends-Statement, das in der Sprache Java eine Vererbungsbeziehung zu einer Oberklasse definiert, sichtbar.

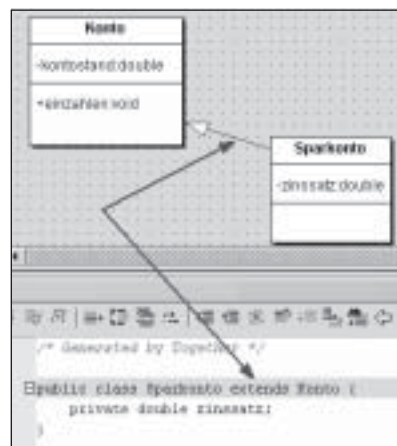


Abbildung 11.13: Vererbungsbeziehung

Fügen Sie zum Abschluss noch eine Klasse Kunde und eine Assoziation Inhaber vom Kunden zum Sparkonto ein. Nachdem Sie das Assoziations-Symbol in der Werkzeugleiste gewählt haben, klicken Sie auf die Klasse Kunde und dann auf die Klasse Sparkonto.



Abbildung 11.14: Assoziationsbeziehung

Solange die Assoziation auf der Zeichenfläche ausgewählt ist, können Sie im Property-Inspektor die Eigenschaften der Assoziation bearbeiten. Vereinbaren Sie das *label*, die *label direction*, die *client cardinality* und die *supplier cardinality* gemäß den Vorgaben.

Sie sehen auch hier die simultane Änderung von UML-Modell und Programmcode. Die Assoziation ist als unidirektionale Beziehung über das Attribut `InkSparkkonto` vom Typ `Sparkkonto` in Zeile 10 des Quellcodes implementiert. Ein solches einfaches Attribut vom Typ der bezogenen Klasse kann nur maximal ein Objekt referenzieren.

Die Kardinalitäten werden hingegen nur als erläuternde Kommentare im Quellcode hinterlegt. Aus diesen Kommentaren konstruiert Together das UML-Diagramm. Um sicherzustellen, dass die Kardinalitäten jederzeit eingehalten werden, müssen Sie speziellen Code schreiben.



Together bietet für die Sprache Java eine Reihe von Patterns, mit denen Assoziationen abgebildet werden können, die eine Beziehung zu mehreren Objekten beinhalten, geordnet sind oder andere spezielle Anforderungen besitzen.

Vom Klassen- zum Objektdiagramm

Die zeitunabhängige Struktur der Anwendung, die sich im Klassendiagramm widerspiegelt, kann in einem Objektdiagramm exemplarisch überprüft werden. Together bietet die Möglichkeit, ein Objektdiagramm in ein Klassendiagramm zu integrieren. Die Zeichenwerkzeuge, die Sie dazu benötigen, zeigt die folgende Abbildung.

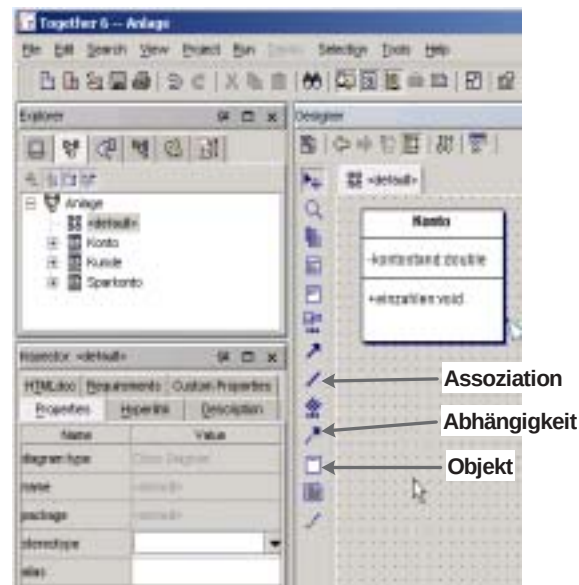


Abbildung 11.15: Zeichenwerkzeuge im Objektdiagramm

Das Objekt-Werkzeug dient zur Positionierung eines Objektes im Klassendiagramm. Mit dem Assoziationswerkzeug stellen Sie Beziehungen dar. Die Zuordnung eines Objektes zu einer Klasse, aus der es abgeleitet ist, stellt der Abhängigkeitspfeil dar.

Klicken Sie auf das Objekt-Werkzeug und anschließend in das Zeichenfeld, um ein neues Objekt zu erstellen.

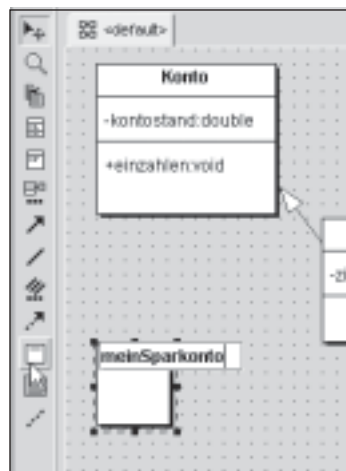


Abbildung 11.16: Objekt im Klassendiagramm

Vereinbaren Sie sofort einen Namen im bereits markierten Namensfeld des Objektes. Im nächsten Schritt vereinbaren Sie eine Verbindung zur Klasse, aus der das Objekt erzeugt wurde, indem Sie einen Abhängigkeitspfeil vom Objekt meinSparkonto zur zugehörigen Klasse Konto zeichnen.

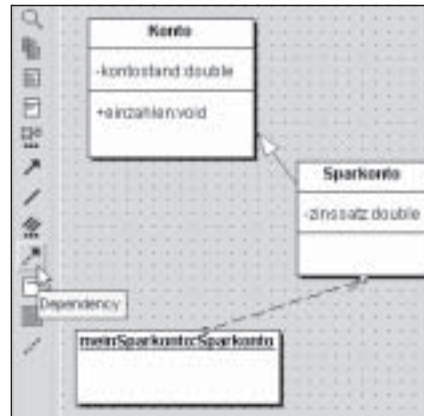


Abbildung 11.17: Klassenzugehörigkeit

Klicken Sie dazu auf das Dependency-Werkzeug, mit dem Abhängigkeiten definiert werden (gestrichelter Pfeil). Anschließend klicken Sie dann auf das Objekt `meinSparkonto`, um den Ursprung des Pfeils zu verankern und abschließend auf die Klasse **Sparkonto**, um die Pfeilspitze festzulegen. Der gestrichelte Pfeil verweist jetzt vom Objekt `meinSparkonto` auf die Klasse **Konto** und legt damit inhaltlich fest, dass das Objekt aus der zugeordneten Klasse abgeleitet ist. Diese Zuordnung zur Klasse **Konto** wird ebenfalls in der Objektbezeichnung `meinSparkonto:Konto` hinter dem Doppelpunkt vermerkt.

Nach dieser Zuordnung besitzt das Objekt `meinSparkonto` alle Attribute und Operationen, die in der Klasse **Sparkonto** und ihrer Oberklasse **Konto** vereinbart sind.

Die Objektattribute können Sie im *Inspector*-Fenster im Bereich *Attributes* mit Attributwerten belegen. Dazu markieren Sie im Klassendiagramm das Objekt `meinSparkonto`. Der *Inspector* zeigt alle Eigenschaften, die für das ausgewählte Objekt festgelegt werden können. Im Bereich *Attributes* werden jedoch Attribute von Oberklassen

nicht berücksichtigt. So bleibt das Attribut `kontostand`, das über die Vererbungsbeziehung von der Klasse `Konto` an die Klasse `Spar-konto` vererbt wird, unberücksichtigt.

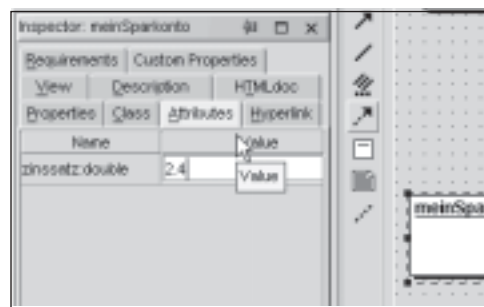


Abbildung 11.18: Attributwerte festlegen

Darüber hinaus existiert keine Möglichkeit, die Attribute mit ihrer Wertbelegung im Diagramm anzuzeigen. Dadurch ist die Aussage der Objektdiagramme eingeschränkt.

Das folgende Diagramm enthält drei zusätzliche Objekte, so dass zwei Sparkonto-Objekte und zwei Kunden-Objekte dargestellt werden. Da zwischen den zugehörigen Klassen eine Inhaber-Assoziation definiert ist, stellen wir diese Verbindung auch zwischen den Objekten her (siehe Abbildung 11.19).

Wählen Sie durch Klicken das Assoziations-Werkzeug aus und klicken Sie anschließend im Diagramm die beteiligten Objekte an. Zwischen diesen Objekten ist damit eine Verbindung definiert. Die Bezeichnung *Inhaber* für die im Diagramm ausgewählte Verbindung vereinbaren Sie im *Inspector* im Bereich *Link* unter der *Label*-Eigenschaft.

Sind in einem Klassendiagramm bereits sehr viele Elemente abgebildet, können Sie das Objektdiagramm auch in einem Kollaborationsdiagramm darstellen (siehe Abbildung 11.20).

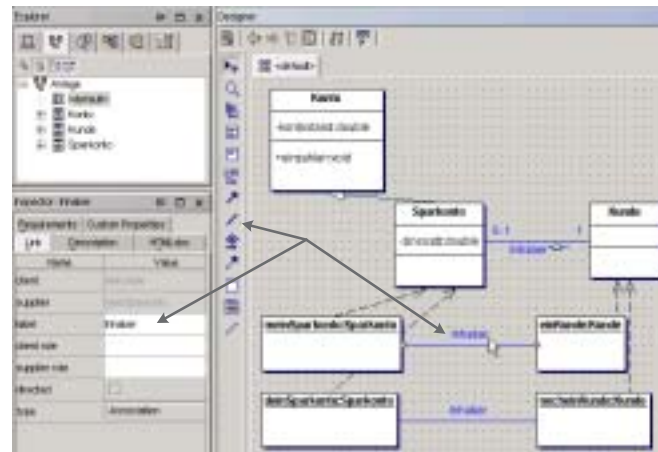


Abbildung 11.19: Objektverbindungen

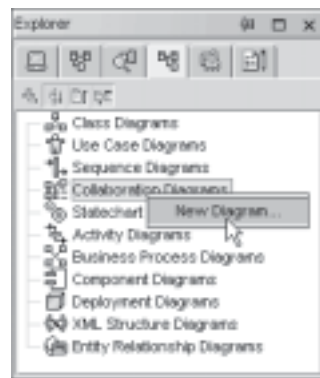


Abbildung 11.20: Kollaborationsdiagramm erstellen I

Ein neues Diagramm erstellen Sie, indem Sie im Explorer die Diagramm-Sicht wählen und auf den gewünschten Typ mit der rechten Maustaste klicken. Anschließend klicken Sie auf *New Diagram*, um den folgenden Auswahl-Dialog zu aktivieren.

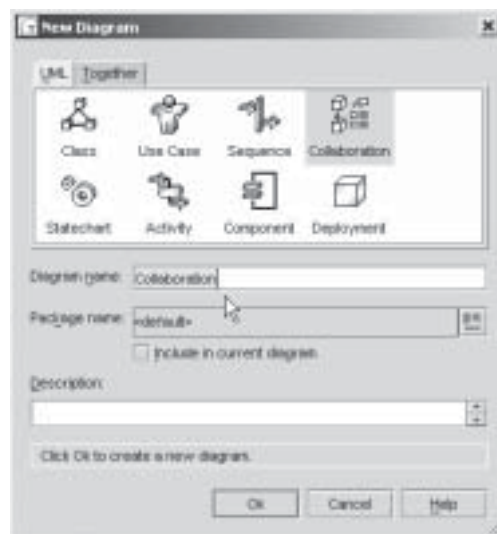


Abbildung 11.21: Kollaborationsdiagramm erstellen II

Geben Sie dem Diagramm einen beliebigen Namen und erstellen Sie es über den OK-Button. Im danach angezeigten Kollaborationsdiagramm können Sie Ihre Objekte genauso erstellen wie im Klassendiagramm. Eine Zuordnung zu einer Klasse können Sie hier im Unterschied zum Klassendiagramm jedoch nur über die direkte Namenseingabe `...:Sparkonto` oder den Auswahl-Dialog *instances* im *Properties*-Bereich des *Inspector* vornehmen.

Vom Klassen- zum Zustandsdiagramm

Um die Veränderungen eines Objektes während seines Lebenszyklus zu zeigen, erstellen Sie ein Zustandsdiagramm. Nachdem Sie bereits die Möglichkeit kennen gelernt haben, ein Diagramm über den Diagramm-Explorer zu erzeugen, wählen wir hier den Weg über das *File*-Menü.

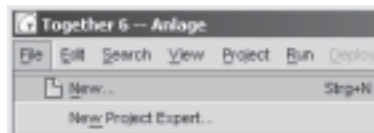


Abbildung 11.22: Zustandsdiagramm erstellen I

Wählen Sie den Menüpunkt *File/New* aus oder drücken Sie die Tastenkombination **Strg** + **N** und Sie gelangen in die *Object Gallery*, in der Sie alle verfügbaren Elemente, also auch Diagramme, erstellen können.

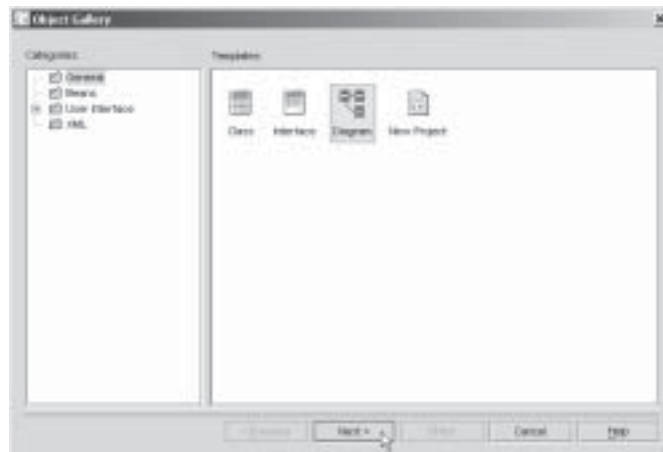


Abbildung 11.23: Zustandsdiagramm erstellen II

Markieren Sie im *Templates*-Fenster das *Diagram*-Symbol und klicken Sie danach auf die Schaltfläche *Next >*. Nun gelangen Sie wiederum in den *New-Diagram-Dialog*. Wählen Sie das *Statechart*-Symbol aus, vereinbaren Sie einen Namen und erstellen Sie das Zustandsdiagramm über den *Finish*-Knopf.

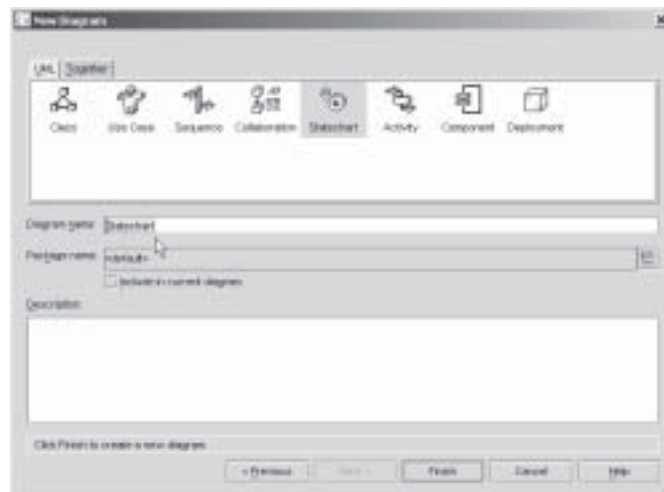


Abbildung 11.24: Zustandsdiagramm erstellen III

Das erstellte Zustandsdiagramm enthält die folgenden elementaren Zeichenwerkzeuge, die Sie benötigen, um Veränderungen von Objekten und anderen Systemelementen zu beschreiben (siehe Abbildung 11.25).

Erstellen Sie zunächst einen Start- und einen Endpunkt, indem Sie die Symbole anwählen und wie gewohnt auf der Zeichenfläche positionieren (siehe Abbildung 11.26).

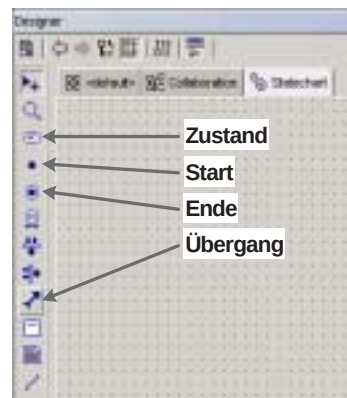


Abbildung 11.25: Werkzeuge im Zustandsdiagramm

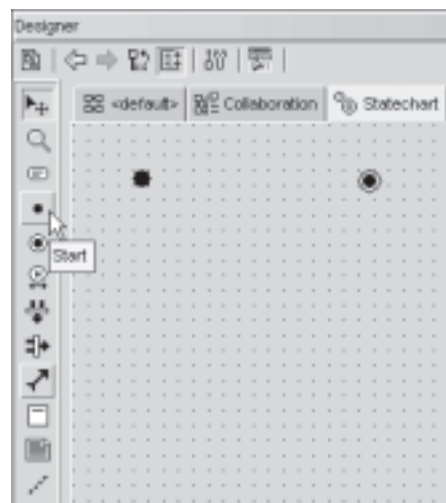


Abbildung 11.26: Zustandsdiagramm Start und Ende

Für ein Konto definieren wir zwei Zustände `KontostandPositiv` und `KontostandNegativ`. Dazu klicken Sie auf das *State*-Symbol in der Werkzeugleiste und anschließend auf die Position der Zeichenfläche, an der Sie den Zustand platzieren möchten. Geben Sie anschließend die Zustandsbezeichnung `KontostandPositiv` in das bereits markierte Bezeichnungsfeld ein.

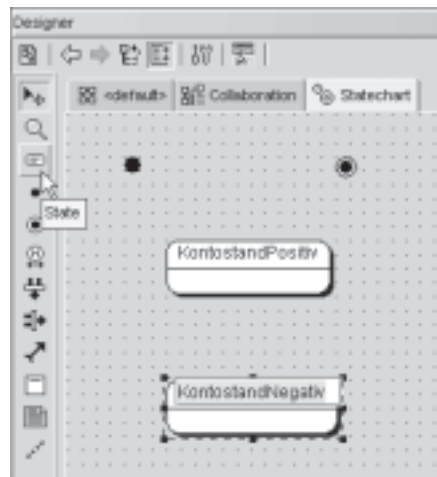


Abbildung 11.27: Zustandsdiagramm Zustand

Wiederholen Sie den Vorgang für den Zustand `KontostandNegativ`. Nach der Vereinbarung der Zustände müssen noch die Zustandsübergänge festgelegt werden, damit Sie sehen können, wie und unter welchen Bedingungen Zustände eintreten und verlassen werden (siehe Abbildung 11.28).

Wählen Sie das *Transition*-Werkzeug und klicken Sie nacheinander auf den Ausgangspunkt (Start-Punkt) und dann auf das Ziel (`KontostandPositiv`) des zu definierenden Zustandsübergangs. Wiederholen Sie das Verfahren zur Definition eines Übergangs vom Zustand `KontostandPositiv` zum End-Punkt.

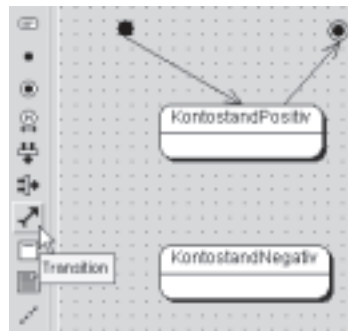


Abbildung 11.28: Zustandsdiagramm Übergang I

Weitere Zustandsübergängen können Sie zwischen den Zuständen KontostandPositiv und KontostandNegativ definieren.

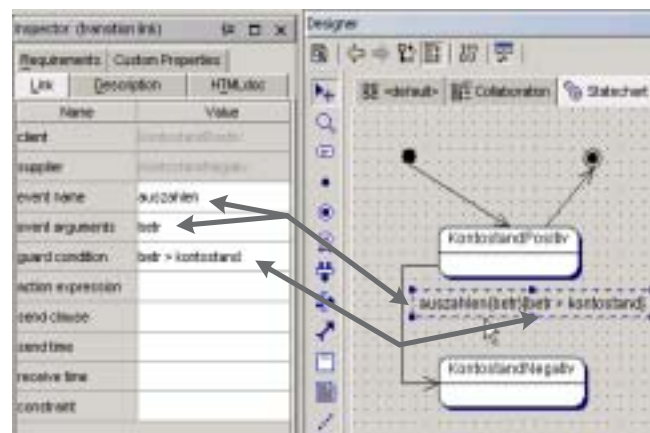


Abbildung 11.29: Zustandsdiagramm Übergang II

Nachdem ein Übergang zwischen den Zuständen `KontostandPositiv` und `KontostandNegativ` vereinbart ist, markieren Sie den Übergang im Diagramm und vereinbaren im Inspektor einen Ereignisnamen (*event name*, `auszahlen`), Argumente (*event arguments*, `betr`) und Bedingungen (*guard condition*, `betr > kontostand`), die an den Übergang geknüpft sind.

Abschließend sei kurz angemerkt, dass auch Übergänge definiert werden können, die Ursprung und Ziel im selben Zustand besitzen. Klicken Sie dazu nach ausgewähltem *Transition*-Werkzeug zweimal auf den betreffenden Zustand.

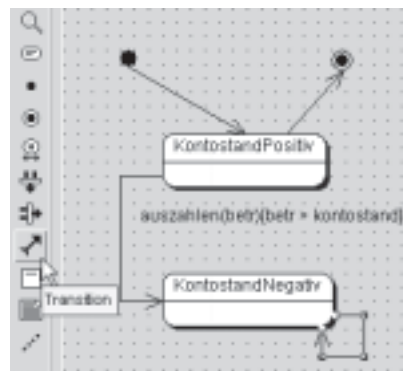


Abbildung 11.30: Zustandsdiagramm Selbstreferenz

Zustände und Zustandsübergänge sind in Together nicht an Klassendefinitionen gebunden. So lassen sich Zustände und Übergänge unabhängig formulieren. Eine syntaktische Prüfung kann demgemäß auch nicht erfolgen.

Vom Klassen- zum Sequenzdiagramm

Im Gegensatz zum Zustandsdiagramm ist das Sequenzdiagramm eng mit dem Klassendiagramm verzahnt. Erstellen Sie ein neues

Sequenzdiagramm, indem Sie im Explorer mit der rechten Maustaste das Kontext-Menü für die *Sequence Diagrams* aktivieren.

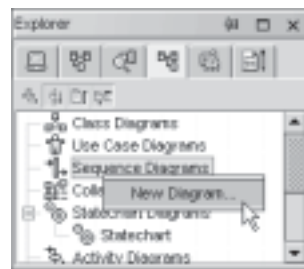


Abbildung 11.31: Sequenzdiagramm erstellen I

Klicken Sie anschließend auf *New Diagram*, um den *New Diagram*-Dialog aufzurufen. Geben Sie einen Namen für das Sequenzdiagramm ein und bestätigen Sie den Dialog über den *OK*-Button (siehe Abbildung 11.32).

Ein neues Sequenzdiagramm erscheint im Zeichenbereich und stellt Ihnen die folgenden elementaren Zeichenwerkzeuge zur Verfügung (siehe Abbildung 11.33).

Im Folgenden demonstrieren wir kurz den Einsatz der Werkzeuge, um sehr schnell formal korrekte Diagramme zu erstellen. Wählen Sie das Akteur-Werkzeug aus und klicken Sie auf die Zeichenfläche. Geben Sie dem erstellten Akteur die Bezeichnung Bankberater (siehe Abbildung 11.34).

Das Diagramm zeigt nun den externen Akteur Bankberater mit Lebenslinie und Aktivitätsbalken. Erstellen Sie im Folgenden ein Konto mit dem Objekt-Werkzeug und geben Sie ihm den Namen meinKonto.

Das Diagramm zeigt das erstellte Objekt mit seiner Lebenslinie (siehe Abbildung 11.35).

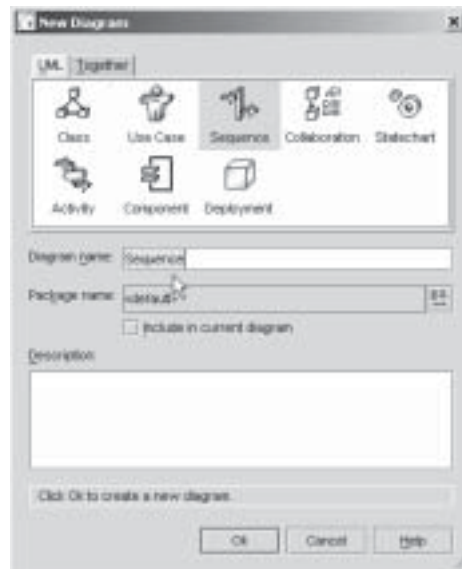


Abbildung 11.32: Sequenzdiagramm erstellen II

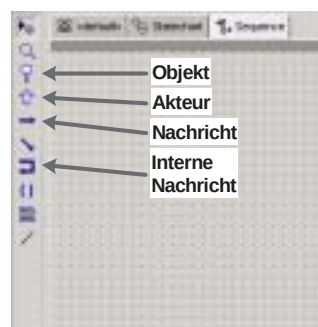


Abbildung 11.33: Werkzeuge im Sequenzdiagramm

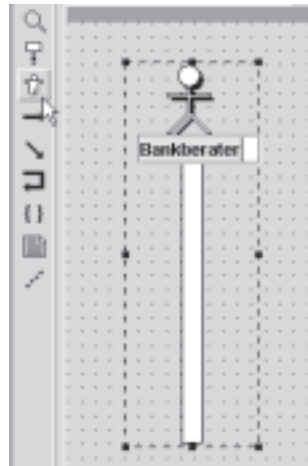


Abbildung 11.34: Sequenzdiagramm Akteur

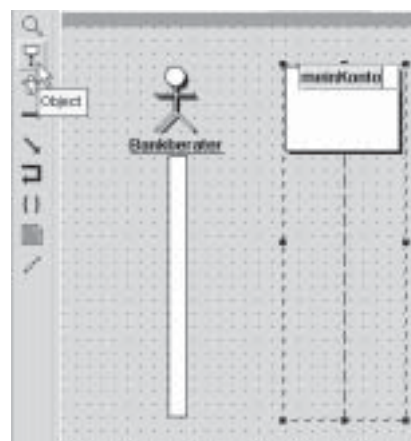


Abbildung 11.35: Sequenzdiagramm Objekt einfügen

Im nächsten Schritt legen Sie fest, aus welcher Klasse das Objekt `meinKonto` abgeleitet ist. Dazu klicken Sie mit der rechten Maustaste auf das Objekt `meinKonto` und wählen im Menü *Choose Class* die Klasse `Sparkonto` aus.

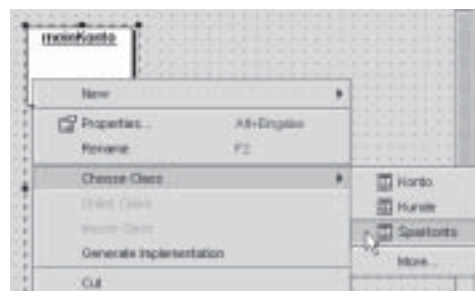


Abbildung 11.36: Sequenzdiagramm Klasse zuordnen

Unterhalb des Namens `meinKonto` wird die ausgewählte Klasse `Sparkonto` nun angezeigt. Im nächsten Schritt vereinbaren Sie mit Hilfe des Nachrichten-Werkzeugs eine Nachricht, die der Bankberater an das `Sparkonto`-Objekt `meinKonto` übermittelt.

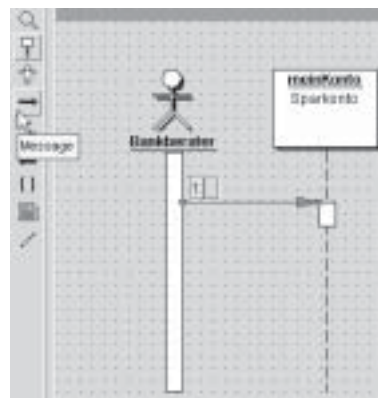


Abbildung 11.37: Sequenzdiagramm Nachricht einfügen

Klicken Sie dazu nach der Auswahl des *Message*-Werkzeugs zunächst auf den Aktivitätsbalken des Bankberater-Akteurs und anschließend auf die Lebenslinie des Objektes *meinKonto*. Im Ergebnis sehen Sie einen Nachrichtenpfeil, der vom Bankberater auf das Sparkonto verweist. Auf der Lebenslinie des Sparkonto-Objekts befindet sich jetzt ebenfalls ein Aktivitätsbalken.

Mit dieser Nachricht aktiviert der Bankberater somit eine Operation des Objektes *meinKonto*, das ja aus der Klasse *Sparkonto* abgeleitet ist.

Klicken Sie mit der rechten Maustaste auf den Nachrichtenpfeil, um den Operationsnamen festzulegen.

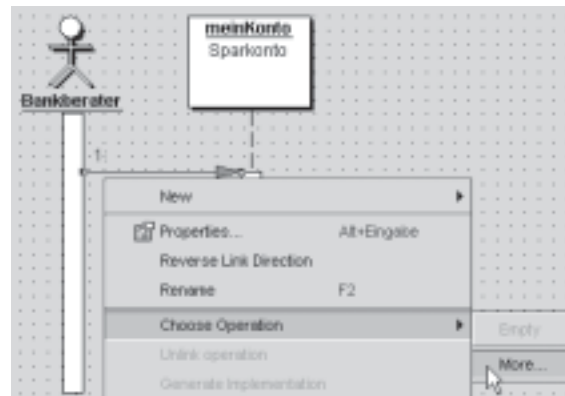


Abbildung 11.38: Sequenzdiagramm Operation einfügen I

Wählen Sie aus dem Kontextmenü *Choose Operation* die Option *More*, um alle Operationen der Vererbungslinie der Klasse *Sparkonto* anzuzeigen.

Wählen Sie im erscheinenden Dialogfenster *Choose Operation Name* die Operation *einzahlen(...)* aus, die durch die Vererbung von der Klasse *Konto* an die Klasse *Sparkonto* weitergegeben wird.



Abbildung 11.39: Sequenzdiagramm Operation einfügen II

Beenden Sie den Dialog, indem Sie Ihre Auswahl mit dem OK-Knopf bestätigen.

Im Diagramm sehen Sie nun über dem Nachrichtenpfeil den Namen der ausgewählten Operation.

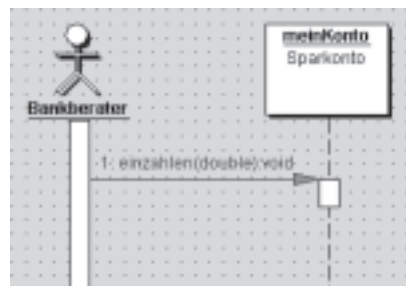


Abbildung 11.40: Sequenzdiagramm Operation

Eine weitere Option dient dazu, interne Nachrichten, die ein Objekt an sich selbst schickt, zu erzeugen. Darüber hinaus bietet Together wohl die ausgereifteste Verbindung aller Tools zwischen Sequenzdiagramm und Java-Quellcode. Sie können visuell Code-Sequenzen für einzelne Aktivitätsbalken definieren, die zwar nicht simultan, aber doch auf Anforderung in den Java-Quellcode der im Aktivitätsbalken aktiven Operation übernommen werden.

Damit haben wir nur einen Bruchteil der Optionen von Together angerissen. Das Tool besitzt eine Fülle weiterer Optionen, um die Effektivität und die Effizienz der Software-Entwicklung zu steigern, die wir hier leider nicht besprechen können. Together ist sicherlich die innovativste Entwicklungsumgebung, die in den letzten Jahren im Bereich der objektorientierten Systementwicklung angeboten wird.

Im Menü *File* können Sie mit der Anweisung *Exit* Together wieder verlassen.

Rational Rose

Rational Rose ist ein grafisches Modellierungswerkzeug der Firma Rational Software. Es unterstützt, außer den Objektdiagrammen, sämtliche Diagrammart der UML. Neben der Entwicklung von Software können Sie mit Rational Rose auch Geschäftsprozesse, Daten und Web-Applikationen analysieren und modellieren sowie Middleware-Komponenten testen. Außerdem wird XML von Rational Rose unterstützt.

Überblick

Die Software bietet umfangreiche Hilfsmittel, die Ihnen bei der Analyse, dem Design und der iterativen Konstruktion Ihrer Anwendungen zur Verfügung stehen. Dazu zählen u.a.:

- ✓ Use-Case-Analysen
- ✓ Objektorientierte Modellierung
- ✓ Konfigurierbare Unterstützung verschiedener Modellierungssprachen (u.a. UML und OMT)
- ✓ Roundtrip Engineering
- ✓ Semantische Überprüfung

- ✓ Multi-User-Unterstützung
- ✓ Integration von Tools zur Datenmodellierung
- ✓ Dokumentationsgenerierung

Wie Sie Rational Rose auf Ihrem Rechner installieren können, wird in Anhang B beschrieben. Um das Programm nach erfolgreicher Installation zu starten, wählen Sie im Startmenü aus dem Programmordner zu Rational Rose den Eintrag *Rational Rose Enterprise Edition* aus.

Daraufhin erhalten Sie zunächst die folgende Meldung, die Ihnen anzeigt, wie lange Ihre Lizenz noch gültig ist.



Diese Meldung erscheint nur, wenn Ihre Lizenz in weniger als zehn Tagen abläuft. Sie ist zudem im Rational License Manager deaktivierbar.

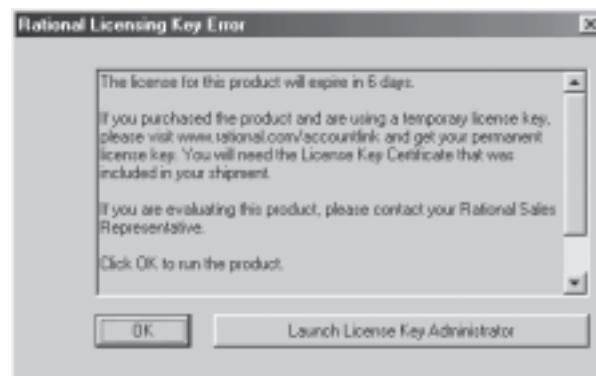


Abbildung 11.41: Meldung Lizenzablauf

Bestätigen Sie diese Meldung mit einem Mausklick auf die Schaltfläche **OK**.

Anschließend erscheint auf Ihrem Bildschirm die Benutzungsoberfläche von Rational Rose. Sie ist aufgeteilt in verschiedene Fenster, über die Sie die einzelnen Funktionen aufrufen und Ihre Anwendung bearbeiten können. Innerhalb des Hauptfensters stehen folgende Fenster für die verschiedenen Funktionen zur Verfügung:

- ✓ Browser Window
- ✓ Documentation Window
- ✓ Diagram Window
- ✓ Overview Window
- ✓ Specification Window
- ✓ Log Window

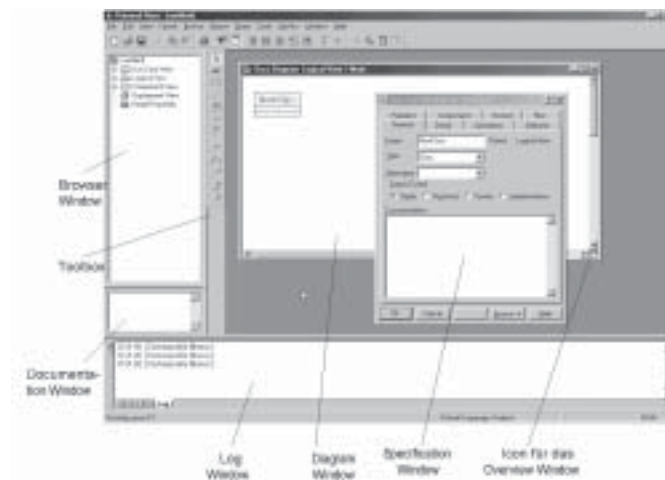


Abbildung 11.42: Benutzungsoberfläche von Rational Rose

Das *Browser Window* ist ein hierarchisch aufgebautes Navigationswerkzeug, mit dem Sie die Anzeige der verschiedenen Modellele-

mente (z.B. Use Cases, Klassen, Interaktionen oder Aktivitäten) aufrufen können.

Das *Documentation Window* dient der näheren Beschreibung von Modellelementen oder Beziehungen. Zu der Beschreibung gehören beispielsweise Informationen über Rollen, Schlüssel, Zusicherungen oder das Verhalten von Elementen. Sie können diese Beschreibungen selbst erstellen, um Ihre Modellelemente mit weiteren Informationen zu versehen und ihre Eigenschaften auch in einem umfangreicheren Modell noch überblicken zu können. Die Position des Fenster innerhalb der Benutzungsoberfläche lässt sich durch einfaches Klicken und Ziehen mit der Maus anpassen.

Die Ergebnisse eines Befehls oder einer Aktion in Ihrem Modell sowie aufgetretene Fehler werden im *Log Window* angezeigt. Diese Meldungen werden mit einem Zeitstempel versehen, so dass jederzeit nachvollziehbar bleibt, wann und in welcher Reihenfolge die gemeldeten Ereignisse aufgetreten sind. Die Inhalte des Fensters können für eine spätere Bearbeitung gespeichert werden. Auch das Log Window ist auf der Oberfläche frei positionierbar.

Die grafische Entwicklung Ihres Objektmodells nehmen Sie im *Diagram Window* vor. Jedes Symbol des Diagramms repräsentiert ein Element des Modells. Die Symbole entsprechen der UML-Spezifikation, die auch in diesem Buch verwendet wird. Das Tool bietet allerdings auch die Möglichkeit, andere Notationen (z.B. OMT) zu benutzen.

Das *Overview Window* dient zur Navigation in umfangreichen Modellen. Falls Ihr Diagramm – und das ist eher die Regel – größer ist als der sichtbare Bereich des Diagram Window, erhalten Sie über das Overview Window eine verkleinerte Ansicht des gesamten Diagramms. So gelangen Sie an jede beliebige Stelle innerhalb des Modells. Das Overview Window aktivieren Sie durch einen Mausklick auf das Icon (stilisierte Hand) unten rechts im Diagram Window.



Das Overview Window bleibt aktiv, solange Sie die Maustaste gedrückt halten.

Im *Specification Window* werden die Eigenschaften und Beziehungen der Modellelemente angezeigt und können dort auch vom Benutzer manipuliert werden. Dieses Fenster ist zunächst noch nicht geöffnet. Sie müssen zuerst ein Objektmodell laden oder zumindest ein Element neu erstellen. Wenn Sie mit der rechten Maustaste auf ein Symbol im *Diagram Window* klicken, öffnet sich ein Kontextmenü. Wählen Sie dort die Option *Open Specification* und das *Specification Window* öffnet sich mit den Eigenschaften des ausgewählten Elements.

Zwischen dem *Browser Window* und dem *Diagram Window* befindet sich die *Toolbox*. Mit den dort angeordneten Icons können Sie die verschiedenen Symbole auswählen, um Ihr Diagramm zu zeichnen.

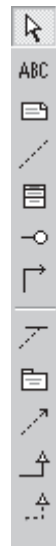


Abbildung 11.43: Toolbox

Die Auswahl der angezeigten Icons bzw. Symbole ist individuell konfigurierbar. In der Voreinstellung stehen die in der obigen Abbildung dargestellten Icons zur Verfügung.

Das erste Icon (Pfeil) dient zur Auswahl von Elementen des Objektmodells. Per Mausklick können die Symbole des Objektdiagramms ausgewählt und bearbeitet werden.

Das Icon mit der Aufschrift *ABC* benötigen Sie, um Text in Ihr Diagramm einzugeben.

Eine neue Klasse fügen Sie in Ihr Diagramm ein, indem Sie auf das Klassensymbol der Toolbox klicken (fünftes Icon von oben) und anschließend den Mauszeiger in das Diagram Window bewegen. Dort klicken Sie erneut mit der linken Maustaste. Es erscheint ein neues Klassensymbol im Diagramm.

Probieren Sie die Funktionen der übrigen Icons einfach aus. Sobald Sie den Mauszeiger über eines der Icons bewegen, erscheint eine entsprechende Beschriftung, die über die Funktion des jeweiligen Icons informiert.



Abbildung 11.44: Menüleiste

Über die *Menüleiste* von Rational Rose werden weitere Programmfunktionen gesteuert. Die Icons für das Öffnen und Speichern von Dokumenten sowie die Steuerung der Zwischenablage kennen Sie sicher bereits von vielen anderen Anwendungen.

Für die ersten Schritte mit der Entwicklungsumgebung ist die kontextsensitive Hilfe eine nützliche Funktion.



Abbildung 11.45: Kontextsensitive Hilfe

Klicken Sie zunächst auf dieses Icon in der Menüleiste und dann auf den Bereich der Benutzungsoberfläche, zu dem Sie eine Erläuterung wünschen (z.B. ein Fenster oder ein anderes Icon). Daraufhin öffnet sich ein Fenster mit einem erklärenden Text, der nähere Informationen zu dem ausgewählten Element enthält.

Mit den Icons weiter rechts in der Menüleiste werden die verschiedenen Diagramme Ihres Objektmodells angezeigt.



Abbildung 11.46: Auswahl von Diagrammen

Hier können Sie auswählen, ob beispielsweise ein Klassendiagramm, ein Zustandsdiagramm oder ein Interaktionsdiagramm im Diagram Window erscheinen soll. Sobald Sie das entsprechende Icon anklicken, öffnet sich ein Fenster, in dem Sie Ihre Auswahl vornehmen können.

Rational Rose enthält über die Grundfunktionen des Programms hinaus einige nützliche Assistenten (Wizards), die Ihnen die Benutzung der Software erleichtern.

Auch für das Erstellen eines Objektmodells gibt es einen solchen Assistenten. Gerade beim Einstieg in die Anwendung der Entwicklungsumgebung ist der *Framework Wizard* eine große Hilfe. Sie rufen den Framework Wizard auf, indem Sie im Menü *File* auf die Option *New* klicken. Daraufhin öffnet sich das in Abbildung 11.47 dargestellte Fenster.



Dieser Wizard ist ein Add-In, welches Sie im Add-In-Manager auch deaktivieren können.

Wählen Sie hier beispielsweise das Icon mit der Beschriftung *jdk-12* aus und bestätigen Sie Ihre Auswahl mit einem Mausklick auf die Schaltfläche *OK*.



Abbildung 11.47: Neues Objektmodell erstellen

Der *Framework Wizard* lädt nun einige Pakete, die zu dem Java Development Kit in der Version 1.2 gehören. Welche Dateien genau geladen werden, wird Ihnen im Log Window angezeigt. Das Diagram Window sieht nach Beendigung des Ladevorgangs wie in Abbildung 11.48 dargestellt aus.

Die Abbildung zeigt die vier Pakete, die zu diesem Framework gehören. Jedes dieser Pakete enthält weitere Unterpakete und schließlich auf der untersten Ebene die entsprechenden Klassendiagramme. Nach einem Doppelklick auf ein Paket öffnet sich ein neues Fenster mit dessen Inhalt.

Über die Pakete *javax – swing – event* erhalten Sie beispielsweise eine grafische Darstellung der Event-Klassen des JDK 1.2. Das Klassendiagramm veranschaulicht die Beziehungen der Klassen in diesem Paket (siehe Abbildung 11.49).

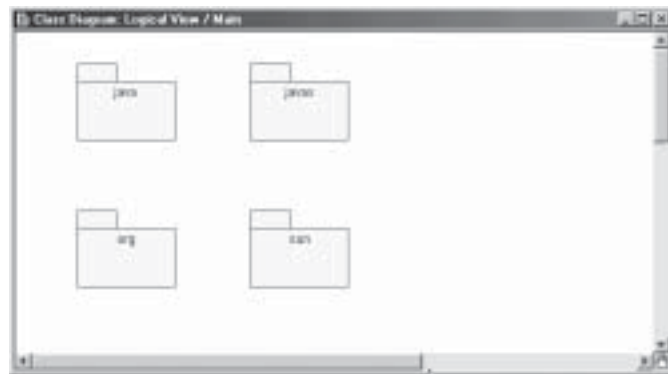


Abbildung 11.48: Framework JDK 1.2

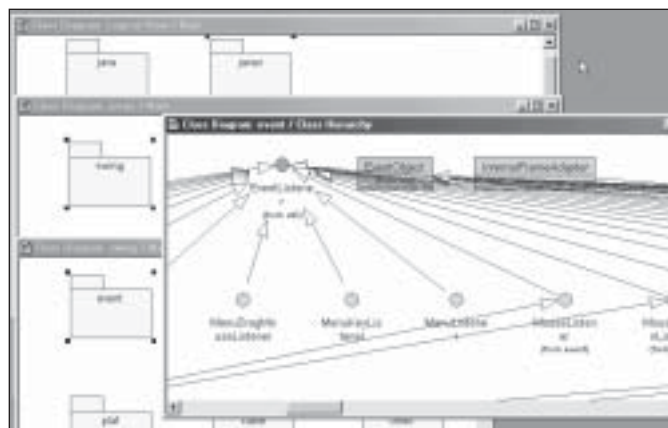


Abbildung 11.49: Java-Events

Klassendiagramme erstellen

Anhand der Klassendiagramme demonstriert der folgende Abschnitt das Arbeiten mit Rational Rose exemplarisch etwas näher.

Klassendiagramme enthalten Symbole, die Klassen, Schnittstellen und ihre Beziehungen untereinander grafisch darstellen.

Wenn Sie unter dem Menüpunkt *View* die Option *As Unified* ausgewählt haben und alle Elemente der Toolbox für Klassendiagramme aktivieren, stehen Ihnen die in der folgenden Abbildung dargestellten Optionen in der Toolbox zur Verfügung.

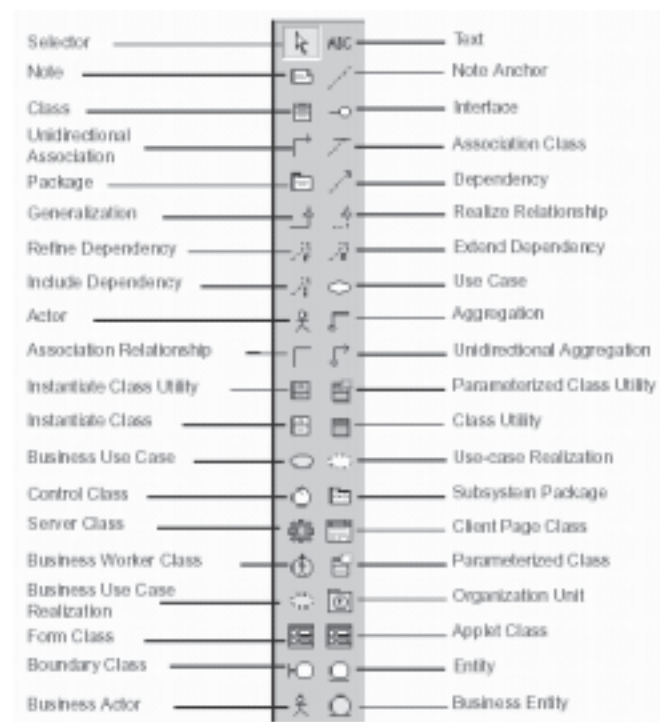


Abbildung 11.50: Toolbox zum Klassendiagramm

Um ein Klassendiagramm zu erstellen, stehen Ihnen drei Möglichkeiten zur Verfügung:

- ✓ Klicken Sie mit der Maus auf *Browse* und anschließend auf *Class Diagram*.
- ✓ Klicken Sie in der Toolbox auf das Symbol für Klassendiagramme.
- ✓ Führen Sie im Browser einen Doppelklick auf das Symbol für Klassendiagramme aus.

Jede Klasse gehört zu einem logischen Paket innerhalb Ihres Modells. Um die Zuordnung einer oder mehrerer Klassen zu Paketen zu ändern, wählen Sie die gewünschten Klassensymbole aus und klicken Sie im Menü *Edit* auf *Relocate*. Daraufhin werden alle Klassendiagramme aktualisiert und die neue Zuordnung dargestellt.

Mit Hilfe der *Class Specification* können Sie die Eigenschaften einer Klasse bestimmen bzw. verändern. Um das zugehörige Dialogfenster zu öffnen, führen Sie einen Doppelklick auf ein Klassensymbol aus.

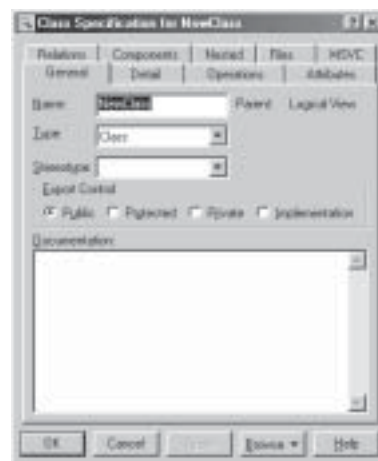


Abbildung 11.51: Dialogfenster *Class Specification*

In diesem Dialogfenster stehen Ihnen nun verschiedene Register zur Verfügung, über die Sie die Eigenschaften der aktuellen Klasse bearbeiten können. Dies sind im Einzelnen die Register *General*, *Detail*, *Operations*, *Attributes*, *Relations*, *Component*, *Nested* und *Files*.

Im Register *General* legen Sie neben dem Namen auch den Typ einer Klasse fest. Zur Auswahl stehen hier u.a. die Typen *einfache*, *parametrisierte* und *Metaklasse*.

Eine weitere Spezifikation der Klasse lässt sich darunter mit Hilfe der Stereotypen vornehmen.

Neben dem Eingabefeld für den Klassennamen wird Ihnen die Paketzugehörigkeit (Parent) angezeigt.

Die Radio Buttons unter der Bezeichnung *Export Control* dienen zur Bestimmung der Sichtbarkeit der Klasse mit Hilfe der Modifizierer *Public*, *Protected*, *Private* und *Implementation*.

Das Register *Detail* ermöglicht eine weiter gehende Spezifikation Ihrer Klassen.



Abbildung 11.52: Register *Detail*

Multiplicity gibt an, wie viele Elemente der jeweiligen Klasse an einer Beziehung beteiligt sind. In dem Feld *Space* kann angegeben werden, wie viel Speicherplatz die Instanzen dieser Klasse bei der Ausführung benötigen.

Die Persistenz bestimmt den Lebenszyklus von Objekten dieser Klasse.

- ✓ Ein persistentes Objekt kann über die Laufzeit des Programms hinaus weiter existieren.
- ✓ Der Lebenszyklus eines transienten Objekts endet (spätestens) mit der Laufzeit des Programms.

Durch die Aktivierung der Checkbox *Abstract* definieren Sie die aktuelle Klasse als abstrakt. Damit können keine Objekte dieser Klasse erzeugt werden. Sie stellt lediglich ein Muster für ihre Unterklassen dar.

Das Register *Operations* definiert die Dienste der Klasse. Dies sind die Methoden für den Zugriff auf Attribute der Klasse und deren Manipulation. Um neben den in der Klasse selbst definierten Methoden auch die aus Oberklassen geerbten Dienste anzuzeigen, aktivieren Sie die Checkbox *Show inherited*.

Einen ähnlichen Aufbau besitzt das Register *Attributes*, in dem die Attribute der Klasse spezifiziert werden. Das Register *Relations* listet die Beziehungen der aktuellen Klasse zu anderen Klassen auf.

Um die Attribute einer Klasse näher zu beschreiben, öffnen Sie im Register *Attributes* durch einen Klick mit der rechten Maustaste das Pop-up-Menü. Dort wählen Sie die Option *Insert* aus (siehe Abbildung 11.53).

Innerhalb des Registers wird nun ein neues Attribut angezeigt, wie in der folgenden Abbildung dargestellt (siehe Abbildung 11.54).

Nach einem Doppelklick auf den Namen des Attributs öffnet sich das Dialogfenster für die Spezifikation dieses Attributs (siehe Abbildung 11.55).

**Abbildung 11.53:** Einfügen von Attributen**Abbildung 11.54:** Ein neues Attribut der aktuellen Klasse

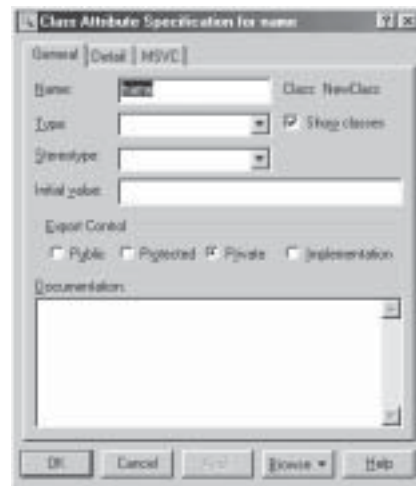


Abbildung 11.55: Dialog für die Spezifikation von Attributen

Auch hier können Sie, wie schon für die Klasse, wiederum den Namen, den Typ, Stereotypen und die Sichtbarkeit festlegen.

Die Klassenzugehörigkeit des gerade bearbeiteten Attributs wird rechts neben dem Textfeld für den Attributnamen angezeigt.

Abbildung 11.56 zeigt die Vorgehensweise am Beispiel einer Klasse Konto (siehe Abbildung 11.56).

Zunächst wird die Klasse Konto wie oben beschrieben angelegt. Danach können die Attribute dieser Klasse mit Hilfe des Eingabedialogs näher bestimmt werden. Die obige Abbildung zeigt dies am Beispiel des Attributs kontoStand.

Auf die gleiche Art gehen Sie vor, um Operationen der Klasse zu bestimmen: Drücken Sie im Register *Operations* die rechte Maustaste, wählen Sie die Option *Insert* und definieren Sie die Operation in dem sich öffnenden Dialogfenster (siehe Abbildung 11.57).

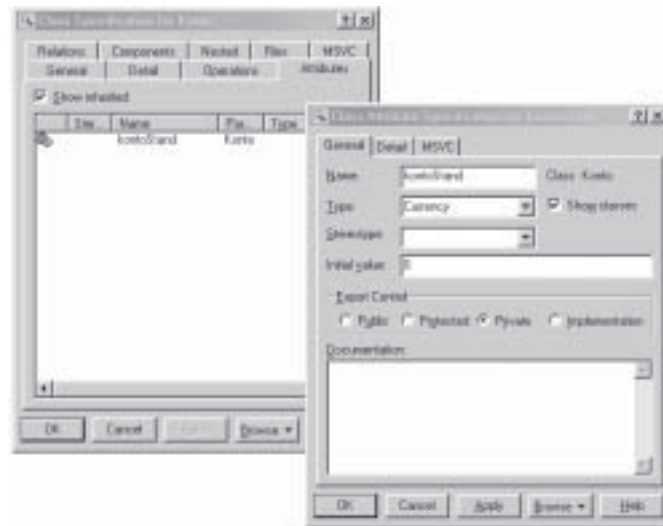


Abbildung 11.56: Attribut-Spezifikation am Beispiel der Klasse Konto

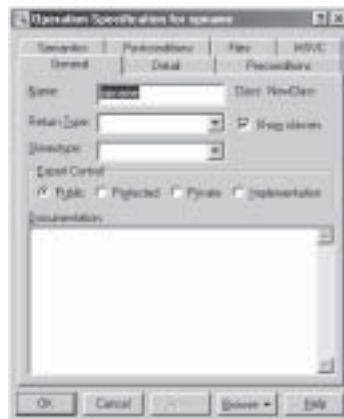


Abbildung 11.57: Dialog für die Spezifikation von Operationen

Neben den bereits bekannten Eigenschaften können Sie hier auch den Rückgabewert der Operation bestimmen.

Das Register *Detail* bietet Ihnen u.a. die Möglichkeit, die Argumente festzulegen, die der Operation beim Aufruf übergeben werden, und die Ausnahmen (Exceptions) zu definieren, die während der Ausführung der Operation auftreten können.

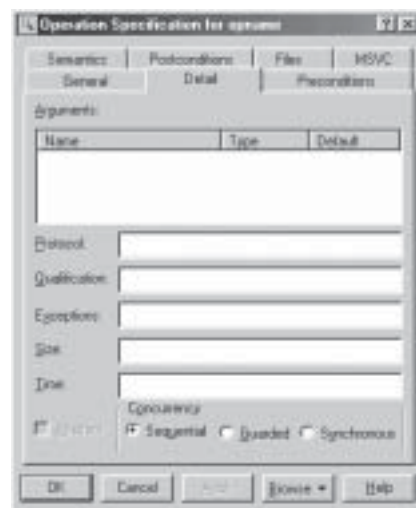


Abbildung 11.58: Register *Detail*

Sobald in Ihrem Klassendiagramm mehrere Klassen existieren, können Sie die Beziehungen zwischen diesen Klassen näher beschreiben. Dazu führen Sie einen Doppelklick auf das zugehörige Symbol aus, das die Klassenbeziehung grafisch darstellt, wie in Abbildung 11.59 zu sehen.

Daraufhin öffnet sich das in Abbildung 11.60 dargestellte Dialogfenster.



Abbildung 11.59: Öffnen des Dialogfensters *Association Specification*



Abbildung 11.60: Dialogfenster *Association Specification*

Die Eingaben werden auch hier wieder über die verschiedenen Register vorgenommen. Wie in der obigen Abbildung zu sehen, werden die an der Assoziation beteiligten Elemente mit ihrem Namen angezeigt (hier Element A: Klasse 2 und Element B: Klasse 1). Links daneben können Sie einen Namen für die jeweilige Rolle der Elemente in der Beziehung vergeben.

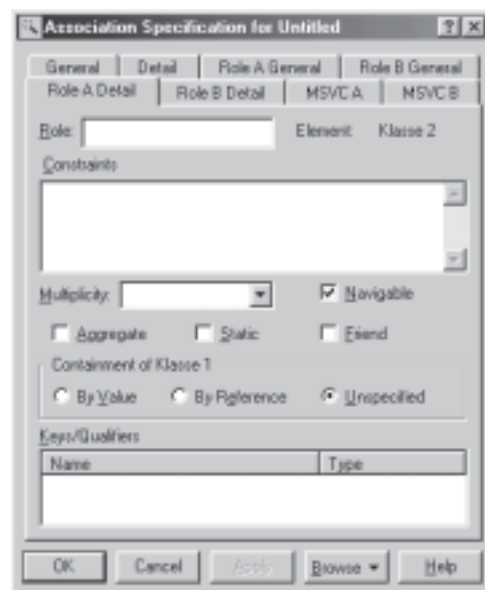


Abbildung 11.61: Register Role Detail

Die beiden Register für die Rollendetails ermöglichen die nähere Bestimmung des Verhältnisses der beiden an der Assoziation beteiligten Elemente.

Je nachdem, um welche Art von Beziehung es sich zwischen den Klassen handelt, öffnet sich nach dem Doppelklick auf das Beziehungssymbol jeweils das passende Dialogfenster. Bei einer Vererbungsbeziehung (Generalisierung) blendet Rational Rose beispielsweise den in Abbildung 11.62 gezeigten Dialog ein.

Rational Rose ermöglicht es Ihnen auch, mehrere Dialogfenster gleichzeitig zu bearbeiten, wie Abbildung 11.63 zeigt.

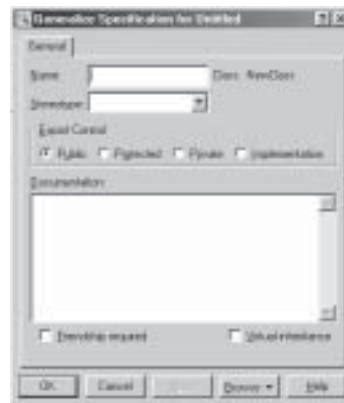


Abbildung 11.62: Dialogfenster *Generalize Specification*

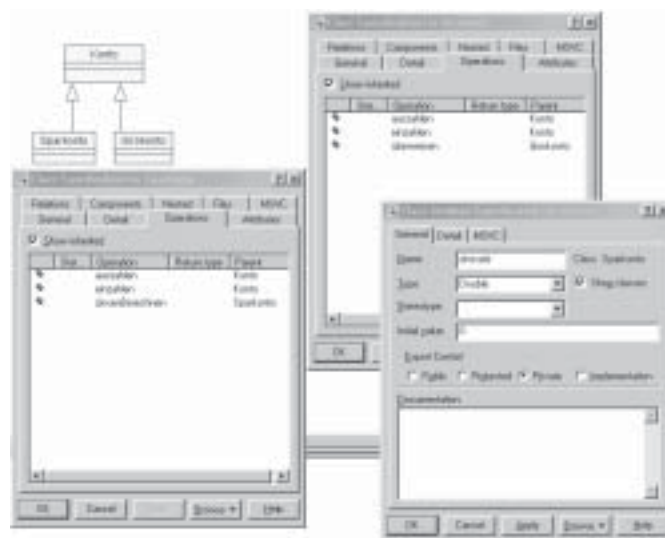


Abbildung 11.63: Gleichzeitiges Bearbeiten mehrerer Dialogfenster

Ausführliche Informationen zum Umgang mit Rational Rose erhalten Sie in der PDF-Datei *Rose_usingrose*, die Sie auf der CD-ROM im Verzeichnis *Tools* und dort im Unterverzeichnis *Rational Rose* finden.

System Architect

Die Firma Popkin versteht Ihre Software System Architect in erster Linie als ein computergestütztes Werkzeug für die Unternehmensmodellierung. Das System umfasst im Wesentlichen die folgenden Funktionen:

- ✓ Geschäftsprozessmodellierung
- ✓ Objektorientierte Modellierung
- ✓ Komponentenbasierte Entwicklung
- ✓ Relationale Datenmodellierung
- ✓ Strukturierte Analyse
- ✓ Strukturiertes Design

Alle diese Elemente werden in einer so genannten *Enzyklopädie* für die Unternehmensmodellierung zusammengefasst.

Die objektorientierte Modellierung, die in unserer Thematik im Vordergrund steht, wird u.a. mit Hilfe der UML umgesetzt. System Architect unterstützt dabei sowohl das Forward Engineering als auch das Backward Engineering zu verschiedenen Sprachen.



Unter dem Begriff *Forward Engineering* ist die Implementierung eines objektorientierten Modells in einer Programmiersprache zu verstehen. *Backward Engineering* bezeichnet den umgekehrten Prozess: Aus einem bestehenden Programmcode wird das entsprechende Objektmodell generiert.

Neben der UML unterstützt System Architect auch andere objektorientierte Notationen, wie beispielsweise OMT, Booch oder Coad/Yourdon. Im Rahmen des Forward Engineering kann aus Klassendiagrammen der UML automatisch Programmcode generiert werden (u.a. in den Sprachen C++, Java, Visual Basic und JavaScript).

Umgekehrt erzeugt die Software aus Programmcode, der in den Sprachen C++ oder Java vorliegt, automatisch die entsprechenden Klassendiagramme in der UML.

Wenn Sie das Programm zum ersten Mal öffnen, öffnet sich ein Fenster, das Sie nach einer *Audit ID* fragt. Anhand der Audit ID unterscheidet System Architect die unterschiedlichen Benutzer, die mit dem System arbeiten. Wenn Sie im Team an einem Projekt arbeiten, lässt sich so jederzeit nachvollziehen, wer wann welche Änderungen oder Ergänzungen vorgenommen hat. Die Software bietet Ihnen die Möglichkeit, diese Informationen in einem Report auszudrucken.

Wie bereits erwähnt, speichert System Architect alle Arbeiten, die zu einem Projekt gehören, in einer Enzyklopädie. Diese besteht aus einer Sammlung von Dateien, die in einem Verzeichnis auf Ihrer Festplatte zusammengefasst werden. Nach dem ersten Programmstart wird automatisch die Enzyklopädie *Samples* geladen. Dieser Name wird in der Kopfzeile des Programmfensters mit dem kompletten Verzeichnispfad angezeigt.

Um ein neues Projekt zu beginnen, erzeugen Sie zunächst eine neue Enzyklopädie.

Rufen Sie dazu im Menü *File* die Option *Open Encyclopedia* auf, wie in der Abbildung 11.64 dargestellt.

In dem anschließenden Dialogfenster klicken Sie auf das Arbeitsblatt mit der Aufschrift *New* und geben einen Namen und ein Zielverzeichnis für das neue Projekt ein (siehe Abbildung 11.65).

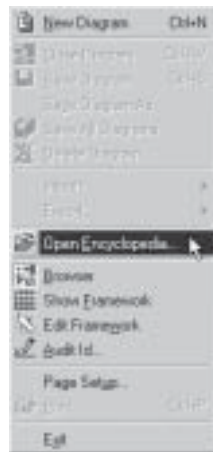


Abbildung 11.64: Enzyklopädie öffnen

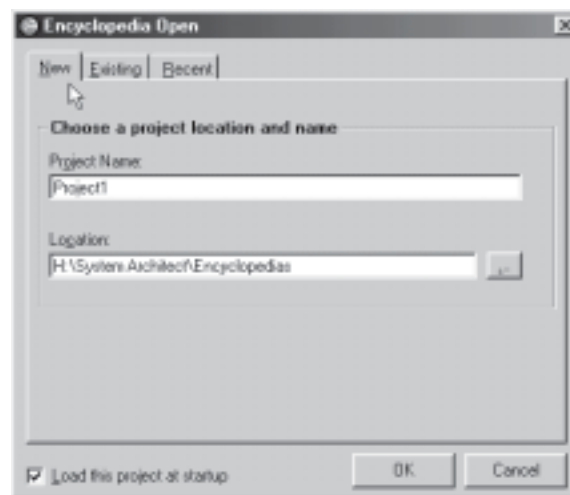


Abbildung 11.65: Neues Projekt anlegen

Klicken Sie dann auf die Schaltfläche **OK**. Nun öffnet sich ein weiteres Fenster, mit dem Sie die Eigenschaften Ihres neuen Projekts konfigurieren können.

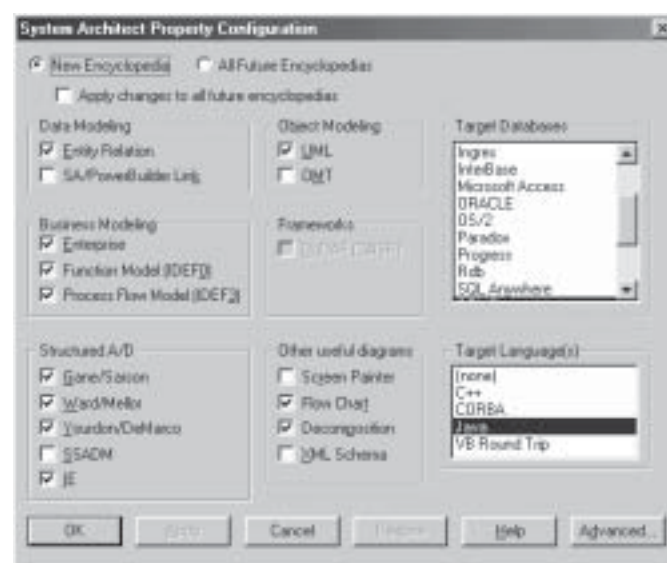


Abbildung 11.66: Konfiguration der Projekteigenschaften

Die Optionen, mit denen Sie Ihr Projekt in diesem Dialog konfigurieren können, sind weitgehend selbsterklärend. Um das Programm erst einmal kennen zu lernen, können Sie die Voreinstellungen einfach übernehmen, indem Sie diese mit einem Klick auf den **OK**-Button bestätigen. Ihr erstes Projekt ist nun mitsamt der erforderlichen Enzyklopädie angelegt.

Die Benutzeroberfläche des System Architect ist in drei Bereiche aufgeteilt, wie Sie der folgenden Abbildung entnehmen können.

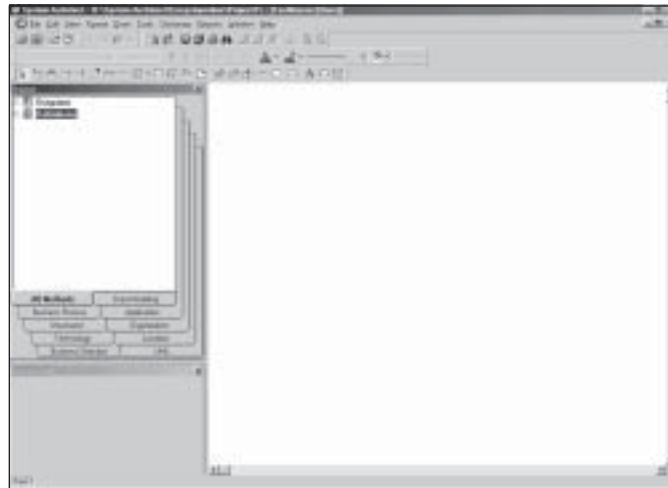


Abbildung 11.67: Benutzungsoberfläche des System Architect

Auf der linken Seite befindet sich in der oberen Hälfte ein Fenster mit der Bezeichnung *Browse*. Dieses Fenster besteht aus verschiedenen Arbeitsblättern, die Sie über stilisierte Karteireiter anwählen können.

In der Voreinstellung ist das Arbeitsblatt *All Methods* aktiv. Es ermöglicht Ihnen den Zugriff auf alle Diagramme und Definitionen, die in der aktuellen Umgebung verfügbar sind. Welche Diagramme und Definitionen das sind, hängt von den Projekteigenschaften ab, die Sie zuvor eingestellt haben.

Die übrigen Arbeitsblätter enthalten jeweils eine Auswahl dieser Elemente, die sich auf den jeweiligen Bereich beziehen. Wenn Sie beispielsweise auf den Karteireiter *UML* klicken, werden im Browser alle Diagramme und Definitionen zur UML angezeigt.

Das untere Fenster auf der linken Seite dient dazu, Details Ihres jeweiligen Modells anzuzeigen. Auf der rechten Seite befindet sich

schließlich das Fenster, in dem Sie Ihre Diagramme erstellen und anzeigen.

Um ein neues UML-Diagramm anzulegen, öffnen Sie das Menü *File* und rufen Sie dort die Funktion *New Diagram* auf. Daraufhin öffnet sich ein weiteres Fenster, in dem Sie die Art des anzulegenden Diagramms bestimmen.

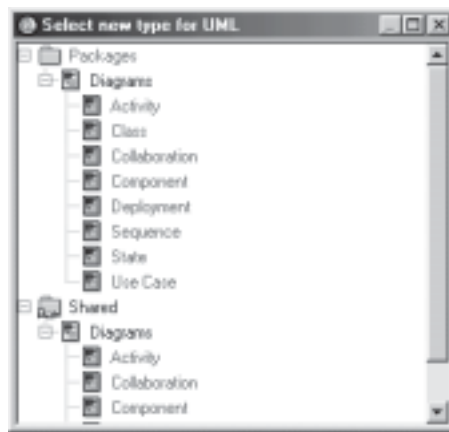


Abbildung 11.68: Auswahl einer neuen Diagrammart

Mit einem Doppelklick auf die Diagrammbezeichnung nehmen Sie Ihre Auswahl vor. Wenn Sie z.B. eine neue Klasse anlegen wollen, führen Sie einen Doppelklick auf den Eintrag *Class* aus.

Es öffnet sich ein anderer Dialog, in dem Sie den Namen der Klasse festlegen (siehe Abbildung 11.69).

Bestätigen Sie die Bezeichnung der Klasse durch einen Mausklick auf den Button *OK*.



Abbildung 11.69: Eingabe des Klassennamens

Der nächste Dialog erwartet eine Zuordnung der neuen Klasse zu einem Paket und die Bestimmung der Programmiersprache, für die diese Klasse angelegt werden soll.

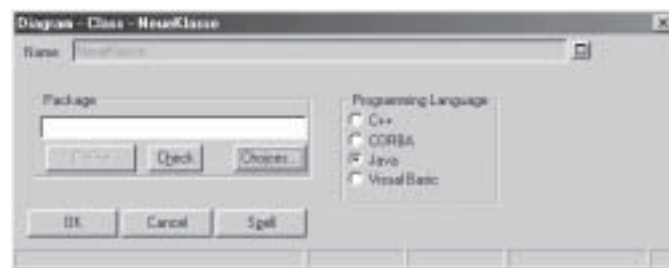


Abbildung 11.70: Auswahl der Paketzugehörigkeit

Sie können die Klasse über die Schaltfläche *Choices* einem bereits existierenden Paket zuordnen oder ein neues Paket erstellen, indem Sie einen noch nicht vergebenen Namen in das Textfeld eintragen und dann auf den Button *Define* klicken. Mit *OK* bestätigen Sie wiederum Ihre Eingaben.

Öffnen Sie nun im Browser das Verzeichnis *Packages*, dann finden Sie dort das zuvor angegebene Paket. Innerhalb dieses Pakets wurde Ihre neue Klasse angelegt.

**Abbildung 11.71:** Die neue Klasse im Browser

Sie zeichnen ein neues Klassensymbol, indem Sie das zugehörige Icon in der Symbolleiste auswählen und anschließend in das große Fenster auf der rechten Seite der Benutzeroberfläche klicken.

**Abbildung 11.72:** Symbolleiste zum Zeichnen von Diagrammen

Das Programm fragt daraufhin nach dem gewünschten Klassennamen. Nachdem Sie diesen eingetragen haben, wird das Klassensymbol im rechten Fenster angezeigt. Mit einem Doppelklick auf das

Diagramm öffnen Sie einen anderen Dialog, der Ihnen die Eingabe sämtlicher Eigenschaften der Klasse ermöglicht.

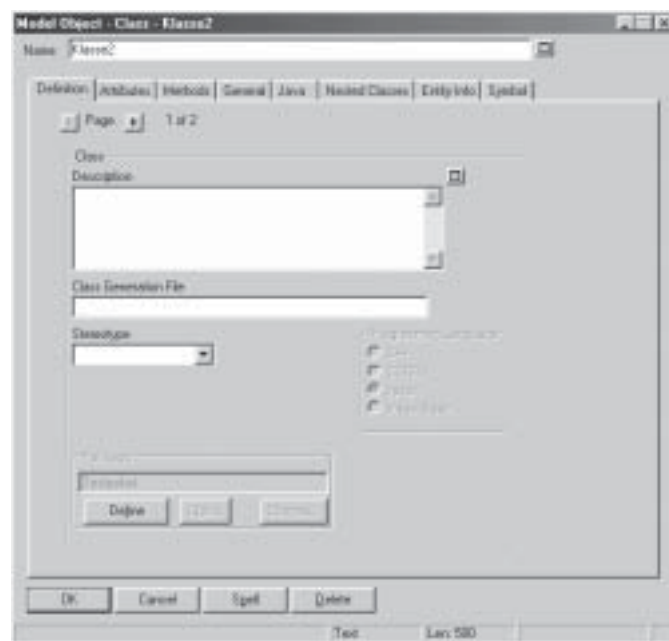


Abbildung 11.73: Eigenschaften einer Klasse festlegen

Klassenbeziehungen lassen sich ebenfalls über die Symbole in der Menüleiste anlegen. Wählen Sie beispielsweise für eine Vererbungsbeziehung das folgende Icon aus.



Abbildung 11.74: Symbol für Vererbungsbeziehungen

Klicken Sie anschließend mit der Maus in Ihrer Grafik zunächst auf die Unterklasse und anschließend auf die zugehörige Oberklasse. Das Programm zeichnet nun den Pfeil ein, der in der UML eine Klassenhierarchie symbolisiert.

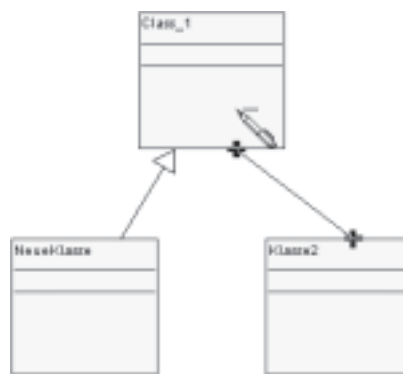


Abbildung 11.75: Zeichnen einer Vererbungsbeziehung

Wie bei den übrigen Tools, die wir Ihnen hier kurz vorstellen, gilt auch für System Architect, dass die Funktionen, die Ihnen das Programm zur Verfügung stellt, weitaus vielfältiger sind, als wir an dieser Stelle demonstrieren können. Mit dieser kurzen Einführung wollen wir Ihnen lediglich die Orientierung auf der Benutzungsoberfläche ein wenig erleichtern.

Verwenden Sie die Tutorials und die Online-Hilfen, um einen tieferen Einblick in die Software mit ihren umfangreichen Funktionen zu gewinnen.

Die PDF-Datei *user_guide* enthält ausführliche Informationen zum Umgang mit System Architect. Sie finden sie auf der CD-ROM im Verzeichnis *Tools* und dort im Unterverzeichnis *System Architect*.

MagicDraw

MagicDraw der Firma No Magic besticht als grafisches UML-Tool durch die wohl exakteste und umfangreichste grafische Umsetzung des UML 1.3-Standards. In der Version 5.1 unterstützt MagicDraw die Entwicklung im Team sowie zusätzlich die Modellierung von Enterprise Java Beans, Datenbanken sowie das Forward- und Reverse-Engineering für die Programmiersprachen Java und C++, IDL, Java Bytecode und Enterprise Java Beans. Darüber hinaus beherrscht MagicDraw das Mapping von Inhalten relationaler Datenbanken auf Klassen.

Einrichten eines Projekts

Rufen Sie MagicDraw unter Windows über die Start-Leiste *Programme/MagicDraw UML 5.1/MagicDraw UML 5.1* auf.

Auch MagicDraw organisiert die Modellierung in Projekten, die alle Elemente der Modellierung sowie die individuellen projektspezifischen Einstellungen der Entwicklungsumgebung zu einer Einheit bündeln.

Legen Sie über das Menü *Datei/Neues Projekt* ein neues Projekt an.

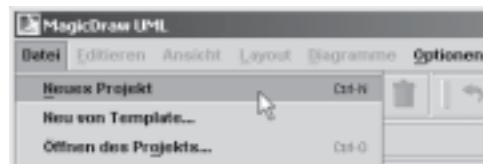


Abbildung 11.76: Neues Projekt anlegen

Im Gegensatz zu vielen anderen Tools müssen Sie keine Parameter bestimmen und können ohne Zeitverlust sofort mit der Modellierung beginnen. Ihr Projekt trägt bis zum ersten Speichern die Bezeichnung *Ohne Titel1*.

Das Hauptfenster zeigt im linken Bereich eine hierarchische Anordnung der Modellelemente.

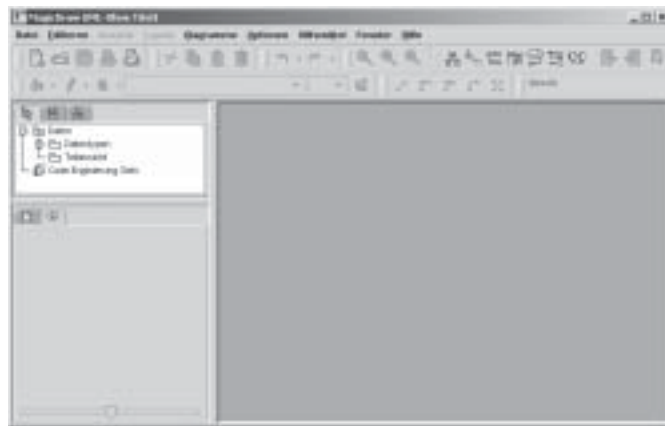


Abbildung 11.77: Projektstart

Auch MagicDraw unterscheidet das UML-Modell (*Daten*) von seiner grafischen Repräsentation (blauer Bereich) und dem erzeugten Code (*Code Engineering Sets*).

Erstellen von Klassendiagrammen

Wählen Sie das Menü *Diagramme/Klassendiagramme...*, um ein Klassendiagramm zu erstellen, das insbesondere die grafischen Fähigkeiten von MagicDraw andeutungsweise zeigt (siehe Abbildung 11.78).

MagicDraw startet einen Assistenten, der die notwendigen Angaben zum Erstellen einer Klasse erfragt (siehe Abbildung 11.79).

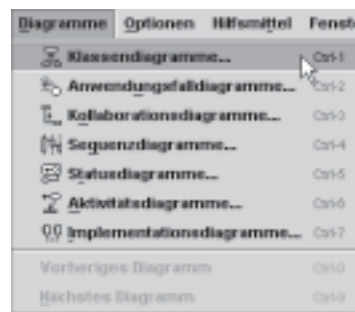


Abbildung 11.78: Klassendiagramm erstellen

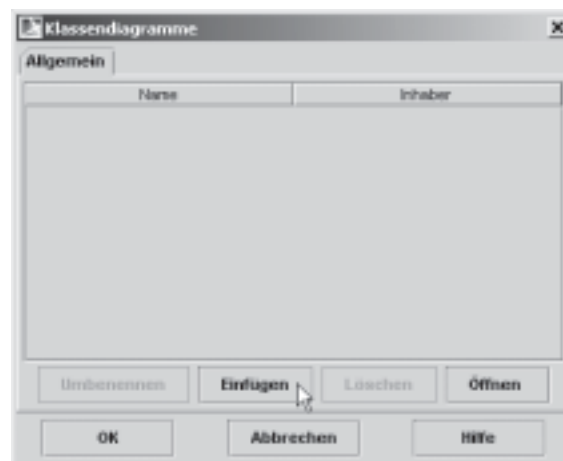


Abbildung 11.79: Klassendiagramm einfügen

Der Dialog bietet eine Übersicht aller Klassendiagramme, die im aktuellen Projekt vereinbart sind. In unserem Fall ist die Liste leer. Klicken Sie auf die Schaltfläche *Einfügen*, um ein Klassendiagramm zu erstellen.



Abbildung 11.80: Name für Klassendiagramm eingeben

Geben Sie *Anlage* als Namen für das Klassendiagramm ein und bestätigen Sie die Eingabe mit einem Klick auf den *OK*-Button.

Die Liste der erstellten Klassendiagramme wird sofort aktualisiert und zeigt das Klassendiagramm mit dem Namen *Anlage* an.

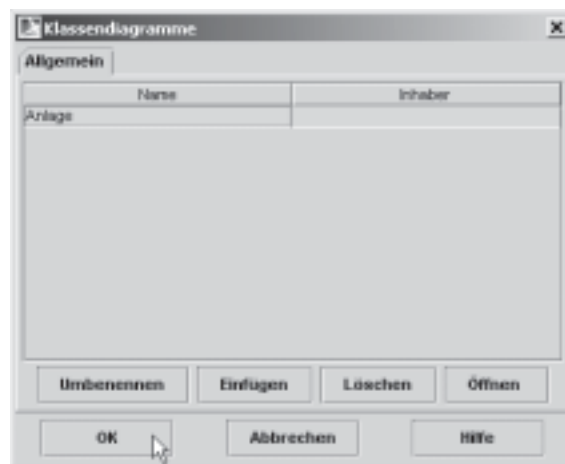


Abbildung 11.81: Klassendiagramm erstellen

Bestätigen Sie mit dem *OK*-Knopf, dass Sie das neu erstellte Diagramm beibehalten wollen, so gelangen Sie zurück zum Hauptfenster.

Auch MagicDraw teilt den Bildschirm in verschiedene Bereiche ein. Der schon von anderen Tools bekannte Explorer befindet sich auch bei MagicDraw im linken Bereich. Er stellt die Modellelemente und Quellcode-Informationen in einer hierarchischen Übersicht dar.

Darunter befindet sich ein so genannter Flieger, mit dem man schnell in großen Diagrammen navigieren kann, ohne die Übersicht zu verlieren. Der Flieger zeigt eine verkleinerte Gesamtsicht auf das Diagramm im Zeichenbereich, der sich rechts befindet. Den sichtbaren Ausschnitt können Sie auf der fixen Grafik verschieben. Der dargestellte Ausschnitt im rechten Zeichenfenster zeigt dann immer den links festgelegten sichtbaren Bereich.

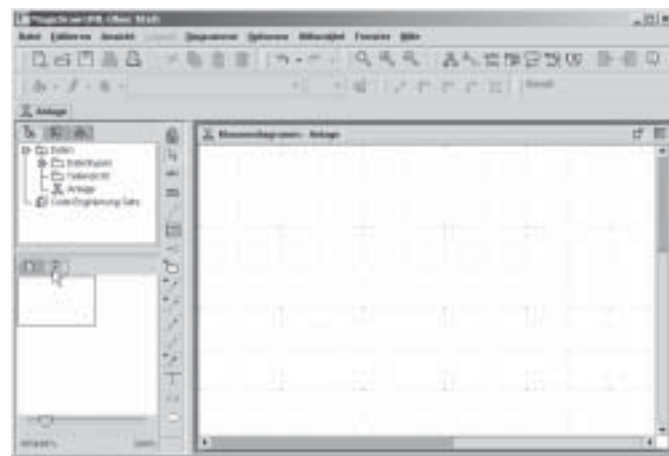


Abbildung 11.82: Hauptfenster mit Klassendiagramm

Der rechte Bereich zeigt ein gewähltes UML-Diagramm, wie gewohnt mit den Werkzeugen, die sich in der Werkzeugleiste links neben der Zeichenfläche befinden.

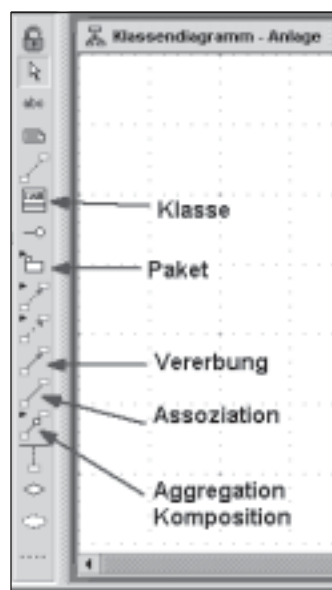


Abbildung 11.83: Zeichenwerkzeuge im Klassendiagramm

Die Abbildung zeigt die wichtigsten Symbole für *Klasse*, *Paket*, *Vererbung*, *Assoziation*, *Aggregation* und *Komposition*. Diese Werkzeuge werden wir im Folgenden einsetzen, um einige Klassen zu modellieren.

Zunächst erstellen Sie die Klasse *Konto*, die Ihnen aus vielen Kontexten des Buches bereits bekannt ist.



Abbildung 11.84: Neue Klasse vereinbaren

Klicken Sie dazu auf das *Class*-Symbol in der Werkzeugleiste und anschließend auf die Zeichenfläche, um eine Klasse im Modell und gleichzeitig als grafische Repräsentation im Klassendiagramm zu erzeugen.

Den Namen der Klasse (Konto) können Sie nach dem Erstellen direkt über die Tastatur eingeben, da die Klasse zu dem Zeitpunkt noch selektiert ist.

Um weitere Angaben zur Klasse in das Modell aufzunehmen, aktivieren Sie das Kontextmenü der Klasse Konto (Klick mit der rechten Maustaste auf das Klassensymbol). Lobenswert ist der saubere Aufbau dieses Menüs, das für fast jedes Element die Menüpunkte *Spezifikation* und *Symbol editieren* besitzt.

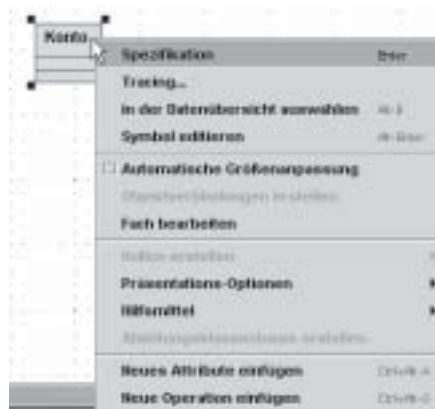


Abbildung 11.85: Trennung Visualisierung und Modell

Hier zeigt sich die saubere Trennung zwischen Modell (*Spezifikation*) und grafischer Darstellung (*Symbol editieren*), die MagicDraw vorbildlich umsetzt.

Wählen Sie den Menüpunkt *Spezifikation*, um Attribute und Operationen für die erstellte Klasse zu vereinbaren. Klicken Sie anschließend auf den Karteireiter *Attribute* und definieren Sie über den Button *Einfügen* ein neues Attribut.

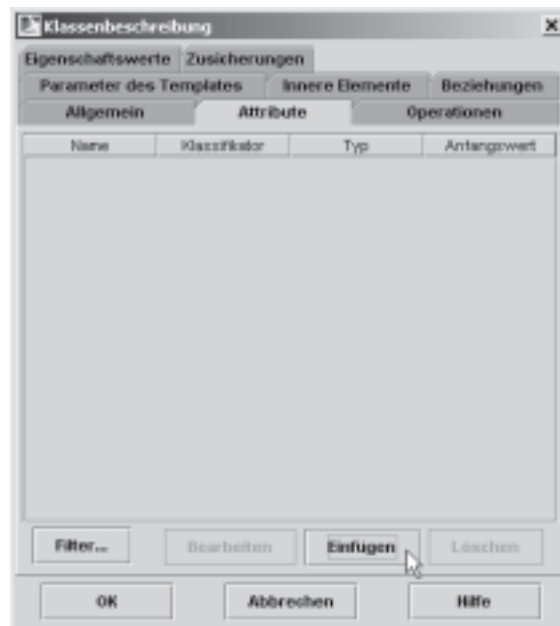


Abbildung 11.86: Attribut einfügen

MagicDraw startet einen Folgedialog zur Vereinbarung der Attributeigenschaften. Geben Sie im Eingabefeld *Name:* den Namen

kontostand ein. Weitere Angaben zu Typ, Modifikatoren und Sichtbarkeiten lassen wir außer Acht.

The screenshot shows the 'Attributbeschreibung' (Attribute Description) dialog box. The 'Allgemein' (General) tab is selected. The 'Name' field contains 'kontostand'. The 'Klasse' (Class) field contains 'Konto'. The 'Stereotyp' (Stereotype) field is set to '<kein Stereotyp>'. The 'Typ' (Type) field is set to '<Nicht spezifiziert>'. The 'Typ-Modifikator' (Type Modifier) field is set to 'changeable'. The 'Anfangswert' (Initial Value) field is empty. The 'Multiplizität' (Multiplicity) field is empty. Under 'Zugriffsmodifikator' (Access Modifier), the 'Private' radio button is selected. Under 'Gültigkeitsbereich' (Scope), the 'Instance' radio button is selected. There is a 'Dokumentation:' (Documentation) text area. At the bottom, there are buttons for 'Sprache der Eigenschaften' (Language of Properties), 'Inhaber' (Owner), 'OK', 'Abbrechen' (Cancel), and 'Hilfe' (Help). A mouse cursor is pointing at the 'OK' button.

Abbildung 11.87: Attribut-Optionen vereinbaren

Bestätigen Sie die gemachten Angaben durch einen Klick auf die Schaltfläche *OK*. Damit ist das Attribut *kontostand* in der Klasse *Konto* vereinbart.

Wählen Sie den Karteireiter *Operationen*. Die Festlegung erfolgt analog zur Attributdefinition über den Knopf *Einfügen*.

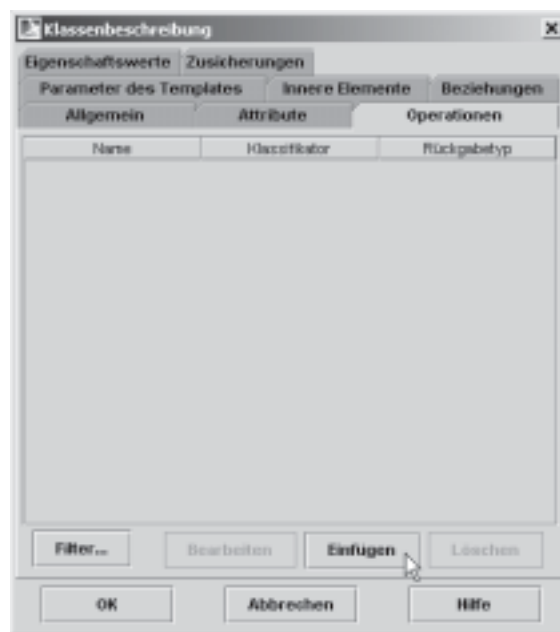


Abbildung 11.88: Operation einfügen

Auch hier startet MagicDraw einen Dialog zur Eingabe der Operations-Optionen. Legen Sie den Namen einzahlen fest. Von der Definition von Parametern, die sich hinter dem gleichnamigen Karteireiter verbergen, wollen wir bei dieser kurzen Einführung absehen.

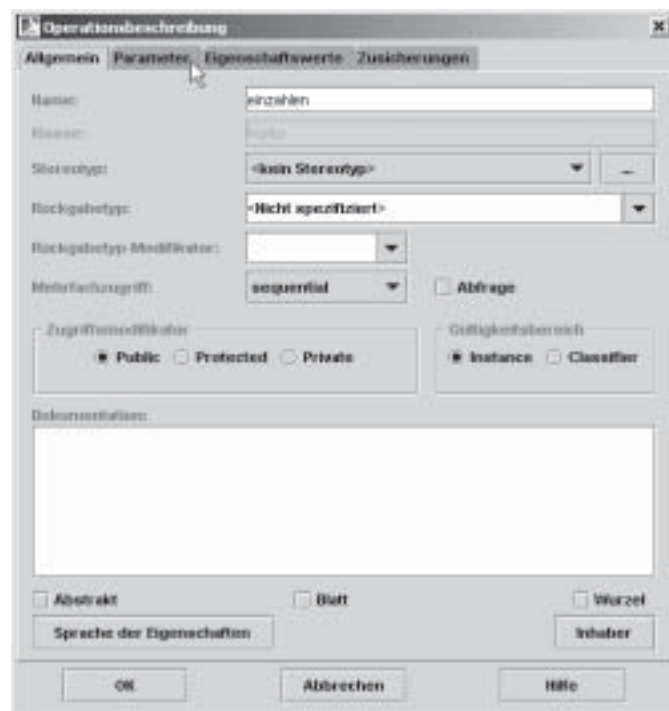
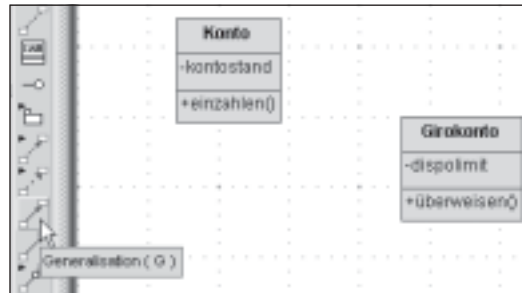


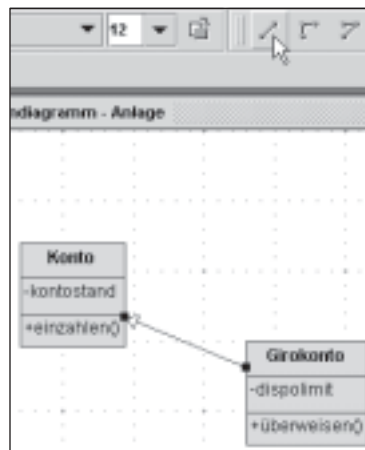
Abbildung 11.89: Operations-Optionen vereinbaren

Bestätigen wir die begonnenen Dialoge jeweils mit *OK*, so gelangen wir zurück zum Hauptfenster und sehen in den beiden unteren Segmenten des Klassensymbols das Attribut `kontostand` und die Operation `einzahlen()`.

Vereinbaren Sie eine Klasse `Girokonto` mit dem Attribut `dispolimit` sowie der Operation `überweisen()`.

**Abbildung 11.90:** Vererbung

Auch hier sehen wir die aktualisierte grafische Repräsentation. Um zwischen **Girokonto** und **Konto** eine Vererbungsbeziehung zu definieren, klicken Sie zur Auswahl auf das Vererbungssymbol in der Werkzeugleiste. Anschließend klicken Sie nacheinander auf die Unterklasse **Girokonto** und die Oberklasse **Konto**.

**Abbildung 11.91:** Feintuning des Vererbungspeils

Die Vererbungsbeziehung wird durch den Vererbungspfeil grafisch dargestellt. Um die rechtwinklige in eine geradlinige Verbindung umzuwandeln, klicken Sie auf die entsprechende Schaltfläche oberhalb der Zeichenfläche.

Zwischen einem Kunden und einem Girokonto vereinbaren wir eine Assoziation. Gehen Sie analog zur Vererbungsdefinitionen vor. Klicken Sie auf das Assoziations-Werkzeug sowie auf die Klassen Kunde und Girokonto.



Abbildung 11.92: Assoziation

Über das Kontext-Menü (rechte Maustaste) der Assoziation und *Spezifikation* gelangen Sie zu dem in Abbildung 11.93 dargestellten Dialog, in dem Sie den Namen und die Kardinalitäten der Assoziation vereinbaren und mit *OK* bestätigen können.

Im Hauptfenster sehen wir die vereinbarten Optionen. Alle im Klassendiagramm dargestellten Elemente finden ihre Entsprechung im Modell-Explorer. So finden Sie in der Datenhierarchie auch die Klasse Girokonto mit ihren Elementen (siehe Abbildung 11.94).



Alle Elemente im Zeichenbereich sind ebenfalls im Modell vereinbart. Umgekehrt muss nicht jedes Element des Modells in einem Diagramm dargestellt sein. Damit enthalten Diagramme immer eine unechte Untermenge der definierten Modell-Elemente.

Attributbeschreibung

Allgemein | Eigenschaftswerte | Zuordnungsgruppen

Name:

Skriptentyp:

Attributname der A:

Für Skriptdatei:

Default A:

Attributname der B:

Für Skriptdatei:

Default B:

Beschreibung des Attributs:

Abbildung 11.93: Assoziations-Optionen eingeben



Abbildung 11.94: Visuelle Sicht und Modell-Sicht

Die sauber definierte Struktur, die sich auch in einer übersichtlich und klar strukturierten Benutzungsoberfläche widerspiegelt, macht die Entwicklungsumgebung gerade für Einsteiger, die schnell produktiv arbeiten wollen, sehr interessant.

Vom Klassen- zum Objektdiagramm

MagicDraw integriert Objektdiagramme in Kollaborationsdiagramme. Im Klassendiagramm ist es dort nicht möglich, Objekte im Kontext der sie beschreibenden Klassen darzustellen.

Erstellen Sie ein Kollaborationsdiagramm über die Schaltfläche in der Toolbar.



Abbildung 11.95: Kollaborationsdiagramm erstellen I

In einem Dialogfenster vereinbaren Sie einen Namen für das Diagramm und bestätigen Ihre Eingabe mit dem OK-Button.

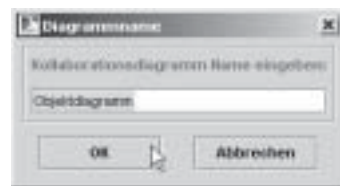


Abbildung 11.96: Kollaborationsdiagramm erstellen II

Eine neue Zeichenfläche erscheint, in der Sie zur Darstellung eines Objektdiagramms nur die beiden Werkzeuge zur Erstellung von Objekten und deren Verbindungen benötigen.

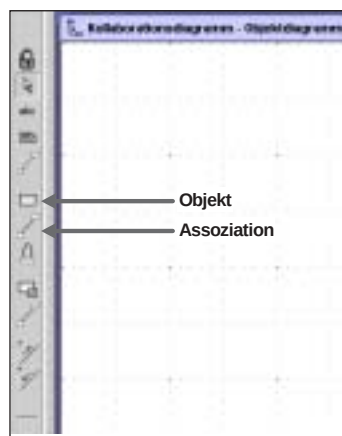


Abbildung 11.97: Werkzeuge im Kollaborationsdiagramm

Zur Erstellung eines Objekts wählen Sie zunächst das *Instanz*-Werkzeug aus und klicken anschließend auf die Zeichenfläche. Tippen Sie danach direkt den Objektnamen ein.

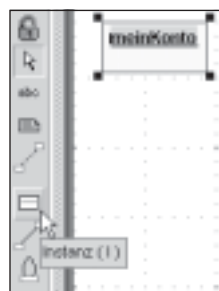


Abbildung 11.98: Kollaborationsdiagramm Objekt

Die Zuordnung des Objektes zur Klasse `Girokonto` vereinbaren Sie, indem Sie mit der rechten Maustaste auf das Objekt `meinKonto` klicken und im erscheinenden Menü *Klassifikator* und dann die Option `Girokonto` auswählen.

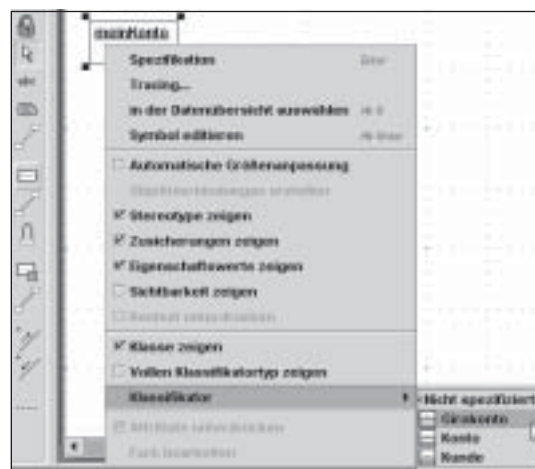


Abbildung 11.99: Kollaborationsdiagramm Objekt-Klasse II

Der festgelegte Typ erscheint dann hinter dem Objekt-Namen durch einen Doppelpunkt getrennt.



Abbildung 11.100: Kollaborationsdiagramm Objekt-Klasse II

Erstellen Sie ein zweites Objekt `einKunde` nach dem gleichen Vorgehen. Verbinden Sie die beiden Objekte mit dem *Link*-Werkzeug. Klicken Sie dazu nach Auswahl des Werkzeugs nacheinander auf die beiden Objekte `meinKonto` und `einKunde`.

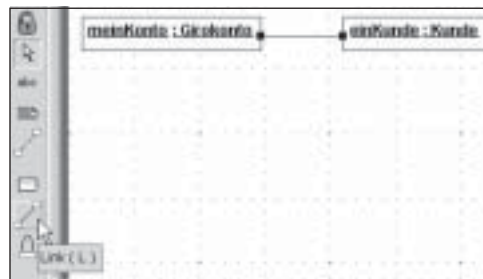


Abbildung 11.101: Kollaborationsdiagramm Assoziation

Die Objekte sind nun, wie in der Assoziation im Klassendiagramm festgelegt, miteinander verbunden.

Oft ist es sinnvoll, Attribute und Attributwerte für ein oder mehrere Objekte zu zeigen, um die Aussagekraft des Diagramms zu erhöhen (siehe Abbildung 11.102).

Klicken Sie dazu mit der rechten Maustaste auf das betreffende Objekt und schalten Sie im erscheinenden Menü die Option *Attribute unterdrücken* aus.

Attributwerte legen Sie am schnellsten fest, wenn Sie auf das nun angezeigte Attribut `dispolimit` klicken. Geben Sie nach dem Namen ein Gleichheitszeichen, gefolgt von dem Wert, der dem Attribut zugewiesen werden soll, ein (siehe Abbildung 11.103).

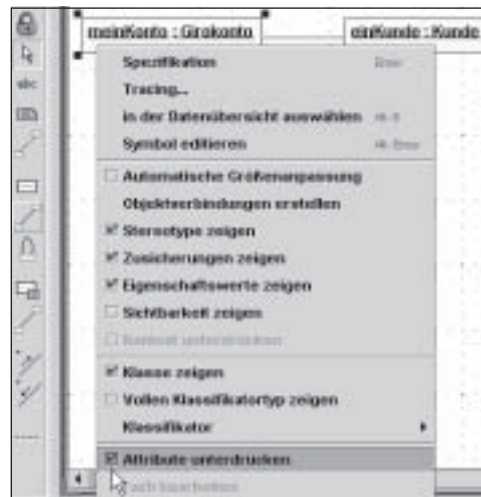


Abbildung 11.102: Kollaborationsdiagramm Attribute zeigen



Abbildung 11.103: Kollaborationsdiagramm Attributwerte

Überprüfen Sie die angezeigten Attribute, stellen Sie fest, dass nur das Attribut `dispolimit`, das in der Klasse `Girokonto` vereinbart ist, angezeigt wird. Das Attribut `kontostand` der Klasse `Konto`, das über die Vererbungsbeziehung zwischen den Klassen `Konto` und der Klasse `Girokonto` ebenfalls angezeigt werden müsste, bleibt jedoch unsichtbar.

Sie können das Attribut kontostand einblenden, wenn Sie zusätzlich neben der Klasse Girokonto einen zweiten Klassifikator, die Klasse Konto, vereinbaren.



Abbildung 11.104: Kollaborationsdiagramm Klassifikator I

Doppelklicken Sie auf das Objekt *meinKonto*, um die Spezifikation des Objektes zu editieren. Wählen Sie das Register *Zugewiesene Klassifikatoren* aus und drücken Sie den *Anweisen*-Knopf.



Abbildung 11.105: Kollaborationsdiagramm Klassifikator II

Wählen Sie im anschließenden Dialog die Klasse `Konto` als zusätzlichen Klassifikator aus.

Das Objekt `meinKonto` zeigt jetzt auch das Attribut `kontostand`.



Abbildung 11.106: Kollaborationsdiagramm Attribute

Das Diagramm lässt sich mit den gezeigten Verfahren um weitere Objekte und Verbindungen erweitern.

Vom Klassen- zum Zustandsdiagramm

Ein Zustandsdiagramm ist auch bei MagicDraw nicht syntaktisch an das Klassendiagramm gekoppelt, so dass auch dort keine automatische Zuordnung von Attributen und Operationen zu Zustän-

den und Zustandsübergängen möglich ist. Das entspricht der formalen Spezifikation der UML, ist jedoch für den Fall von Objekt-Lebenszyklen unpraktisch.

Wählen Sie im Menü *Diagramme* die Option *Statusdiagramme...* aus, um ein neues Zustandsdiagramm zu erstellen.



Die deutsche Übersetzung der Menüs und Befehle ist an manchen Stellen etwas holprig und vor allem nicht konsistent. Wenn Sie die englischen Begriffe der UML kennen, sollten Sie die Sprache der Entwicklungsumgebung umstellen.

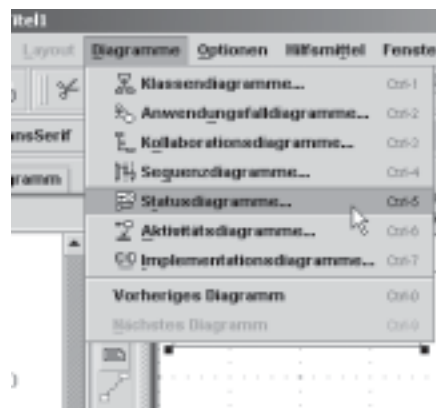


Abbildung 11.107: Erstellen eines Zustandsdiagramms I

Drücken Sie im Dialog *Zustandsdiagramme* den *Einfügen*-Knopf und legen Sie anschließend den Diagrammnamen fest. Bestätigen Sie beide Dialogfenster über den *OK*-Button.



Abbildung 11.108: Erstellen eines Zustandsdiagramms II

Eine neue Zeichenfläche, die alle wichtigen Werkzeuge zum Zeichnen von Zustandsdiagrammen enthält, wird im Zeichenbereich angezeigt.

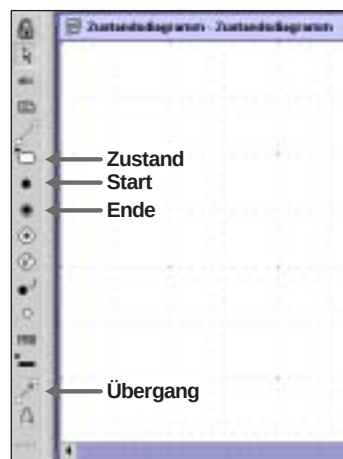


Abbildung 11.109: Werkzeuge im Zustandsdiagramm

Um elementare Zustandsdiagramme zu zeichnen, benötigen Sie nur die vier Werkzeuge, um Zustände, die Pseudozustände Start und Ende sowie Übergänge zwischen den Zuständen zu definieren.

Wählen Sie diese Werkzeuge nacheinander mit einem Mausklick aus und klicken Sie auf die Zeichenfläche, um die Elemente zu positionieren.

Zustände definieren Sie mit dem *Status*-Werkzeug. Möchten Sie spezielle Zustände vereinbaren, so können Sie mit der rechten Maustaste auf das Status-Werkzeug klicken. Sie erhalten dann eine Auswahl aller Zustandsarten, die Sie festlegen können.



Enthält das Symbol eines Zeichen-Werkzeugs in der linken oberen Ecke eine schwarze Pfeilspitze, so können Sie mit der rechten Maustaste das Zeichen-Werkzeug auf spezielle Zeichen-Werkzeuge umstellen. Jedes Symbol besitzt dann ein anderes spezielles Piktogramm.

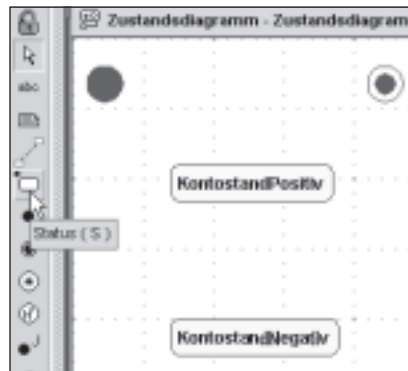


Abbildung 11.110: Zustandsdiagramm Zustände definieren

Tippen Sie direkt nachdem Sie einen Zustand erzeugt haben seine Bezeichnung ein.



Die Bezeichnung eines beliebigen Elements kann direkt über die Tastatur eingegeben werden, wenn das entsprechende Element markiert wurde.

Wählen Sie das *Transition*-Werkzeug aus, um die Übergänge zwischen den Zuständen zu definieren. Klicken Sie bei ausgewähltem *Transition*-Werkzeug zuerst auf den Ausgangs- und dann auf den Zielzustand.

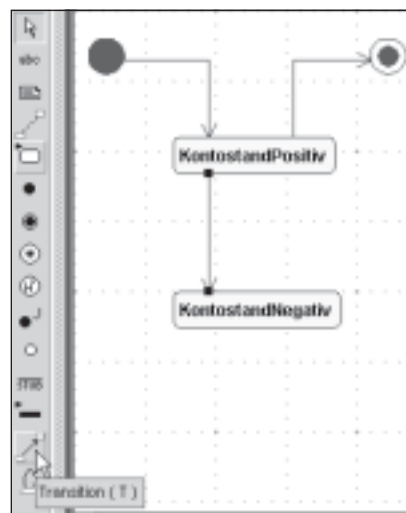


Abbildung 11.111: Zustandsdiagramm Übergänge

Benennen Sie Ihre Übergänge, indem Sie einen Übergangs-Pfeil auswählen und anschließend die -Taste drücken.

Im folgenden Dialog vereinbaren Sie alle Eigenschaften des markierten Übergangs.

Übergangsbeschreibung

Allgemein Eigenschaftswerte Zuweisungen

Name:

Stereotyp:

Quelle:

Ziel:

Auslöser

Typ:

Effekt

Typ:

Schutz

Ausdruck:

Dokumentation:

Abbildung 11.112: Zustandsdiagramm Übergang Name

Vereinbaren Sie zumindest einen Namen, der die inhaltliche Bedeutung eines Übergangs vermittelt. Dient das Zustandsdiagramm wie in dem hier abgebildeten Fall dazu, einen Objektlebenszyklus

zu beschreiben, so sind die Zustände durch Attributwerte und die Übergänge durch Operationsaufrufe dargestellt.

Ist der Übergang an eine *guard-condition*, die in der deutschen Übersetzung als Schutz bezeichnet ist, gebunden, so vereinbaren Sie diese Bedingung im Bereich *Schutz* über den *Bearbeiten*-Knopf. Nur wenn eine solche Bedingung erfüllt ist, findet der Zustandsübergang statt.

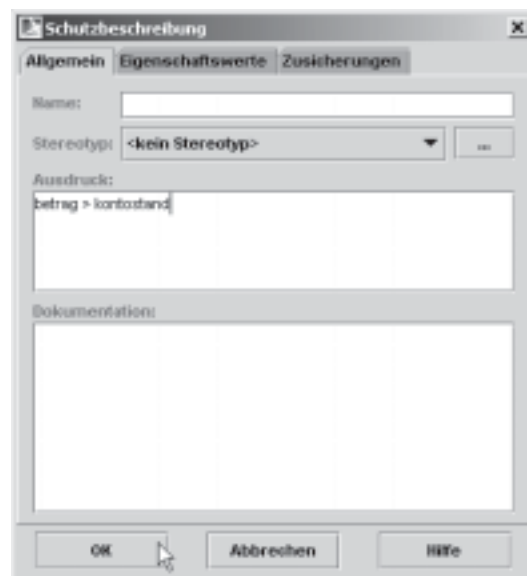


Abbildung 11.113: Zustandsdiagramm Übergang-Guard

Geben Sie die Bedingung in das *Ausdruck*-Feld im Dialog *Schutzbeschreibung* an. Beenden Sie den Dialog, indem Sie den *OK*-Button drücken.

Auf der Zeichenfläche sehen Sie jetzt den Namen `auszahlen()` über dem ausgewählten Zustand, nicht aber die Schutz-Bedingung. In MagicDraw gibt es keine Möglichkeit, diese Bedingung anzuzeigen. Sie können jedoch alternativ die Schutz-Bedingung als Constraint eintragen, der sich dann anzeigen lässt.

Vom Klassen- zum Sequenzdiagramm

Das Sequenzdiagramm ist auch in MagicDraw formal mit dem Klassendiagramm verknüpfbar.

Erstellen Sie ein Sequenzdiagramm über die Werkzeugleiste mit einem Klick auf das Sequenzdiagramm-Symbol.

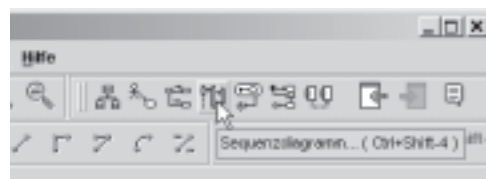


Abbildung 11.114: Sequenzdiagramm erstellen I

Vereinbaren Sie einen Namen für das neue Diagramm und klicken Sie auf den *OK*-Button, um die Eingabe zu bestätigen.



Abbildung 11.115: Sequenzdiagramm erstellen II

Das neu erzeugte Sequenzdiagramm erscheint auch als Zeichenfläche mit den obligatorischen Zeichenwerkzeugen.

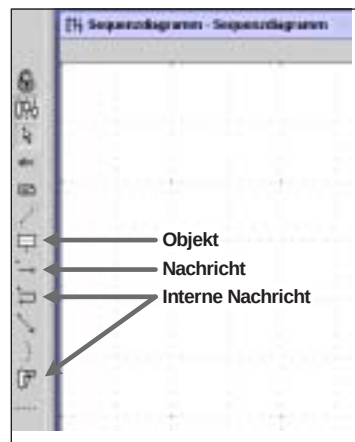


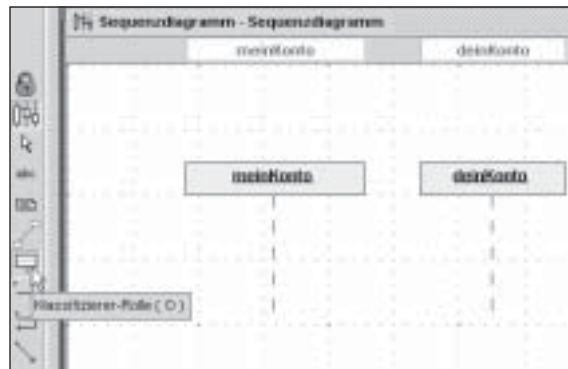
Abbildung 11.116: Werkzeuge im Sequenzdiagramm

Auch hier setzen wir nur die elementaren Zeichenwerkzeuge *Objekt*, *Nachricht* und *Interne Nachricht* ein, mit denen wir bereits sinnvoll arbeiten können.

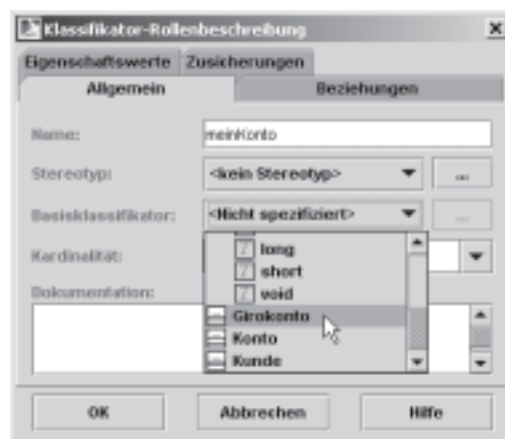
Positionieren Sie zunächst zwei Objekte auf der Zeichenfläche. Wählen Sie dazu das *Klassifizierer-Rolle*-Objekt aus. Die Bezeichnung *Klassifizierer-Rolle* deutet an, dass es sich bei den Objekten im Sequenzdiagramm um anonymisierte Objekte handelt, die stellvertretend für alle Objekte dieser Klasse eine Abfolge von Nachrichten darstellen (siehe Abbildung 11.117).

Die Namen tippen Sie bei markiertem Objekt, wie bei allen anderen Diagramm-Elementen auch, einfach ein.

Eine Verbindung zum Klassendiagramm stellen Sie her, indem Sie für die anonymisierten Objekte des Klassendiagramms einen Basisklassifikator festlegen, der angibt, aus welcher Klasse dieses Objekt abgeleitet ist.

**Abbildung 11.117:** Sequenzdiagramm Objekte erstellen

Per Doppelklick oder durch Drücken der -Taste bei markiertem Objekt gelangen Sie in den *Klassifikator-Rollenbeschreibung*-Dialog.

**Abbildung 11.118:** Sequenzdiagramm Objekte klassifizieren

Vereinbaren Sie als Basisklassifikator für das Objekt `meinKonto` die Klasse `Girokonto` und für das Objekt `deinKonto` die Klasse `Konto`.

Im nächsten Schritt wählen Sie das *Nachricht*-Werkzeug aus und definieren eine Nachricht, die vom Objekt `meinKonto` an das Objekt `deinKonto` sendet.

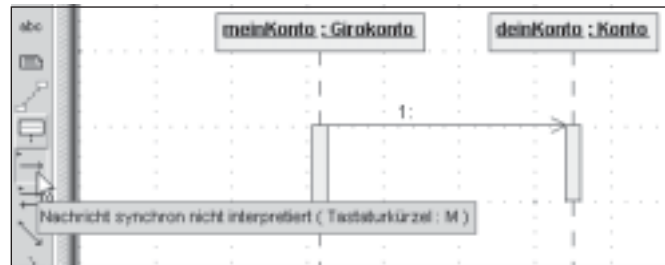


Abbildung 11.119: Sequenzdiagramm Nachricht erstellen

Da die Objekte einer Klasse zugeordnet sind, bietet MagicDraw die Möglichkeit, die Operationen dieser Klasse für einen Nachrichtenaufwurf direkt auszuwählen.

Klicken Sie mit der rechten Maustaste auf den Nachrichtenpfeil. Unter dem Menü *Aufrufaktion* werden alle Operationen der zugeordneten Klasse zur Auswahl angeboten. Wählen Sie die Operation `einzahlen()` aus (siehe Abbildung 11.120).



Um einen externen Akteur in ein Sequenzdiagramm einzufügen, vereinbaren Sie ein Use-Case-Diagramm und erstellen Sie dort einen Akteur. Diesen Akteur können Sie aus dem Modell-Explorer per Drag&Drop in ein Sequenzdiagramm übertragen.

Speichern Sie Ihr Projekt über das Menü *Datei/Projekt speichern*, mit anschließendem Speichern-Dialog und verlassen Sie danach das Programm über *Datei/Ausgang*.

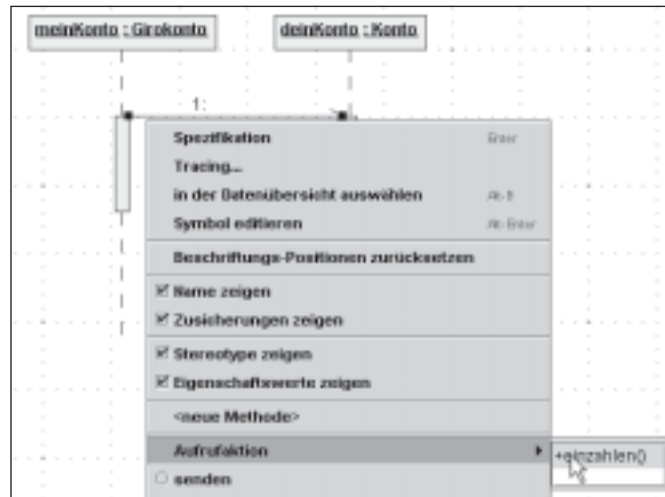


Abbildung 11.120: Sequenzdiagramm Operation festlegen

MagicDraw ist für Designer, die intuitive Bedienbarkeit schätzen und Wert auf eine exakte Umsetzung der UML-Spezifikation legen, ein hervorragend geeignetes Tool.

Poseidon

Poseidon von der Hamburger Gentleware AG ist das jüngste Tool in der Reihe der hier vorgestellten Programme. Es basiert jedoch auf der bekannten Open-Source Software ArgoUML und hat somit einen konsolidierten Kern.

Poseidon ist ein ganz auf UML und Java ausgerichtetes Werkzeug. Innerhalb dieses Bereiches weist das Programm jedoch viele innovative und gut durchdachte Ansätze zur Effizienzsteigerung der

Modellierung, aber auch der Handhabung des Werkzeugs selbst auf. Neben der hier beschriebenen Community Edition bietet die Gentleware AG weitere kommerzielle Versionen des Tools an.

Hervorzuheben ist der perspektivische Explorer, der die Modellelemente nach verschiedenen Kriterien anordnet. So können die Modellelemente nach den folgenden Gesichtspunkten geordnet werden.

- ✓ Pakete
- ✓ Diagramme
- ✓ Vererbung
- ✓ Klassen und Assoziationen
- ✓ Navigierbare Assoziationen
- ✓ Assoziationen

Darüber hinaus verwaltet Poseidon eine Tätigkeitsliste, deren Einträge ebenfalls perspektivisch sortiert werden können. Poseidon unterscheidet zwischen den Sichten:

- ✓ Priorität
- ✓ Entscheidung
- ✓ Ziel
- ✓ Verursacher
- ✓ Poster
- ✓ Wissenstyp

Zur Prüfung Ihrer erstellten Modelle stehen so genannte *Kritiken* zur Verfügung.

Erwähnenswert sind ebenfalls die so genannten *Rapid Buttons*, die ausgehend von einem grafischen Symbol schnell die Anbindung weiterer Elemente ermöglichen. Für Klassen werden Sie die Rapid Buttons kennen lernen.

Starten Sie Poseidon aus dem Windows Start-Menü über *Programme/Poseidon for UML Community Edition 1.2.1/Poseidon for UML CE*.

Wie Sie mit dem Registrierungsdialog verfahren, erklärt Ihnen Anhang B, der wichtige Informationen zur Installation und den Lizenzschlüsseln gibt.

Einrichten eines Projektes

Auch Poseidon arbeitet projektorientiert. Über das Menü *Datei/Neues Projekt* vereinbaren Sie ein neues Projekt. Nach dem ersten Start ist automatisch das Projekt *modell 1* verfügbar. Die Zeichenfläche für ein Klassendiagramm ist ebenso verfügbar.

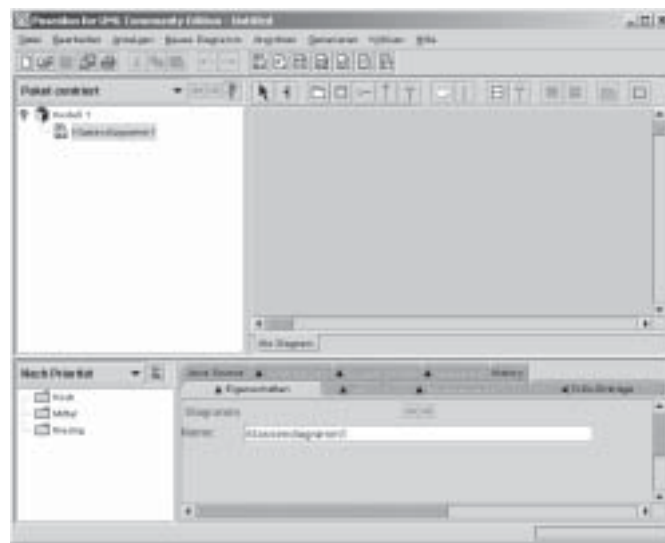


Abbildung 11.121: Hauptfenster

Poseidon unterteilt sein Hauptfenster ebenfalls in unterschiedliche Bereiche. Links finden wir die bereits erwähnten perspektivischen Werkzeuge Explorer (oben) und Tätigkeits-Liste (unten). Oben rechts befindet sich wie bei den anderen Tools auch die Zeichenfläche für Diagramme.

Unterhalb des Diagrammeditors finden Sie einen für viele Elemente sehr unübersichtlichen Bereich, den man als Eigenschaftseditor bezeichnen könnte. Hier stellen Sie Optionen für das ausgewählte Element des Diagrammeditors bzw. des Modell-Explorers ein.

Erstellen von Klassendiagrammen

Poseidon stellt eine Zeichenfläche für ein Klassendiagramm beim Anlegen eines neuen Projektes bereits zur Verfügung.

Die Werkzeugleiste befindet sich in Poseidon oberhalb des Diagrammeditors und enthält die auch von anderen Tools bekannten Elemente. Einige wichtige Symbole stellt die nächste Abbildung im Überblick dar.

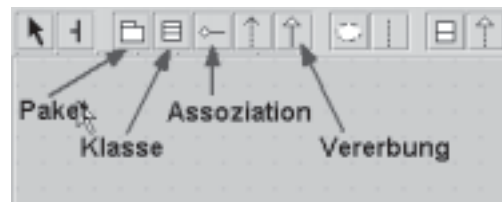


Abbildung 11.122: Werkzeuge im Klassendiagramm

Klicken Sie auf das Klassen-Symbol der Werkzeugleiste und anschließend in das Zeichenblatt, erscheint ein neues Klassensymbol.



Abbildung 11.123: Klasse definieren

Den Namen der Klasse vereinbaren Sie, indem Sie bei markiertem Klassensymbol die Zeichenfolge einfach über die Tastatur eintippen.

Ein Klick auf das Attribut-Segment des Klassensymbols ermöglicht das direkte Eingeben eines neuen Attributes über die Tastatur.



Abbildung 11.124: Attribut vereinbaren

Alternativ lassen sich Attribute und Operationen auch über den Eigenschaftsbereich unterhalb des Diagrammeditors festlegen. Um eine Operation hinzuzufügen, klicken Sie auf das entsprechende Symbol (Klassensymbol mit blau hinterlegtem Operations-Segment).

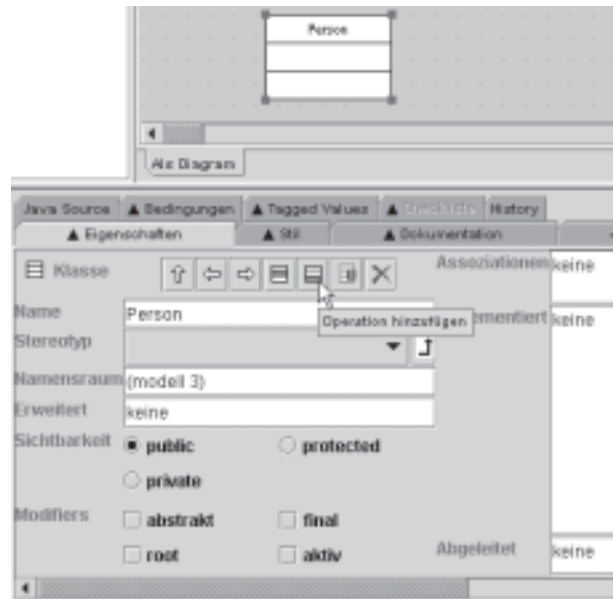


Abbildung 11.125: Operation hinzufügen

Der folgende Dialog legt die Operation `getName()` im Eigenschaftsbereich fest. Dazu geben Sie im Feld *Name* den Operationsnamen `getName` (ohne Klammern) an (siehe Abbildung 11.126).

Auch in diesem Dialog befindet sich die Schaltfläche, um weitere Operationen hinzuzufügen. Der Cursor zeigt in der obigen Abbildung auf dieses Symbol.

Die Community Edition erzeugt aus Klassendiagrammen automatisch Java Code. Die erzeugten Code-Skelette sind jedoch nicht editierbar. Sie sind im Eigenschaftseditor über den Karteireiter *Java Source* erreichbar, wenn eine Klasse, ein Attribut oder eine Operation aktuell ausgewählt sind (siehe Abbildung 11.127).

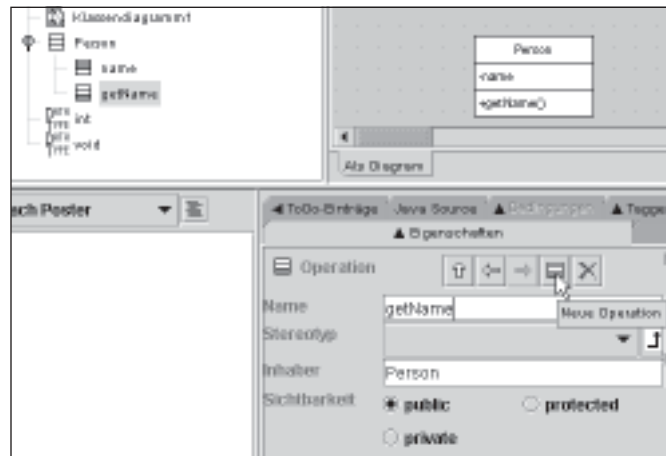


Abbildung 11.126: Operationsnamen festlegen

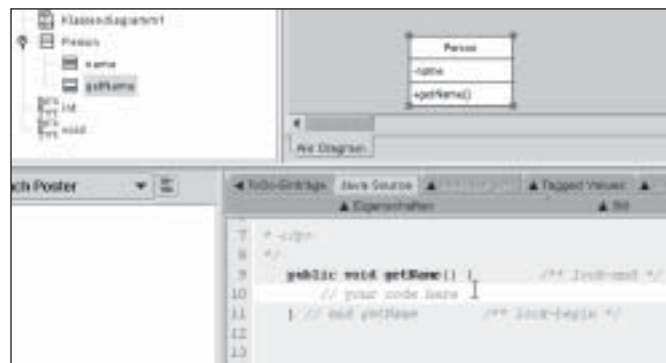


Abbildung 11.127: Java-Anbindung

Auswählen können Sie eine Klasse, ein Attribut oder eine Operation im Diagrammeditor oder alternativ auch im Modell-Explorer. Für die Operation `getName()`, die Sie durch einen Klick im Modell-Explorer auswählen, zeigt der Bereich unter dem *Java Source*-Karteireiter im Eigenschaftsbereich das erzeugte Java-Skelett.

Blau hinterlegte Bereiche sind fix und nicht editierbar. Operationsrumpfe sind in ihrem Inneren editierbar. Der weiße Hintergrund kennzeichnet solche Bereiche. So haben Sie die Möglichkeit, Ihre Klassen zu vervollständigen.

Um Vererbungsbeziehungen zu definieren, können die Rapid Buttons für Klassen eingesetzt werden. Klicken Sie dazu auf die Klasse *Mitarbeiter*. Bewegen Sie jetzt den Mauszeiger innerhalb der Klasse, so erscheinen vier Rechtecke, die an allen vier Seiten des Klassensymbols andocken.

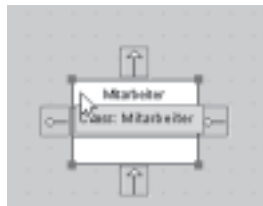


Abbildung 11.128: Rapid Button für Vererbung

Per einfachem Klick auf den oberen Rapid Button können Sie Oberklassen, die in einer Vererbungsbeziehung zur aktuellen Klasse stehen, vereinbaren. Durch einen Klick auf den unteren Button entsteht eine Unterklasse, die ebenfalls über den Vererbungspfeil mit der aktuellen Klasse verbunden ist. Über die Rapid Buttons links und rechts vom Klassensymbol erstellen Sie Assoziationen, Aggregationen und Kompositionen.

Im hier vorliegenden Fall bestehen die beiden Klassen *Mitarbeiter* und *Person* bereits, die über eine Vererbungsbeziehung miteinander

der verbunden werden sollen. Klicken Sie auf den oberen Rapid Button und ziehen Sie den Vererbungsfeil bei gedrückter Maustaste in die Klasse Person.

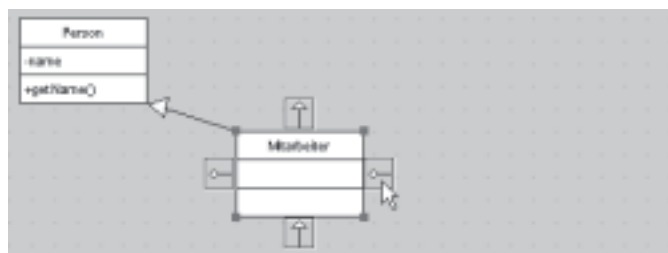


Abbildung 11.129: Rapid Button für Assoziationen I

Klicken Sie anschließend auf den rechten Rapid Button, um eine neue Klasse zu erzeugen, die mit der aktuellen Klasse Mitarbeiter über eine einfache Assoziation verbunden ist.

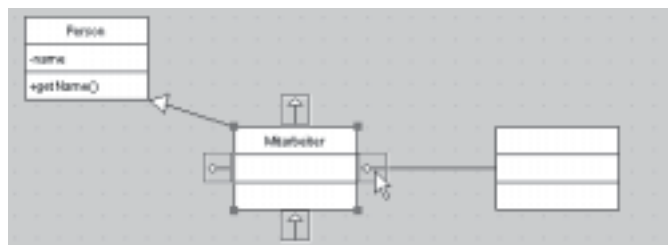


Abbildung 11.130: Rapid Button für Assoziationen II

Die neu erstellte Klasse erhält den Namen *Projekt*. Im Eigenschaftsbereich tragen Sie im Feld *Namen* die Bezeichnung *arbeitet in* als Assoziations-Namen ein.



Abbildung 11.131: Assoziations-Name

Oben rechts im Eigenschaftsbereich markiert der Mauszeiger das Assoziationsende *Mitarbeiter*. Ein einfacher Klick öffnet einen Dialog zur Vereinbarung von Optionen für das Assoziationsende, das in die Klasse *Mitarbeiter* mündet.

Im Auswahlménü *Multiplizität* wählen Sie *1..**, um festzulegen, dass einem Projekt mindestens ein Mitarbeiter zugeordnet wird (siehe Abbildung 11.132).

Die Kardinalität auf der *Projekt*-Seite können Sie alternativ auch über das Kontextménü, das Sie für das Assoziationsende aktivieren, mit der Auswahl von *Multiplizität/0..1* festlegen. Damit kann ein Mitarbeiter höchstens einem Projekt zugeordnet werden (siehe Abbildung 11.133).

Ausgehend von diesem Klassendiagramm lassen sich weitere Diagramme in Poseidon erstellen.



Abbildung 11.132: Kardinalität im Eigenschaftsbereich



Abbildung 11.133: Kardinalität per Kontextmenü

Vom Klassen- zum Objektdiagramm

Objektdiagramme lassen sich in Poseidon nur im Kollaborationsdiagramm darstellen. Klicken Sie dazu auf das *Kollaborationsdiagramm*-Symbol in der Werkzeugleiste.

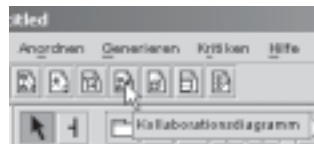


Abbildung 11.134: Kollaborationsdiagramm erstellen

Oberhalb der neu angelegten Zeichenfläche erscheint eine Werkzeugleiste mit den folgenden Zeichenwerkzeugen.



Abbildung 11.135: Kollaborationsdiagramm Werkzeuge

Die Assoziation ist hier eine Verbindung zwischen Objekten. Da sie sich formal aus der Assoziation, die im Klassendiagramm vereinbart ist, herleitet, übernehmen wir diese Bezeichnung auch für das Objektdiagramm.

Wählen Sie das Objekt-Werkzeug (*ClassifierRole*) aus und klicken Sie danach auf die Zeichenfläche (siehe Abbildung 11.136).

Den Namen des Objektes *leiter* tragen Sie im *Eigenschaften*-Register im *Namen*-Feld ein.

Erstellen Sie die in der nächsten Abbildung gezeigten Objekte nach dem gleichen Vorgehen (siehe Abbildung 11.137).

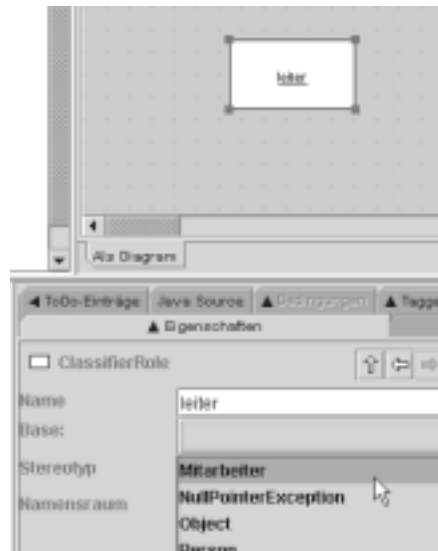


Abbildung 11.136: Kollaborationsdiagramm Klasse



Abbildung 11.137: Kollaborationsdiagramm Objektverbindung I

Um die Objektverbindungen anzuzeigen, klicken Sie einmal auf das Ausgangsobjekt, bewegen den Mauszeiger auf den Rand des

Objekt-Symbols, klicken auf einen Rapid-Button und ziehen die Objektverbindung bei gedrückter Maustaste bis zum Ziel-Objekt.

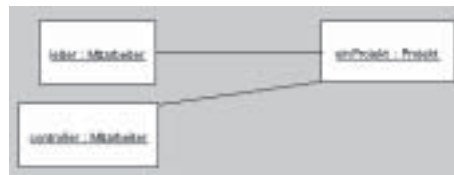


Abbildung 11.138: Kollaborationsdiagramm Objektverbindung II

Im Ergebnis entsteht ein Projekt, dem zwei Mitarbeiter zugeordnet sind..

Vom Klassen- zum Zustandsdiagramm

Ein Zustandsdiagramm ist in Poseidon ausschließlich zur Erklärung eines Anwendungsfalls oder einer Klasse verfügbar. Folgerichtig erwartet Poseidon, dass Sie einen Anwendungsfall oder eine Klasse markieren, bevor Sie ein Zustandsdiagramm erstellen können.

Markieren Sie die Klasse *Projekt* auf der Zeichenfläche des Klassendiagramms.

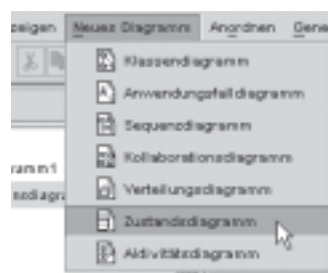


Abbildung 11.139: Zustandsdiagramm erstellen

Eine Zeichenfläche für das neu erstellte Diagramm wird zusammen mit der zugehörigen Werkzeugleiste eingeblendet.

Auch hier finden Sie die für das Zustandsdiagramm obligatorischen Zeichen-Werkzeuge für Pseudo-Zustände, Zustände und Übergänge.



Abbildung 11.140: Zustandsdiagramm Werkzeuge

Wählen Sie mit einem Klick das *Zustand*-Werkzeug aus. Klicken Sie auf die Zeichenfläche, um einen Zustand im Diagramm zu positionieren.



Abbildung 11.141: Zustandsdiagramm Zustand einfügen

Tippen Sie für den ausgewählten Zustand den Namen *vorgeschlagen* über die Tastatur ein. Er erscheint dann der UML-Konvention entsprechend im oberen Bereich des *Zustand*-Symbols.

Vereinbaren Sie drei weitere Zustände *freigegeben*, *in Realisierung* und *abgeschlossen*. Die elementaren Phasen eines Projektes werden jetzt durch Zustände abgedeckt.



Abbildung 11.142: Zustandsdiagramm Zustände benennen

Im nächsten Schritt vereinbaren Sie die Zustandsübergänge, die dann zusammen mit den Zuständen einen typischen Ablauf eines Projektes verdeutlichen.



Abbildung 11.143: Zustandsdiagramm Übergang

Wählen Sie dazu das *Transition*-Werkzeug aus und klicken Sie anschließend auf den Ausgangszustand vorgeschlagen und danach auf den Zielzustand freigegeben.

Beschriften Sie den Übergang mit *Freigabe prüfen*. Geben Sie die Bezeichnung im Eigenschaftsfenster im Eingabefeld *Name* ein.



Abbildung 11.144: Zustandsdiagramm Übergang Bezeichnung

Erstellen Sie weitere Übergänge sowie die Pseudozustände Start und Ende. Poseidon bietet leider keine Möglichkeit, interne Übergänge zu vereinbaren.



Poseidon bietet außer der obligatorischen Zuordnung des gesamten Diagramms zu einer Klasse oder einem Anwendungsfall-diagramm keine formale Bindung von Zuständen an Attributwerte oder von Übergängen an Operationen.

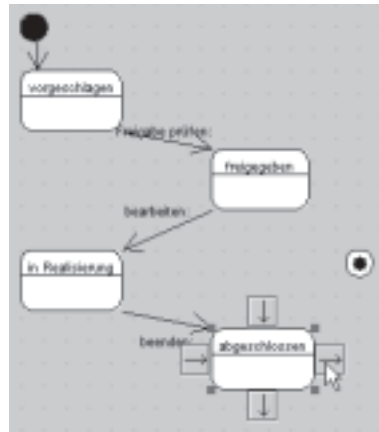


Abbildung 11.145: Zustandsdiagramm Übergang Rapid-Button

Besonders schnell vereinbaren Sie einen Übergang, indem Sie einen Zustand markieren und den Mauspfel zum Rand des Zustand-Symbols bewegen.

Markieren Sie den Zustand *abgeschlossen*. Bewegen Sie den Mauszeiger zum rechten Rand des Zustand-Symbols. Klicken Sie auf den Rapid-Button, der einen ausgehenden Pfeil zeigt und ziehen Sie bei gedrückter Maustaste den Übergangspfeil bis auf den Pseudozustand Ende. Lassen Sie die Maustaste los.

Jetzt zeigt das Zustandsdiagramm vier grundlegende Zustände eines Projekts und seine Zustandsfolge.



Guard-Conditions oder Constraints können in Poseidon für Übergänge nicht vereinbart werden.

Vom Klassen- zum Sequenzdiagramm

Erstellen Sie ein neues Sequenzdiagramm, indem Sie im Menü *Neues Diagramm* die Option *Sequenzdiagramm* auswählen.

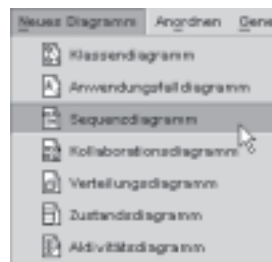


Abbildung 11.146: Sequenzdiagramm erstellen

Die neu erstellte Zeichenfläche enthält elementare Zeichen-Werkzeuge, mit denen die Elemente auf der Zeichenfläche erstellt werden.

Neben dem Nachricht-Symbol können Sie weitere spezielle Nachrichtenarten auswählen.



Abbildung 11.147: Werkzeuge im Sequenzdiagramm

Erstellen Sie ein Objekt, indem Sie das *Objekt*-Werkzeug auswählen und anschließend auf die Zeichenfläche klicken. Dort erscheint ein Objekt mit einem Aktivitätsbalken.



Abbildung 11.148: Sequenzdiagramm Objekt

Im nächsten Schritt vereinbaren Sie den Namen und den Typ des Objekts.



Abbildung 11.149: Sequenzdiagramm Objekt-Name

Geben Sie für das markierte Objekt im *Name-Feld* des Registers *Eigenschaften* die Bezeichnung *einProjekt* ein. Der Name erscheint auch im Objekt-Symbol auf der Zeichenfläche, gefolgt von einem Doppelpunkt.

Vereinbaren Sie danach den Typ des Objektes. Dazu wählen Sie die Klasse aus, aus der das Objekt erzeugt wurde.

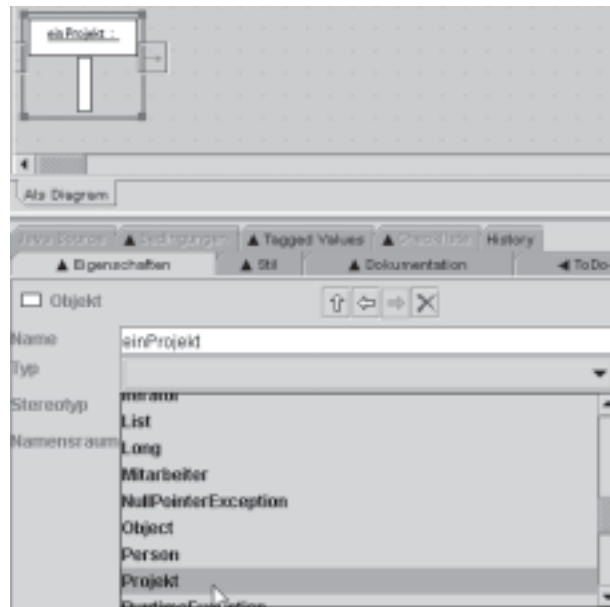


Abbildung 11.150: Sequenzdiagramm Objekt-Typ

Wählen Sie dazu im *Eigenschaften*-Register in der Auswahlliste *Typ* die Klasse *Projekt* aus. Der ausgewählte Typ wird auch im Objekt-Symbol im Diagramm hinter dem Objektnamen angezeigt.

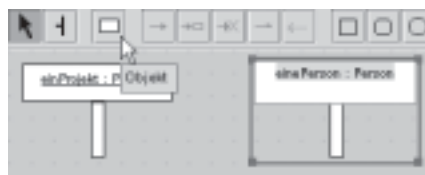


Abbildung 11.151: Sequenzdiagramm Objekt editieren

Vereinbaren Sie ein zweites Objekt mit dem *Objekt*-Werkzeug. Name und Typ des Objektes können Sie alternativ direkt in das markierte Objekt eingeben. Geben Sie dabei zuerst den Objektname *einePerson*, danach einen Doppelpunkt und dahinter den Typ *Person* an. Diese Angaben werden auch in die entsprechenden Stellen des *Eigenschaften*-Registers übernommen.

Das Zusammenspiel der Objekte vereinbaren Sie mit dem *Stimulus-Call*-Werkzeug, das einen Nachrichten-Pfeil erzeugt.



Abbildung 11.152: Sequenzdiagramm Nachricht

Wählen Sie das *StimulusCall*-Werkzeug aus und klicken Sie mit der Maus nacheinander auf die Lebenslinien des Ausgangsobjekts *einProjekt* und des Zielobjekts *einePerson*.

Der neu erzeugte Nachrichten-Pfeil verbindet die Aktivitätsbalken der beteiligten Objekte miteinander.

Um die Nachricht zu beschriften, markieren Sie die Bezeichnung *:new action* oberhalb des Nachrichten-Pfeils. Geben Sie in das Feld *Aktion* die Bezeichnung *Bericht erstellen* ein.



Eine formale Bindung der Nachrichten an die Operationen kann Poseidon nicht vereinbaren, obwohl die Objekte formal an eine Klasse gebunden werden können.

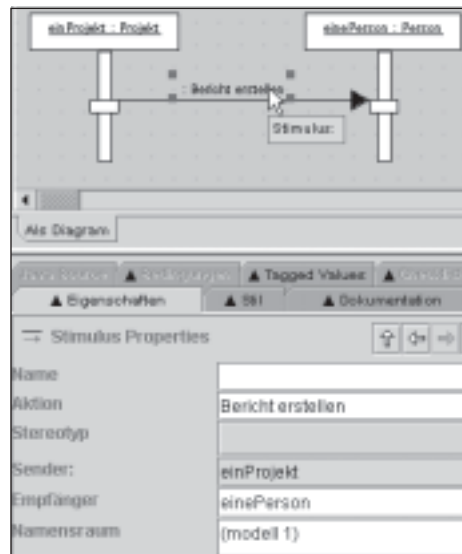


Abbildung 11.153: Sequenzdiagramm Aktion

Speichern Sie Ihr Projekt über das Menü *Datei/Projekt speichern...* und folgen Sie dem anschließenden Datei-Dialog. Damit haben Sie das Handling von Poseidon grundlegend kennen gelernt. Beenden Sie die Anwendung, indem Sie das Menü *Datei/Beenden* auswählen.

Nachdem Sie die auf der CD zusammengestellten Tools kennen gelernt haben, stellt der folgende Abschnitt noch einmal die Eigenschaften der Tools einander gegenüber, die für den UML-Modellierer wichtig sind.

Toolübersicht

In den folgenden Tabellen haben wir wesentliche Eigenschaften der vorgestellten Tools in einer Übersicht zusammengefasst. Weitere

Informationen über die Werkzeuge erhalten Sie in den jeweiligen Dokumenten auf der CD-ROM oder über die Internet-Seiten der Hersteller.

Tool	Hersteller / Internet-Adresse
Together	TogetherSoft http://www.togethersoft.de
Rational Rose	Rational Software http://www.rational-software.de
System Architect	Popkin Software http://www.popkin.com
Magic Draw	No Magic http://www.nomagic.com
Poseidon	Gentleware http://www.gentleware.de

Tabelle 11.1: Hersteller und Internet-Adressen

Die folgende Übersicht zeigt die von den Tools unterstützten UML-Diagramme und Programmiersprachen.

Tool	Unterstützte UML-Diagramme	Forward Engineering	Reverse Engineering
Together	Use Cases Klassendiagramme Zustandsdiagramme Aktivitätsdiagramme Sequenzdiagramme Kollaborationsdiagramme Komponentendiagramme Verteilungsdiagramme	C++ Java C# Visual Basic CORBA IDL	C++ Java C# Visual Basic CORBA IDL

Tool	Unterstützte UML-Diagramme	Forward Engineering	Reverse Engineering
Rational Rose	Use Cases Klassendiagramme Zustandsdiagramme Aktivitätsdiagramme Sequenzdiagramme Kollaborationsdiagramme Komponentendiagramme Verteilungsdiagramme	C++ Visual Basic	
System Architect	Use Cases Klassendiagramme Zustandsdiagramme Aktivitätsdiagramme Sequenzdiagramme Kollaborationsdiagramme Komponentendiagramme Verteilungsdiagramme	C++ Java Visual Basic JavaScript Delphi Smalltalk Power Builder	C++ Java (jeweils nur Klassendiagramme)
Magic Draw	Use Cases Klassendiagramme Zustandsdiagramme Aktivitätsdiagramme Sequenzdiagramme Kollaborationsdiagramme Komponentendiagramme Verteilungsdiagramme	C++ Java CORBA IDL	
Poseidon	Use Cases Klassendiagramme Zustandsdiagramme Aktivitätsdiagramme Sequenzdiagramme Kollaborationsdiagramme Komponentendiagramme Verteilungsdiagramme		Java

Tabelle 11.2: Diagrammarten und Code-Generierung

Die folgende Übersicht gibt Ihnen Information darüber, ob das jeweilige Tool auch die Modellierung von Geschäftsprozessen unterstützt, für welche Plattformen es verfügbar ist und welche Besonderheiten zu erwähnen sind.

Tool	Geschäftsprozess-Analyse	Plattformen	Sonstiges
Together	ja	Windows 95/98 Windows 2000 Windows NT 4.0 AS/400 Linux Macintosh Solaris	Metriken Patterns GUI-Builder Debugging Versionsverwaltung Datenbanken EJB
Rational Rose	ja	Windows 98 Windows XP Prof. Windows 2000 Windows NT 4.0 Linux Red Hat 6.2, 7.0 Sun Solaris 2.5.1, 2.6, 7, 8 HP-UX 10.20, 11.00, 11.11i SGI IRIX 6.5.5 AIX 4.3.2, 4.3.3 True64 4.0f.	Schnittstelle MS Visual Source-Safe Booch- und OMT-Diagramme
System Architect	ja	Windows 95/98 Windows ME Windows 2000 Windows NT 4.0	
Magic-Draw		Java	Import und Export von Rational Rose-Modellen

Tool	Geschäftsprozess-Analyse	Plattformen	Sonstiges
Poseidon		Java	Community Edition ist frei verfügbar Basiert auf der bekannten Open-Source-Software ArgoUML

Tabelle 11.3: Weitere Merkmale

Legen Sie bei der Evaluation von Tools Kriterien und Rahmenbedingungen fest und erarbeiten Sie eine Evaluierungsstrategie, die auf Ihre individuellen organisatorischen und technischen Rahmenbedingungen zugeschnitten ist.

Wir wünschen Ihnen viel Freude und vor allem Erfolg bei der Evaluation der Tools.





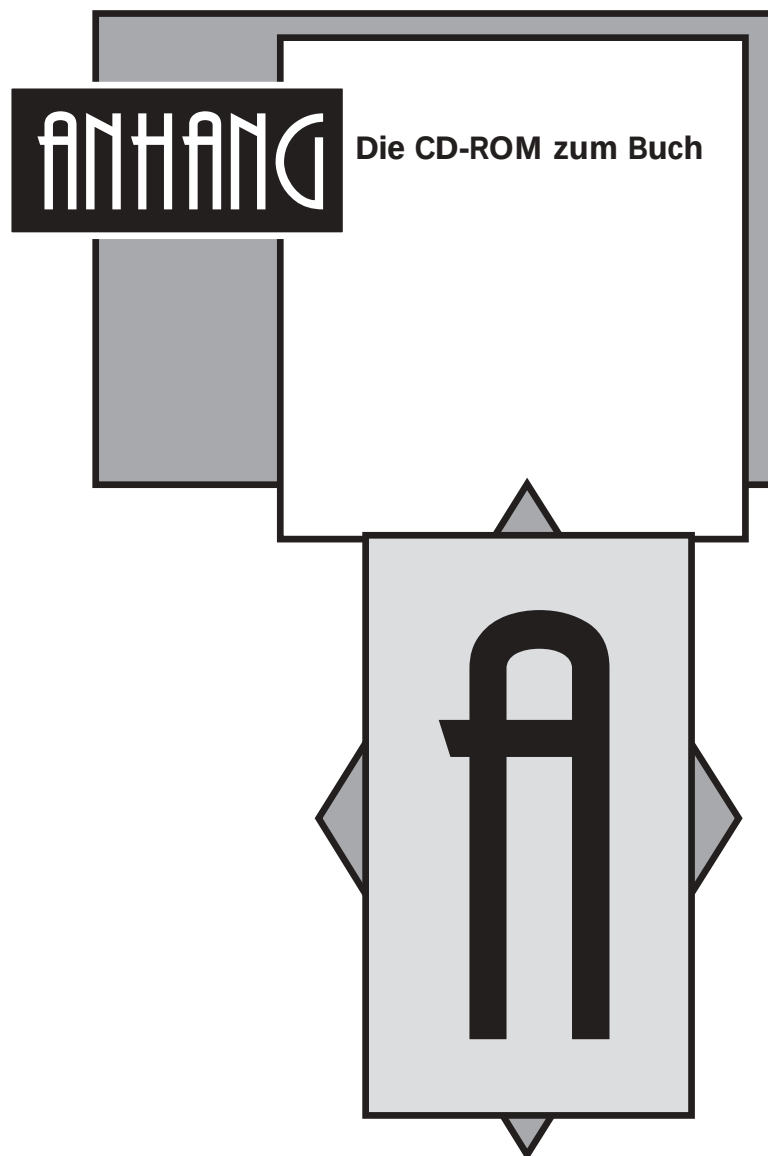
TEIL

Anhang

In diesem Anhang finden Sie u.a. Informationen über die CD-ROM, die diesem Buch beiliegt. Zunächst erläutern wir Ihnen die Installation der mitgelieferten Entwicklungswerkzeuge auf der CD. Darüber hinaus enthält der Anhang auch ein Glossar mit den wichtigsten Begriffen der Objektorientierung im Allgemeinen und der UML im Besonderen.









Die CD-ROM zum Buch

Auf der beiliegenden CD-ROM finden Sie einige Evaluationsversionen der objektorientierten Entwicklungswerkzeuge, die wir Ihnen in diesem Buch kurz vorstellen.

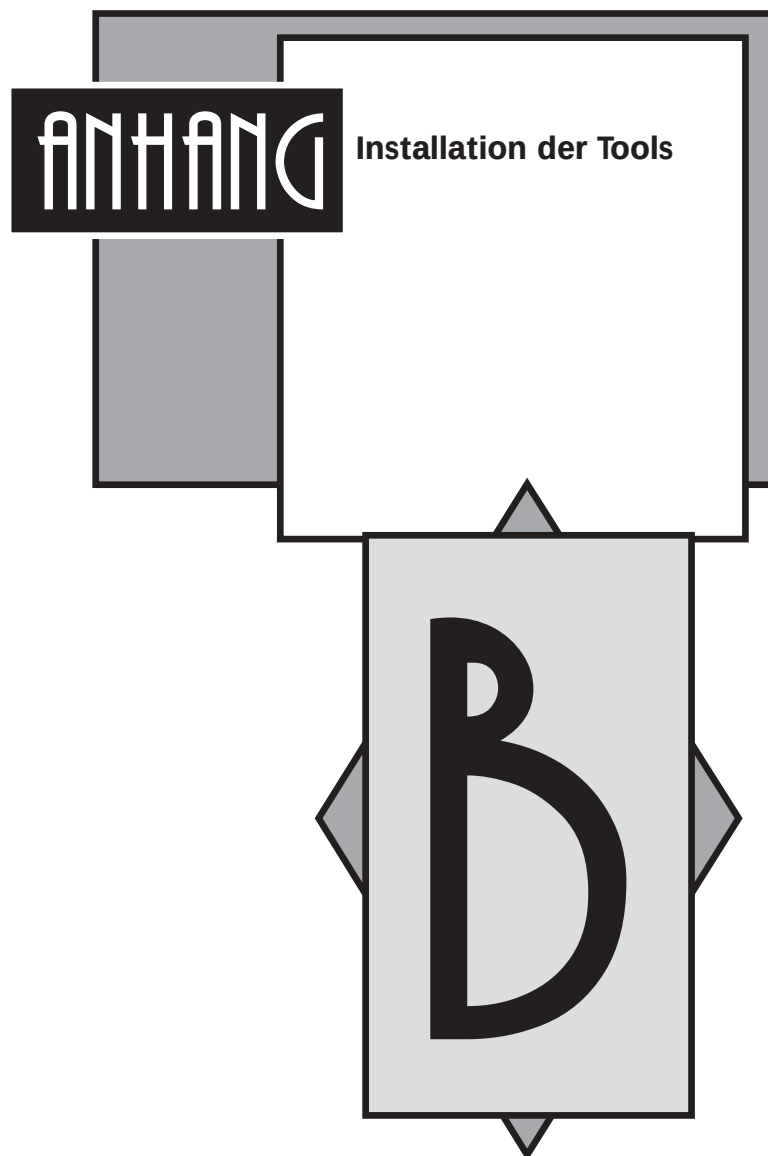
Wir haben uns bemüht, Ihnen eine Auswahl guter und in der Praxis weit verbreiteter Werkzeuge zur Verfügung zu stellen. Mit Hilfe der Evaluationsversionen erhalten Sie einen Eindruck von den jeweils verfügbaren Funktionen und der Arbeit mit der Software.

Wenn Sie sich intensiver mit einem Tool auseinander setzen möchten, sollten Sie das jeweilige Tutorial der Hersteller zu Rate ziehen. Dort erhalten Sie ausführliche Erklärungen der Programmfunktionen. Auf den Internetseiten der Hersteller, deren Adressen in Anhang C aufgelistet sind, finden Sie weitere nützliche Informationen.

Die Dateien der Entwicklungswerkzeuge sind in dem Ordner *Tools* abgelegt. Jedes Tool befindet sich wiederum in einem eigenen Verzeichnis. Anhang B erklärt die Vorgehensweise bei der Installation der Software-Werkzeuge. Sollten Sie darüber hinaus weitergehende Informationen benötigen, so finden Sie in den jeweiligen Verzeichnissen Hilfedateien der Software-Hersteller. Sie geben detaillierte Auskünfte über Hardware- und Software-Anforderungen. Außerdem sind dort Tutorials abgelegt, die Ihnen die ersten Schritte bei der Arbeit mit den Tools erleichtern.

Beachten Sie bitte auch die Datei *Readme.txt*. Sie liefert Ihnen weitere Informationen über die auf der CD befindlichen Programme und gibt wichtige Hinweise zu Lizenzbestimmungen.







Installation der Tools

Die folgenden Abschnitte unterstützen Sie bei der Installation der UML-Tools, die sich auf der CD-ROM zum Buch befinden bzw. im Internet zum Download bereitstehen.

Together

Together 6.0 von der Firma TogetherSoft finden Sie auf der mitgelieferten CD-ROM im Ordner *Tools* und darin im Unterverzeichnis *Together*.

Um mit dem Programm arbeiten zu können, müssen Sie es von der CD-ROM installieren. Zusätzlich benötigen Sie einen Lizenzschlüssel, der Ihnen eine Evaluation über einen Monat ermöglicht.

Wenden Sie sich in dieser und allen weiteren Fragen an Frau *Marion Eickmann* (Marketing Manager Germany). Sie erreichen Frau Eickmann per E-Mail mit Angabe des Stichworts *UML-Buch Erler/Ricken* unter marion.eickmann@togethersoft.com. Im Internet finden Sie TogetherSoft Deutschland unter <http://www.togethersoft.de>.

Beachten Sie die im Folgenden aufgeführten Systemanforderungen und gehen Sie bei der Installation des Programms und des Lizenzschlüssels nach der darauf folgenden Installationsanleitung vor.

Systemanforderungen

TogetherSoft bietet seine Model-Build-Deploy-Plattform Together 6.0 für verschiedene Betriebssysteme an. Auf der CD finden Sie die Windows-Version.

Aus Platzgründen enthält die CD nur die Windows-Version. Die Linux-Variante sowie die Versionen für andere Unix-Derivate und Macintosh können Sie als Evaluierungs-Version auf der Webseite von TogetherSoft unter <http://www.togethersoft.de> abrufen. Hier finden Sie auch jederzeit die aktuelle Version der Windows-Variante.

Verwenden Sie Together in der Windows-Version, sollte Ihr System laut Herstellerangaben die folgenden Voraussetzungen erfüllen:

Ressource	Spezifikation	Bemerkungen
CPU	Pentium, 233 MHz oder schneller	Empfohlen: Pentium III, 500 MHz oder schneller
Hauptspeicher	254 MB erforderlich	512 MB sind empfohlen (aber nicht erforderlich), wenn projektübergreifende Dokumentationen von großen Projekten erstellt werden
Festplattenspeicher	NTFS: 100 MB FAT: 95-120 MB abhängig von der Clustergröße	
Video	SVGA, Minimum High Color Modus	Empfohlene Auflösung: 1024x768 oder mehr
OS	Windows NT 4.0® Windows 2000® Windows 98® Windows 95®	Service Pack 4 oder größer für Windows NT 4.0
Java VM oder JDK	(SDK 1.3, wird mitgeliefert)	Erforderlich für Together®
HTML-Browser	Netscape Navigator 4.5 oder höher; oder Microsoft Internet Explorer 4.0 oder höher	Erforderlich zum Anzeigen und Drucken der HTML Hilfe-Dokumentation und selbst generierter HTML Dokumentationen.

Ressource	Spezifikation	Bemerkungen
CD-ROM-Laufwerk oder Internet Verbindung	entfällt	Erforderlich zum Installieren / Downloaden und Aktualisierung von Informationen

Tabelle B.1: Systemanforderungen an Together

Wenn Together® unter Windows eingesetzt werden soll, benötigen Sie *Sun Java® 2 SDK* (früher JDK), um einige Entwicklungs-Features nutzen zu können. *SDK 1.3* wird auf dem Windows-System automatisch installiert und Together® läuft unter seiner JRE.

Installation

Starten Sie das Installationsprogramm *together_1762v5.exe*, das Sie im Verzeichnis *Tools/Together/* der beiliegenden CD-ROM finden. Nach dem Aufruf erscheint das folgende Fenster auf Ihrem Bildschirm.

**Abbildung B.1:** Installationsvorbereitung

Warten Sie, während das Installationsprogramm die Installation vorbereitet. Achten Sie darauf, dass möglichst keine Programme oder Tasks im Hintergrund laufen und die Performance senken.

**Abbildung B.2:** Start der Installation

Klicken Sie auf den *Next*-Button, um die Installation fortzuführen. Im nächsten Fenster können Sie die Lizenzbedingungen von Together lesen und mit *I accept...* bestätigen.

**Abbildung B.3:** Lizenzbedingungen

Wählen Sie wieder die *Next*-Schaltfläche, um die Installation fortzusetzen.



Abbildung B.4: Installationspfad

Übernehmen Sie den vom Programm vorgeschlagenen Installationspfad *C:\TogetherSoft\Together6.0*. Alle weiteren Dateioperationen im folgenden Text beziehen sich auf diesen Pfad. Sollten Sie durch direkte Eingabe in das Eingabefeld oder den Dialog über die Schaltfläche *Choose...* ein anderes Programm-Verzeichnis bestimmen, so müssen Sie auch alle weiteren Dateioperationen in dem von Ihnen vereinbarten Verzeichnis vornehmen.

Gehen Sie wieder mit *Next* zum nächsten Fenster (siehe Abbildung B.5).

Together selbst ist in der Sprache Java implementiert und benötigt daher eine virtuelle Java-Maschine. Together enthält bereits eine passende Version des *Java 2 SDK 1.3*. Wählen Sie daher die erste Option *Install a Java VM specifically for this application*, um Versionskonflikten, die bei der Verwendung anderer Virtueller Maschinen auftreten können, aus dem Weg zu gehen.



Abbildung B.5: Auswahl der Java Virtual Machine



Verwenden Sie auf keinen Fall eine VM des *Java 2 SDK 1.4*, da damit nicht alle Optionen von Together nutzbar sind.

Der *Next*-Knopf führt Sie zum nächsten Schritt der Installation. Hier können Sie bestimmen, ob unter Windows eine neue Programmgruppe angelegt wird und ob die für alle registrierten User verfügbar ist (siehe Abbildung B.6).

Wählen Sie die erste Option *In a new Programm Group: Together 6.0*. Setzen Sie ein Häkchen vor *Create Icons for All Users*, damit alle weiteren registrierten User Together nutzen können.

Next führt Sie zum nächsten Schritt. Hier können Sie wählen, welche Dateinamenserweiterungen mit Together verknüpft werden sollen (siehe Abbildung B.7).



Abbildung B.6: Programmgruppe wählen



Abbildung B.7: Dateiverknüpfungen wählen

Wählen Sie eine Option an, so können Sie per Doppelklick auf eine zugehörige Datei diese mit Together zusammen starten. Aus Together heraus lassen sich unabhängig von der hier getroffenen Wahl alle Dateitypen bearbeiten. Die Verknüpfung macht nur dann Sinn, wenn sie einen bestimmten Dateityp ausschließlich mit Together bearbeiten.

Mit dem *Next*-Knopf gelangen Sie zum letzten Schritt der Installation. Hier können Sie festlegen, ob eine für die Testwerkzeuge von Together notwendige Bibliothek dem *JDK* hinzugefügt werden soll.

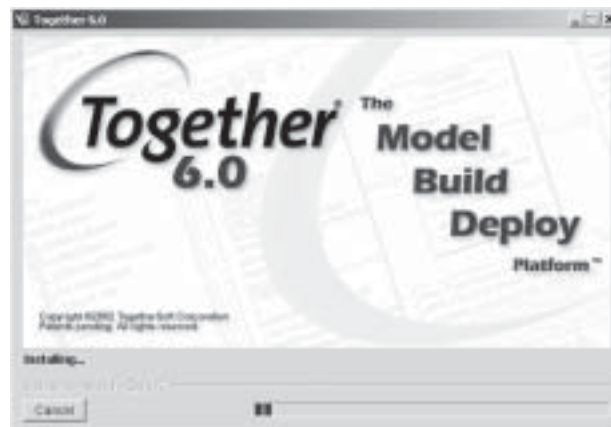


Abbildung B.8: Erweiterung des JDK

Diese Option können Sie bestätigen und gelangen mit *Next* zum Abschluss der Eingabe von Optionen für die Installation.

**Abbildung B.9:** Installations-Einstellungen

Mit Klick auf den *Install*-Button startet die Installation, die einige Zeit in Anspruch nimmt.

**Abbildung B.10:** Installationsprozedur

Letzte Aktion ist die Wahl der Sichtbarkeit und Anordnung der Werkzeuge. Je nachdem, welche Schwerpunkte Sie bei der Arbeit mit Together setzen, bietet das Tool für die Benutzertypen Business Modeller, Designer, Developer und Programmierer unterschiedlich angeordnete Benutzungsoberflächen an. Wählen Sie den Developer, da hier die Oberfläche so ausgerichtet ist, dass die meisten Optionen im schnellen Zugriff liegen.



Abbildung B.11: Ausrichtung der Benutzungsoberfläche

Betätigen Sie die *Next*-Schaltfläche, um zum nächsten Dialog zu gelangen (siehe Abbildung B.12).

Ist das Häkchen gesetzt, erhalten Sie Informationen über Together (siehe Abbildung B.13).



Abbildung B.12: Systeminformationen anfordern



Abbildung B.13: Readme-Datei mit Systeminformationen

Wechseln Sie, wenn Sie die Informationen zu Together studiert haben, zurück zur Installationsroutine, um mit *Done* im letzten Dialogabschnitt, die Installation abzuschließen.



Abbildung B.14: Abschluss der Installation

Damit ist das System vollständig installiert. Zusätzlich benötigen Sie einen Lizenzschlüssel, den Sie bei TogetherSoft wie bereits beschrieben erhalten.

Sie erhalten als Lizenzschlüssel eine Datei mit dem Namen *license.tg*. Kopieren Sie die erhaltene Lizenzschlüssel-Datei in das Verzeichnis *C:\TogetherSoft\Together6.0\bin*. Sollten Sie ein anderes Programmverzeichnis gewählt haben, so kopieren Sie die Datei in das entsprechende Verzeichnis. Achten Sie darauf, dass Sie die Datei auch dort im Unterverzeichnis *bin* ablegen.

Starten Sie jetzt Together über das Start-Menü *Start-Programme-Together 6.0-Together*. Erhalten Sie als Ergebnis den folgenden Fehlerdialog, sollten Sie überprüfen, ob die Lizenzschlüssel-Datei *license.tg* im korrekten Verzeichnis abgelegt ist.

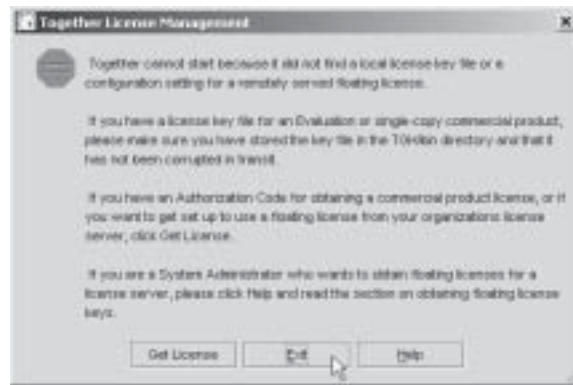


Abbildung B.15: Start ohne Lizenzschlüssel

Wenn Sie die Lizenzschlüssel-Datei im korrekten Verzeichnis abgelegt haben, und Together über das Start-Menü *Start-Programme-Together 6.0-Together* aktivieren, so sollte folgender Startbildschirm erscheinen.



Abbildung B.16: Start mit korrektem Lizenzschlüssel

Nachdem die Fortschrittsanzeige signalisiert, dass der Lizenzschlüssel korrekt installiert ist, erscheint der folgende Startbildschirm. Die Installation war erfolgreich.

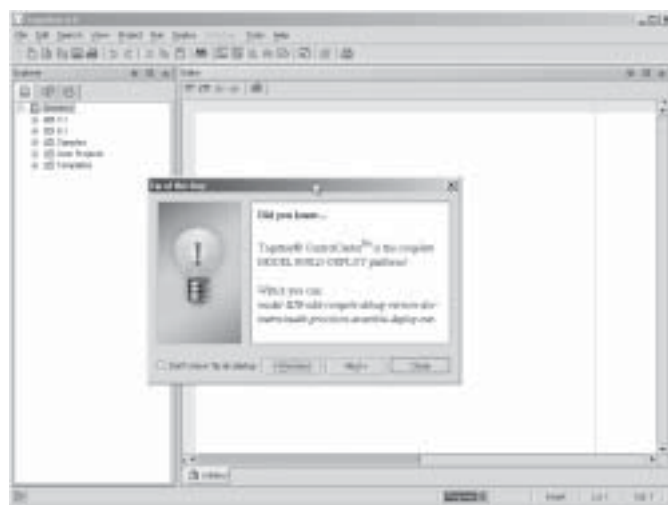


Abbildung B.17: Eingangsbildschirm

Eine kurze Einführung in Together erhalten Sie in Teil IV.

Rational Rose

Die auf dieser CD befindlichen Dateien erlauben Ihnen die Installation von Rational Rose Enterprise 2001A auf Ihrem Rechner. Sie finden diese auf der CD-ROM in dem Ordner *Tools* und dort in dem Unterverzeichnis *Rational Rose*.

Um mit dem Programm arbeiten zu können, benötigen Sie zusätzlich einen Lizenzschlüssel. Er berechtigt Sie, das Programm 15 Tage lang zu evaluieren. Sie können den Schlüssel im Internet unter der Adresse <http://www.rational-software.de/eval-keys> bestellen. Er wird Ihnen dann per E-Mail zugesandt.

Falls Sie weitere Fragen zu der Software haben, können Sie sich per Telefon unter der Nummer +49 (0)89-628 38-0 an die Firma Rational Software wenden.

Systemanforderungen

Das Programm stellt bestimmte Mindestanforderungen an Ihr System, die erfüllt sein sollten, damit die Software ordnungsgemäß funktioniert und eine gute Performance aufweist. Diese Anforderungen betreffen Ihre Hardware und das darauf installierte Betriebssystem.

Die folgende Tabelle enthält eine Übersicht der erforderlichen Hard- und Software-Ausstattung. Weitere Informationen finden Sie auch in der Datei *rose_relnote.html*, die sich unter den gepackten Dateien auf der CD befindet.

Systembestandteile	Mindestanforderungen
Betriebssystem	Windows 98 SE Windows NT 4.0 - Service Pack 6 Windows 2000 - Service Pack 1 Windows ME
Hardware	Pentium-Prozessor mit mindestens 300 MHz Taktfrequenz (empfohlen werden 600 MHz) mindestens 128 MB RAM (256 MB empfohlen) Hinweis: Die Anforderungen an die Größe des Arbeitsspeichers steigen mit zunehmender Größe Ihrer Objektmodelle Mindestens 500 MB Festplattenspeicher (230 Rose, 270 Installation)
Monitor	SVGA-kompatibler Monitor mit 256 oder mehr Farben und einer Auflösung von mindestens 1024 x 768 Pixel empfohlen
Internet-Browser	Microsoft Internet Explorer 4.01 oder höher Einige Add-Ins von Rational Rose benötigen einen Internet-Browser (z.B. Web-Publisher). Die Software unterstützt hierzu die meisten gängigen Browser. Allerdings erfordern einige Komponenten bestimmte Dienste des Microsoft Internet Explorer. Falls Rational Rose bei der Installation nicht mindestens den Internet Explorer in der Version 4.01 entdeckt, wird daher das Installations- programm abgebrochen. Fall Sie den Internet Explorer zunächst installieren möchten, erhalten Sie ihn unter der Internet-Adresse http://www.microsoft.com/windows/ie
Java	Rational Rose unterstützt die folgenden Java-IDEs (verwenden Sie das Java Development Kit, das zu Ihrer IDE passt):

Systembestandteile	Mindestanforderungen
	VisualCafe in der Studio 4 Professional Edition VisualCafe 4 Enterprise Edition VisualCafe 4 Expert Edition IBM VisualAge for Java Professional Edition IBM VisualAge for Java Enterprise Edition Forte for Java Internet Edition 2.0 Forte for Java Community Edition 2.0 JBuilder 4.0 Enterprise Edition JBuilder 4.0 Professional Edition JBuilder 4.0 Foundation Edition
Rose VC++	erfordert Microsoft Visual C++ 5.0 oder 6.0 (Visual Studio 6.0)
Rose Visual Basic	verlangt Microsoft Visual Basic 6.0 (Visual Studio 6.0)
Rose J	erfordert JDK 1.1.6 oder höher
Dokumentation	Adobe Acrobat Reader 4.x oder höher für die PDF-Dateien der Dokumentation Kostenloser Download des Acrobat Reader unter http://www.adobe.de

Tabelle B.2: Systemanforderungen

Installation

Zur Installation von Rational Rose entpacken Sie zunächst die Datei mit dem Namen *RationalRoseEnterpriseEdition.2001A.04.00.293.00* in einem neuen Verzeichnis auf Ihrer Festplatte. Achten Sie darauf, dass Sie beim Entpacken auch die Pfadangaben wiederherstellen.

Unter den entpackten Dateien finden Sie das Programm *Setup*, das Sie durch den Installationsprozess führt. Nachdem Sie dieses Programm aufgerufen haben, erscheint folgendes Fenster auf Ihrem Bildschirm.



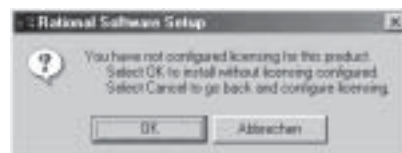
Abbildung B.18: Startfenster

Bestätigen Sie den Start der Installation, indem Sie auf den Button mit der Aufschrift *Weiter* klicken.

Die folgende Produktauswahl bestätigen Sie ebenfalls durch einen Mausklick auf die Schaltfläche *Weiter*. Alternativ können Sie auch einfach die -Taste drücken.

**Abbildung B.19:** Produktauswahl

Anschließend müssen Sie Ihre Auswahl noch einmal mit dem Button **OK** bestätigen.

**Abbildung B.20:** Bestätigung der Auswahl

Der folgenden Lizenzvereinbarung stimmen Sie zu, indem Sie das Optionsfeld *Yes, I accept the Agreement* auswählen und dann erneut den Button *Weiter* anklicken. Falls Sie nicht zustimmen, wird das Installationsprogramm abgebrochen.



Abbildung B.21: Lizenzvereinbarung

Anschließend können Sie das Verzeichnis auswählen, in dem Rational Rose installiert werden soll. Die Installation aller erforderlichen Dateien benötigt etwa 762 MB Speicherplatz auf Ihrer Festplatte. In der Compact-Version brauchen Sie nur etwa 75 MB. Das Programm zeigt Ihnen an, wie viel Platz benötigt wird und über welche freie Kapazität das von Ihnen gewählte Verzeichnis noch verfügt.

Wählen Sie ein Verzeichnis mit ausreichender Kapazität aus und bestätigen Sie Ihre Auswahl wiederum durch einen Mausklick auf den Button *Weiter* (siehe Abbildung B.22).

Das Programm wird während des Installationsvorgangs einige gemeinsam genutzte Dateien Ihres Betriebssystems aktualisieren. Im nächsten Fenster werden diese Dateien aufgelistet (siehe Abbildung B.23).

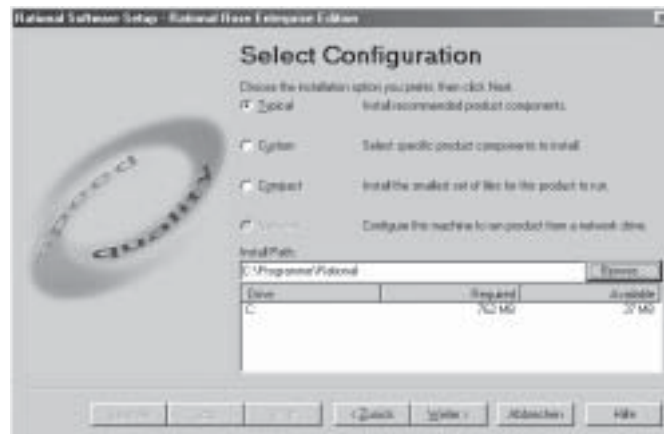


Abbildung B.22: Konfiguration und Verzeichnis wählen

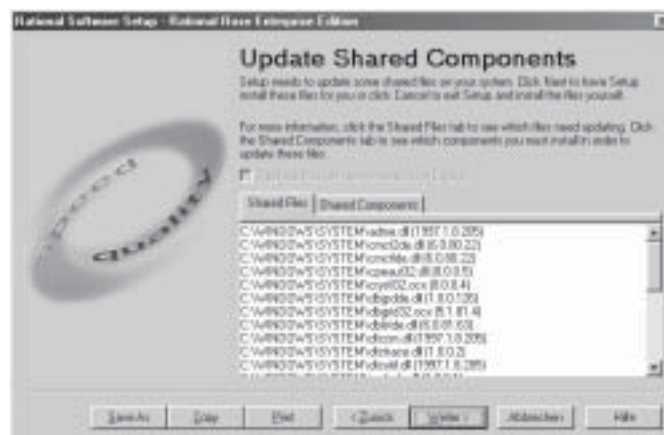


Abbildung B.23: Dateien aktualisieren

Mit einem Mausklick auf die Schaltfläche *Shared Components* in der Mitte des Fensters erhalten Sie nähere Informationen über die Komponenten, die von dieser Aktualisierung betroffen sind.



Abbildung B.24: Komponenten

Fahren Sie erneut durch einen Mausklick auf die Schaltfläche *Weiter* mit dem Installationsprogramm fort.

Abschließend zeigt das Programm Ihnen noch einmal die von Ihnen gewählten Optionen für den Installationsprozess an.



Abbildung B.25: Bestätigung der Optionen für die Installation

In einem Fenster werden Ihnen die eingangs gewählte Art der Installation, das gewünschte Verzeichnis sowie die zu installierenden

Elemente der Software angezeigt. Von hier aus haben Sie noch einmal die Möglichkeit, die Installation abubrechen oder Änderungen an der Konfiguration vorzunehmen. Dazu klicken Sie einfach auf den Button *Zurück*.

Ein Klick auf die Schaltfläche *Weiter* startet die Installation der benötigten Dateien.



Abbildung B.26: Installation

Während das Programm die erforderlichen Dateien auf Ihre Festplatte kopiert, zeigt eine Balkengrafik den Fortschritt des Installationsprozesses an.

Der Vorgang sollte nur wenige Minuten dauern. Danach ist ein Neustart Ihres Systems erforderlich, um die Installation abzuschließen. Hierzu erscheint das folgende Fenster.



Abbildung B.27: Neustart des Systems

Beenden Sie dazu alle anderen, eventuell noch aktiven Programme und nehmen Sie alle Datenträger aus den Laufwerken. Klicken Sie anschließend auf den Button *Fertig stellen*, um das System neu zu starten.

Nachdem Ihr Rechner wieder hochgefahren ist, führt das Installationsprogramm noch einige abschließende Setup-Routinen durch und startet dann den *License Key Administration Wizard*, der in Abbildung B.28 dargestellt ist.

Aktivieren Sie das Optionsfeld *Enter a Temporary or Evaluation License Key* und bestätigen Sie Ihre Auswahl anschließend durch einen Klick auf den Button *Weiter*.

Das nächste Fenster dient der Auswahl des Lizenztyps. Quittieren Sie die Meldung einfach durch Anklicken von *Weiter* (siehe Abbildung B.29).

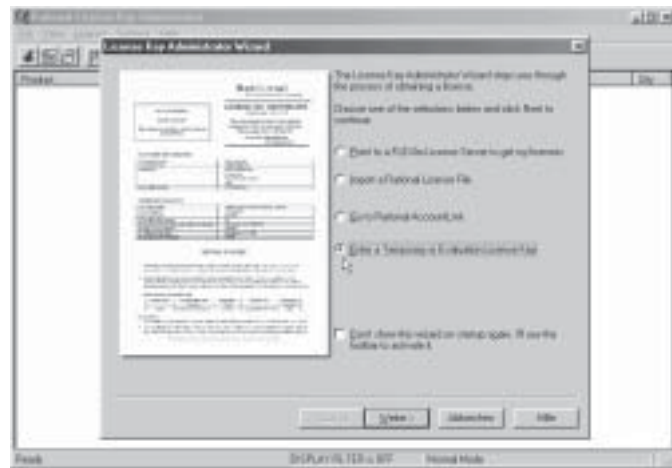


Abbildung B.28: License Key Administration Wizard

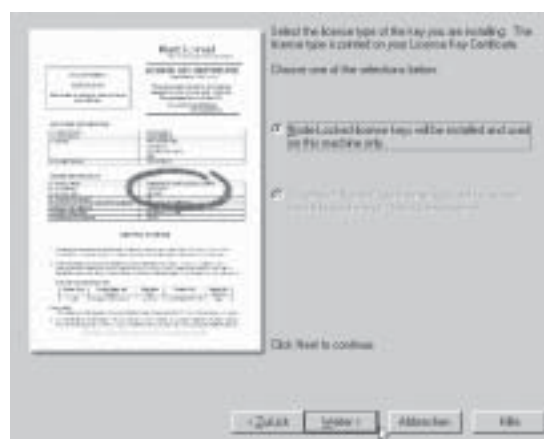


Abbildung B.29: Auswahl Lizenztyp

Nun sind Sie endlich an der Stelle angelangt, an der Sie Ihren Lizenzschlüssel eingeben können. Wählen Sie hierzu zunächst aus dem oberen Listenfeld den Eintrag *Rational Rose Enterprise Edition* aus.

Im zweiten Listenfeld tragen Sie das Datum ein, an dem Ihre Lizenz abläuft. Diese Information erhalten Sie per E-Mail zusammen mit dem Lizenzschlüssel von der Firma Rational.

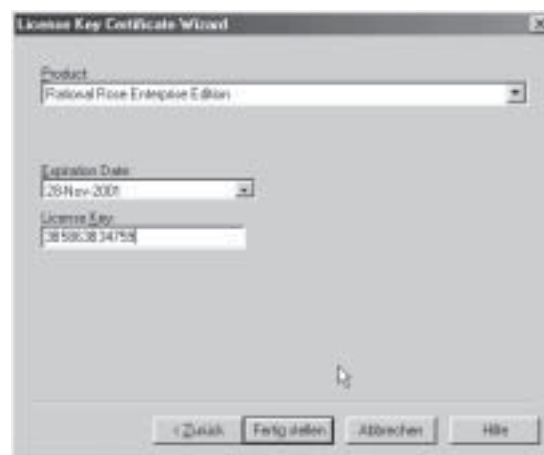


Abbildung B.30: Eingabe des Lizenzschlüssels

Schließlich übernehmen Sie in das darunter liegende Textfeld den Lizenzschlüssel. Am besten kopieren Sie ihn aus der E-Mail von Rational, um Schreibfehler zu vermeiden.

Beenden Sie Ihre Eingaben durch einen Mausklick auf die Schaltfläche *Fertig stellen*. Damit haben Sie die Installationsarbeiten abgeschlossen.

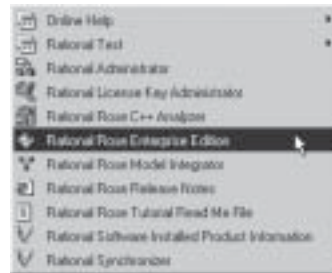


Abbildung B.31: Programmstart

Um das Programm zu starten, wählen Sie im Startmenü aus dem Programmordner zu Rational Rose den Eintrag *Rational Rose Enterprise Edition* aus. Eine kurze Einführung in die Grundfunktionen des Programms erhalten Sie in Teil IV dieses Buchs.

System Architect

Systemanforderungen

Popkin Software benennt ebenfalls Mindestanforderungen an die Systemumgebung des System Architect.

Die empfohlene Ausstattung der Hardware und Software Ihres Systems ist in der folgenden Tabelle zusammengefasst. Weitere Informationen finden Sie auch in der Datei *installationguide.pdf*, die auf der CD-ROM im Verzeichnis *Tools* und dort im Unterverzeichnis *System Architect* abgelegt ist.

Systembestandteile	Mindestanforderungen
Betriebssystem	Windows 95/98 Windows NT 4.0 (aktuelles Service Pack) Windows 2000 Windows ME
Hardware	Pentium-Prozessor mit mindestens 400 MHz Taktfrequenz Mindestens 64 MB RAM (128 MB empfohlen) Festplattenspeicher: 200 MB während der Installation, anschließend 130 MB im normalen Betrieb
Monitor	SVGA-kompatibler Monitor mit einer Auflösung von 800 x 600 Pixel
Internet-Browser	Falls Sie mit den Betriebssystemen Windows NT 4.0 oder Windows 95 arbeiten, benötigen Sie zusätzlich den Microsoft Internet Explorer 4.01 (Service Pack 1) oder höher. Sie erhalten den Internet Explorer unter der Internet-Adresse http://www.microsoft.com/windows/ie .
Dokumentation	Adobe Acrobat Reader 4.x oder höher für die PDF-Dateien der Dokumentation Kostenloser Download des Acrobat Reader unter http://www.adobe.de

Tabelle B.3: Systemanforderungen

Installation

Auf der CD-ROM zu diesem Buch finden Sie eine Evaluationsversion der Software *System Architect* in der Version 8.5. Der Name der Installationsdatei lautet *setup.exe*. Sie finden die Datei im Verzeichnis *Tools* und dort im Unterverzeichnis *System Architect*.

Zusätzlich benötigen Sie noch ein Installations-Passwort und einen Produktcode, die Sie über die Internetseiten der Firma Popkin unter der Adresse <http://www.popkin.com/sales/evals/download-sa-form.htm> anfordern können. Füllen Sie dazu das Formular auf der angegebenen Seite aus. Sie haben danach die Möglichkeit, eine aktuelle Version der Software herunterzuladen (ca. 76 MB). Das Passwort und der Produktcode (in der Regel gültig für 30 Tage) werden Ihnen über die im Formular angegebene E-Mail-Adresse zugeschickt.

Bevor Sie die Software installieren, sollten Sie alle anderen Anwendungen schließen. Nach der Installation müssen Sie Ihr Betriebssystem zunächst neu starten, bevor Sie mit der Software arbeiten können.

Um mit der Installation zu beginnen, führen Sie die Datei *setup.exe* aus. Geben Sie das Installations-Passwort ein, das Ihnen per E-Mail zugeschickt wurde, wenn das Programm Sie dazu auffordert.



Abbildung B.32: Start des Installations-Assistenten



Achten Sie auf die korrekte Schreibweise, wenn Sie das Passwort eintragen. Das Installationsprogramm unterscheidet zwischen Groß- und Kleinschreibung.

Es werden einige Dateien extrahiert und anschließend automatisch der Installations-Assistent gestartet. Klicken Sie nun auf die Schaltfläche *Next*, um zum Dialogfenster *Customer Information* zu gelangen.

Abbildung B.33: Dialogfenster *Customer Information*

Tragen Sie dort die gewünschten Daten ein (Name und Firma) und geben Sie in der dritten Zeile den Produktcode ein, den Sie zusammen mit dem Installations-Passwort per E-Mail erhalten haben.



Achten Sie auf die korrekte Schreibweise des Product Key. Das Installationsprogramm unterscheidet auch hier zwischen Groß- und Kleinschreibung.

Fahren Sie mit einem weiteren Mausklick auf den Button *Next* fort.



Abbildung B.34: Lizenzvereinbarung

Bestätigen Sie die Lizenzvereinbarung mit *Yes*. Falls Sie auf den Button *No* klicken, wird das Installationsprogramm abgebrochen.

Anschließend werden Sie nach dem Zielverzeichnis für die Programmdateien gefragt. Übernehmen Sie die Voreinstellung oder wählen Sie über die Funktion *Browse* ein anderes Verzeichnis aus.

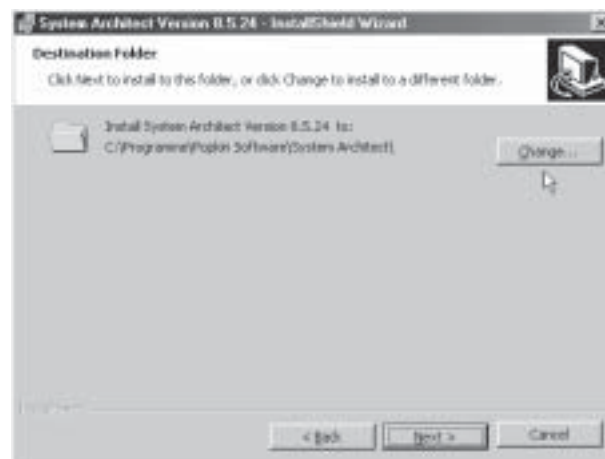


Abbildung B.35: Auswahl des Zielverzeichnis

Per Mausklick auf *Next* gelangen Sie zum nächsten Schritt der Installation des System Architect. Das Programm fordert Sie nun auf, eine Voreinstellung der so genannten Frameworks vorzunehmen, in deren Umgebung Sie später mit dem Programm arbeiten wollen (siehe Abbildung B.36).

Wir schlagen vor, dass Sie an dieser Stelle erst einmal die Option *None* auswählen und die Frameworks erst später festlegen, wenn Sie sich ein wenig mit der Software beschäftigt haben. Sie können aber auch eines der drei angegebenen Frameworks einstellen und diese Auswahl jederzeit wieder rückgängig machen.

Klicken Sie anschließend auf die Schaltfläche *Next*, um im folgenden Dialogfenster weitere Einstellungen für die Installation vorzunehmen (siehe Abbildung B.37).

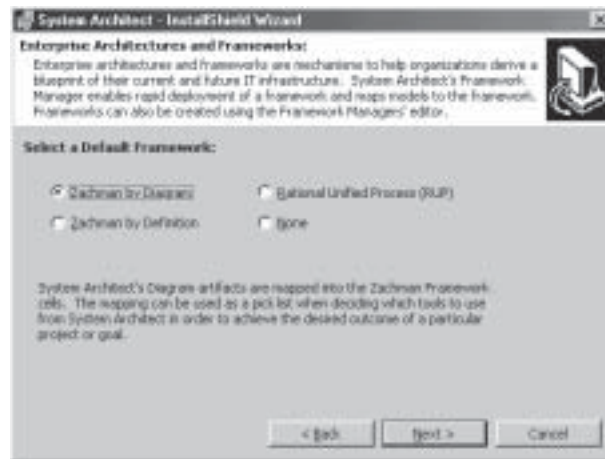


Abbildung B.36: Installation der Frameworks



Abbildung B.37: Art der Installation

Hier können Sie den Umfang der zu installierenden Programmfunktionen bestimmen. In der Option *Complete* werden alle Programmfunktionen installiert. Wenn Sie sich für die Option *Custom* entscheiden, öffnet sich ein Dialogfenster, in dem Sie die zu installierenden Programmmodule individuell bestimmen können.

Damit haben Sie alle benötigten Eingaben vorgenommen. Das Setup-Programm installiert nun die Software auf Ihrem Rechner.

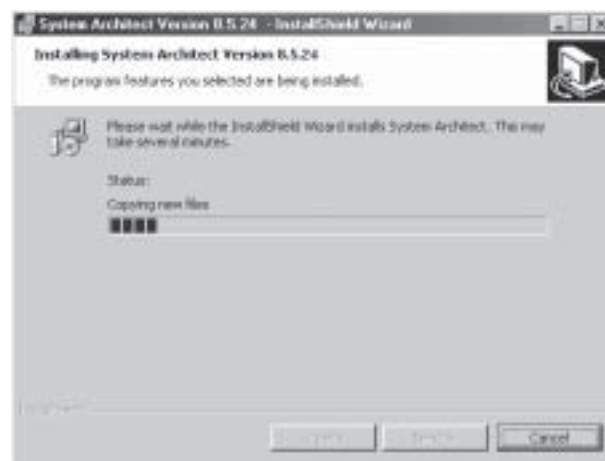


Abbildung B.38: Installation der Programmdateien

Sobald die Installation beendet ist, meldet sich der Assistent und bestätigt die erfolgreiche Installation (siehe Abbildung B.39).

Mit einem Mausklick auf die Schaltfläche *Finish* beenden Sie die Installation.

Es erscheint ein allerletztes Dialogfenster, in dem Sie die Möglichkeit haben, das System sofort oder erst später neu zu starten. Klicken Sie die Alternative an, die Sie bevorzugen (siehe Abbildung B.40).

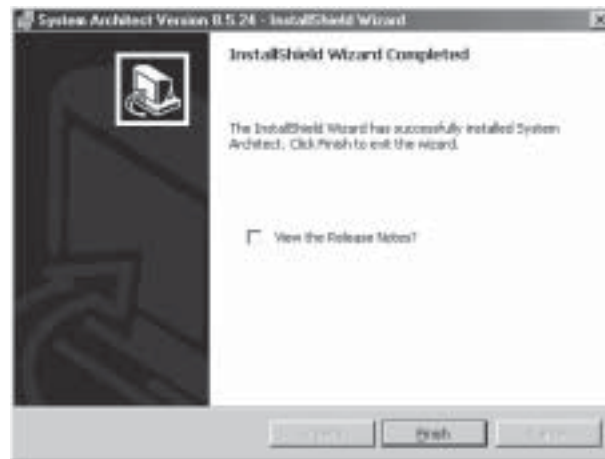


Abbildung B.39: Installation erfolgreich beendet



Abbildung B.40: Neustart des Systems

Um das Programm zu starten, wählen Sie im Startmenü aus dem Programmordner zu System Architect (die standardmäßige Benennung lautet *Popkin Software*) den Eintrag *System Architect* aus.

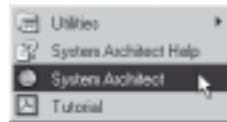


Abbildung B.41: Programmstart

Eine kurze Einführung in die Grundfunktionen des Programms erhalten Sie in Teil IV dieses Buchs.

MagicDraw

Im Internet finden Sie die Firma nomagic, die das UML-Tool MagicDraw unter <http://www.nomagic.com> anbietet und auch eine Demoversion der aktuellen Softwareversion zum Download zur Verfügung stellt. Die Webseite zum Tool MagicDraw ist unter <http://www.magicdraw.com> auch direkt erreichbar.



Die MagicDraw-Demoversion ist beschränkt auf das Speichern von 20 Klassen sowie 25 Zuständen, Use Cases, Aktivitäten und Objekten.

Wir beschreiben im Folgenden die Windows-Version von *MagicDraw UML 5.1*.



Da die Softwarehersteller ihre Produkte regelmäßig aktualisieren, kann es sein, dass Sie bereits eine aktuellere Demoversion von MagicDraw im Web finden.

Dennoch werden Ihnen die im Folgenden aufgeführten Systemanforderungen sowie die Installationsanleitung weiterhelfen.

Systemanforderungen

Der Hersteller legt die folgenden Systemanforderungen fest.

	Minimum	Sinnvoll
CPU	Pentium II 300	Pentium 4 1600
RAM	128 MB	256 MB 512 Mb-1 GB für große Projekte
Festplattenplatz	60 MB	60 MB oder mehr
Auflösung	800*600 mit 64k Farben	1280*1024 mit 64k Farben
Betriebssystem	Alle mit JVM 1.3	Windows 2000 oder Linux
HTML-Browser	Any	Netscape Navigator oder IE
Java Virtual Machine (JVM)	JDK 1.3	JDK 1.3 (für Windows) Blackdown JDK 1.3 (für Linux)

Tabelle B.4: Systemanforderungen

Sie können alternativ zur angegebenen *Java 2 Platform SDK 1.3* auch die *Java 2 Platform SDK 1.4* verwenden. Folgen Sie bei der Installation den Hinweisen im nächsten Abschnitt.

Installation

Starten Sie das in dem angegeben Verzeichnis abgelegte Installationsprogramm *MD_UML_51.exe*.



Abbildung B.42: Vorbereitung der Installation

Warten Sie einen Augenblick, bis die Vorbereitungen zur Installation getroffen sind. MagicDraw ist in verschiedenen Sprachen verfügbar. Voreingestellt ist immer die Sprache des Windows-Betriebssystems, auf dem das Tool installiert wird.

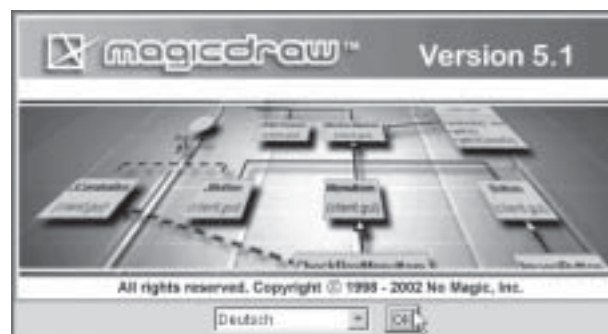


Abbildung B.43: Sprache wählen

Sollten Sie eine andere Sprache als die in der Voreinstellung vorgeschlagene Sprache bevorzugen, können Sie die Auswahl ändern. Später haben Sie über die Programm-Optionen jederzeit die Möglichkeit, die Sprache zu wechseln. Beachten Sie, dass in jeder Sprachversion die System-Hilfe nur in Englisch verfügbar ist.

Bestätigen Sie Ihre Wahl mit einem Klick auf den **OK**-Button.



Abbildung B.44: Einführung zur Installation

Beenden Sie alle Programme und Dienste, die die Performance des Systems bremsen, damit die Installation zügig voranschreiten kann.

Über die *Weiter*-Schaltfläche gelangen Sie zum nächsten Dialog (siehe Abbildung B.45).

Legen Sie in diesem Dialog fest, in welchem Ordner MagicDraw installiert wird. Übernehmen Sie, wenn auf Laufwerk C: genügend Platz vorhanden ist, die Voreinstellung *C:\Programme\MagicDraw UML 5.1*. Ein anderes Verzeichnis können Sie direkt in das Eingabefeld eingeben oder über den *Auswählen...*-Dialog bestimmen.



Abbildung B.45: Installationsordner auswählen

Klicken Sie auf die Schaltfläche *Weiter*, um den folgenden Dialog zu aktivieren (siehe Abbildung B.46).

MagicDraw ist ein reinrassiges Java-Programm und benötigt als solches eine JVM (Java Virtual Machine). Übernehmen Sie die Voreinstellung *Java-VM speziell für diese Anwendung installieren*. So laufen Sie nicht Gefahr, eine inkompatible oder korrupte Version der JVM zu verwenden.

Verlassen Sie auch diesen Bildschirm mit *Weiter*. Der folgende Dialog wird Ihnen wohl ein leichtes Schmunzeln entlocken (siehe Abbildung B.47).



Abbildung B.46: Virtuelle Maschine wählen

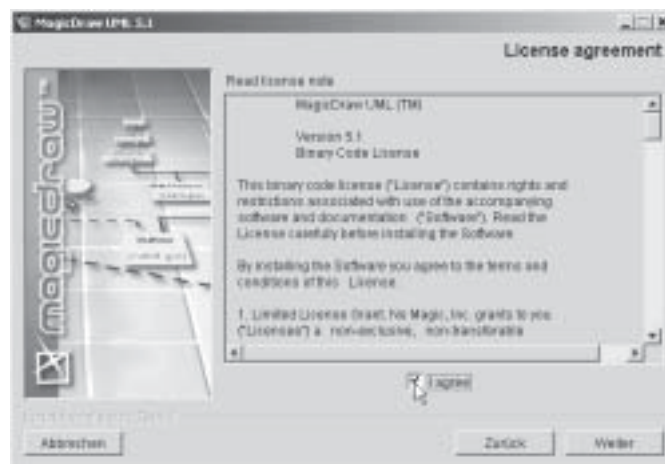


Abbildung B.47: Lizenztext zustimmen

Stimmen Sie mit einem Häkchen zu, dass Sie die Lizenzvereinbarung akzeptieren und schreiten Sie mit *Weiter* zum nächsten Dialog. Dort erwartet Sie die nächste Bestätigung. Haben Sie die Lizenzvereinbarung gelesen und akzeptieren Sie die Bedingungen?

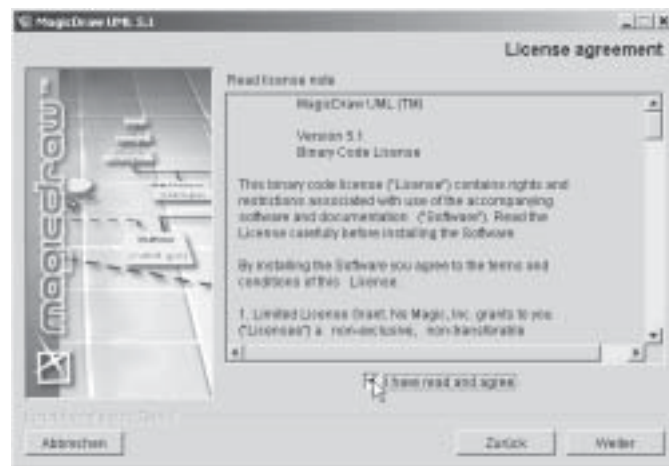
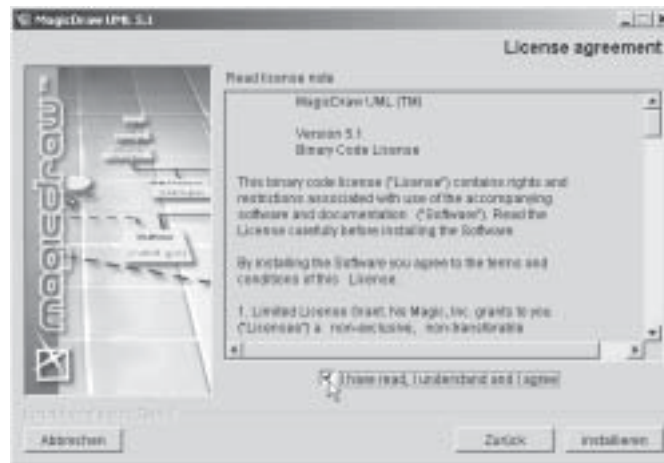


Abbildung B.48: Lizenztext lesen und zustimmen

Dann setzen Sie Ihr Häkchen und hoffen auf den nächsten weiterführenden Dialog, indem Sie auf *Weiter* klicken.

Sie werden noch einmal enttäuscht. Erst wenn Sie hier bestätigen, dass Sie die Lizenzvereinbarung gelesen, verstanden haben und akzeptieren, gelangen Sie mit *Weiter* zum Abschluss der Voreinstellungen der Installation (siehe Abbildung B.49).

Sind alle Einstellungen korrekt, starten Sie die Installation über die Schaltfläche *Installieren* (siehe Abbildung B.50).

**Abbildung B.49:** Lizenztext lesen, verstehen und zustimmen**Abbildung B.50:** Installation läuft

Warten Sie, bis die Installation vollständig erfolgt ist und der abschließende Dialog erscheint.



Abbildung B.51: Installation abgeschlossen

Mit einem Klick auf den *Fertig*-Button wird die Installation abgeschlossen. Starten Sie über das Start-Menü die Anwendung mit *Start-MagicDraw UML 5.1-MagicDraw UML 5.1*.

Ist die Installation korrekt verlaufen, geben Sie in dem folgenden Dialog Ihren Benutzernamen und den Lizenzschlüssel ein.



Abbildung B.52: Lizenzschlüssel eingeben

Kopieren Sie den Benutzernamen und Lizenzschlüssel direkt aus der E-Mail, die Sie von nomagic nach der Registrierung und dem Download erhalten haben, in die Eingabefelder.

Bestätigen Sie die Eingaben mit *OK*.

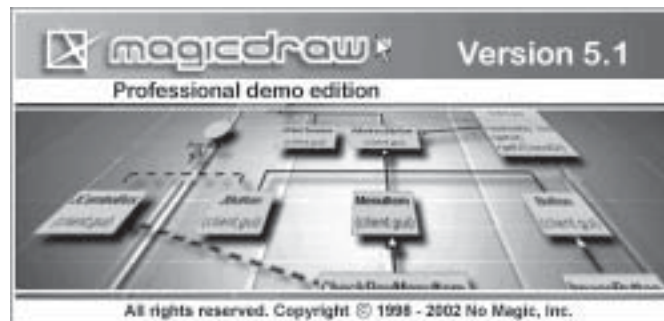


Abbildung B.53: Startbildschirm

MagicDraw zeigt während der Ladezeit das oben dargestellte Eingangsbild an, bis das MagicDraw-Hauptfenster erscheint (siehe Abbildung B.54).

Bevor Sie mit der Anwendung arbeiten können, müssen Sie der *Binary Code License* durch einen Klick auf die Schaltfläche *Ich stimme zu* beipflichten.

Teil IV enthält eine kurze Einführung in MagicDraw.



Abbildung B.54: Eingangsbildschirm

Poseidon

Poseidon von der Gentleware AG ist das einzige Tool auf der CD-ROM zum Buch, das wir Ihnen in der Community Edition als Vollversion zur Verfügung stellen können. Sie finden die Programmdatei der Windows-Version *PoseidonCE1_2_1Installer.exe* im Verzeichnis *Tools/Poseidon*.

Über die Webseite <http://gentleware.de> können Versionen für alle Betriebssysteme, für die eine JVM (Java Virtual Machine) existiert, abgerufen werden.

Systemanforderungen

Als Installationsvoraussetzungen führt der Hersteller lediglich die Existenz einer JVM der Java 2 Platform SDK 1.3 an.

Installation

Starten Sie die Installationsroutine, indem Sie das Programm *PoseidonCE1_2_1Installer.exe* im Verzeichnis *Tools/Poseidon* von der Buch-CD starten.

Der Installer bereitet anschließend die Installation vor.



Abbildung B.55: Vorbereitung der Installation

Nach einer kurzen Wartezeit erscheint der folgende Zwischenbildschirm.



Abbildung B.56: Abschluss der Installationsvorbereitung

Die Anzeige des Zwischenbildschirms wird unterbrochen von einer Systemmeldung des Installers, die Sie mit dem *OK*-Knopf bestätigen.



Abbildung B.57: Installer bestätigen

Anschließend meldet sich der Einführungsdialog zur Vereinbarung der Installations-Optionen. Klicken Sie auf die Schaltfläche *Next*, um zum nächsten Dialog zu gelangen (siehe Abbildung B.58).

Hier legen Sie nun das Verzeichnis fest, in dem das Programm installiert wird (siehe Abbildung B.59).

Übernehmen Sie die von der Installationsroutine vorgeschlagene Pfadangabe *C:\Programme\PoseidonForUMLCE*. Sollte das Laufwerk C: nicht genügend freien Speicher besitzen, können Sie auch einen beliebigen anderen Pfad angeben. Mit *Next* gelangen Sie in den Dialog zur Festlegung einer JVM (Java Virtual Machine), die für das in Java geschriebene Tool notwendige Voraussetzung ist.

Sollte auf Ihrem System keine JVM installiert sein, so installieren Sie zuerst eines der Tools, die eine JVM enthalten (Together oder MagicDraw). Am Beispiel von Together zeigen wir, wie Sie die dort installierte JVM auch für Poseidon nutzen können.



Abbildung B.58: Installation Einführung

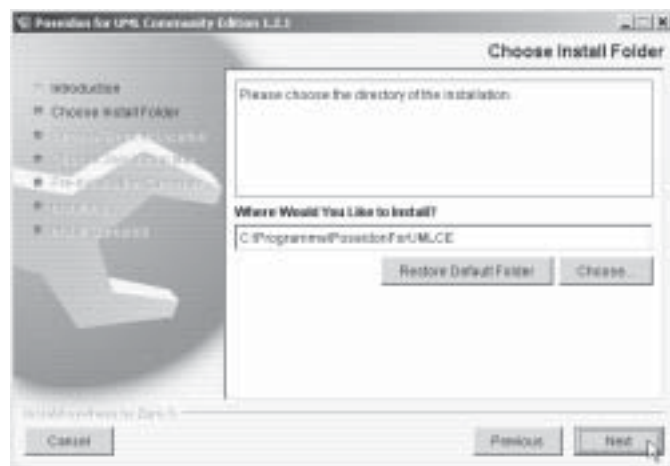


Abbildung B.59: Wahl des Installationsverzeichnis

Wählen Sie die Schaltfläche *Choose Another...*, um in dem anschließenden Datei-Dialog die Datei *java.exe* im Verzeichnis *c:\TogetherSoft\Together6.0\jdk\jre\bin* auszuwählen. Bestätigen Sie Ihre Auswahl mit *Öffnen*.

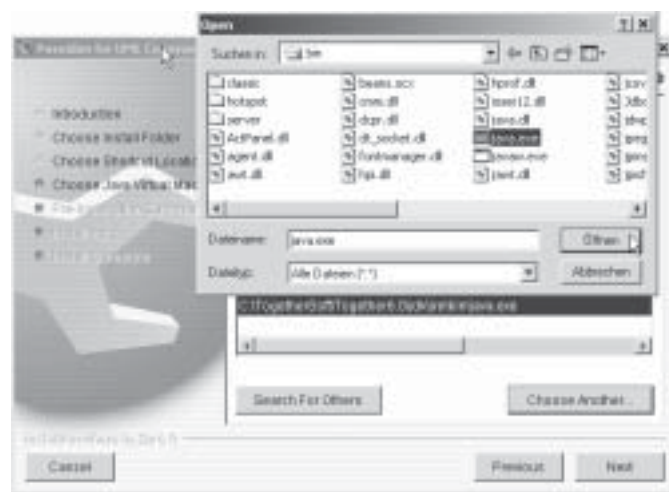


Abbildung B.60: Wahl des JVM

Anschließend wählen Sie die jetzt im Auswahlfenster des Installations-Dialogs erscheinende JVM aus und gehen mit *Next* zur nächsten Auswahl über.

Wählen Sie eine neue Programmgruppe (*In a new Program Group*) mit der vorgegebenen Bezeichnung aus. Setzen Sie ein Häkchen vor die Option *Create Icons for All Users*, damit alle User des Systems einen schnellen Zugriff auf Poseidon erhalten.

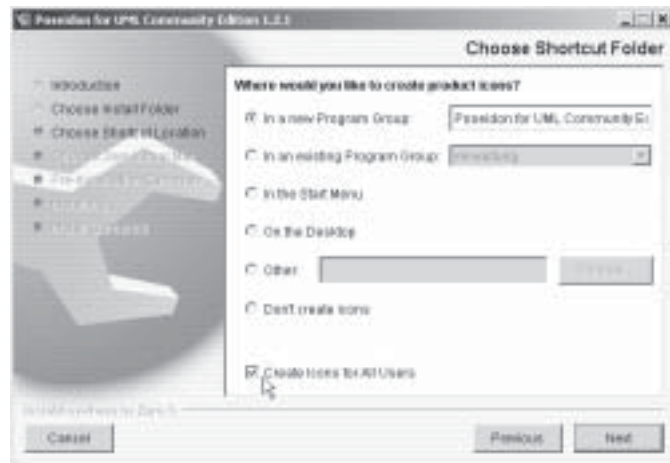


Abbildung B.61: Schnellwahl definieren

Beenden Sie den Dialog mit *Next*.

Die folgende Übersicht zeigt noch einmal alle gewählten Installationsoptionen an (siehe Abbildung B.62).

Wählen Sie *Install*, um den Installationsprozess zu starten. Der Fortschrittsbalken im unteren Teil zeigt den Fortgang der Installation an (siehe Abbildung B.63).

Bestätigen Sie das Ende der Installation, indem Sie auf die Schaltfläche *Done* klicken (siehe Abbildung B.64).

Damit ist Poseidon vollständig installiert und kann über das Start-Menü von Windows aufgerufen werden. Wählen Sie dazu *Start-Programme-Poseidon for UML Community Edition 1.2.1-Poseidon for UML CE*.

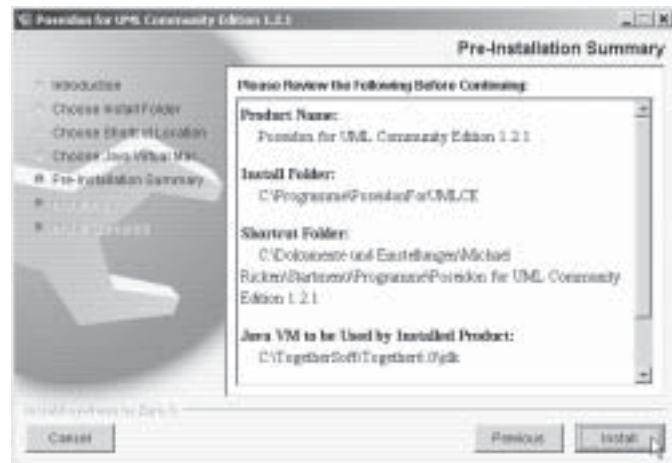


Abbildung B.62: Installationsparameter



Abbildung B.63: Installationsprozedur



Abbildung B.64: Installation beenden

Die Entwicklungsumgebung startet mit einem Dialog zur Registrierung der Software. Starten Sie das Programm vor dem 31.12.2004, können Sie das Fenster ohne Registrierung schließen. Nach diesem Datum müssen Sie die Registrierung vornehmen (siehe Abbildung B.65).

Starten Sie Poseidon über die Schaltfläche *Poseidon starten* (siehe Abbildung B.66).

Nach einer Wartezeit, die durch die Anzeige des folgenden Bildes überbrückt wird, startet das Programm (siehe Abbildung B.67).

Sehen Sie den folgenden Hauptbildschirm, ist die Installation glücklich und alles korrekt verlaufen (siehe Abbildung B.68).

Poseidon wird in seinen grundlegenden Strukturen in einer kurzen Einführung in Teil IV vorgestellt.



Abbildung B.65: Final-Key-Dialog



Abbildung B.66: Interim-Key-Dialog

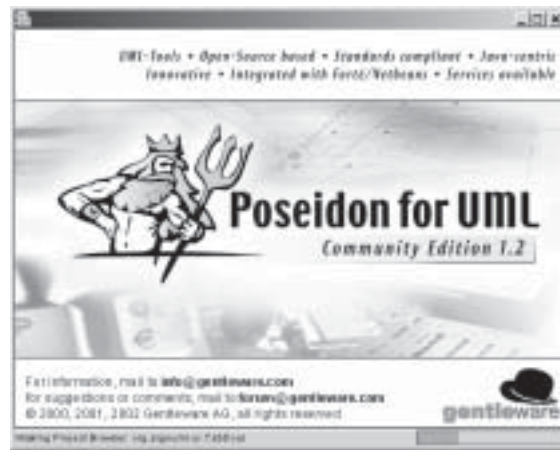


Abbildung B.67: Ladevorgang

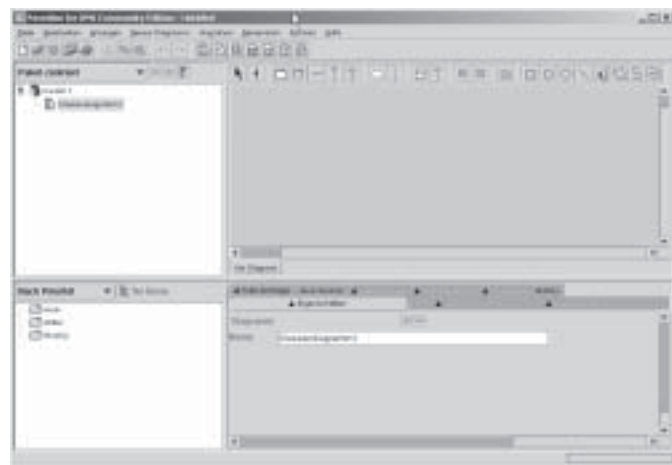
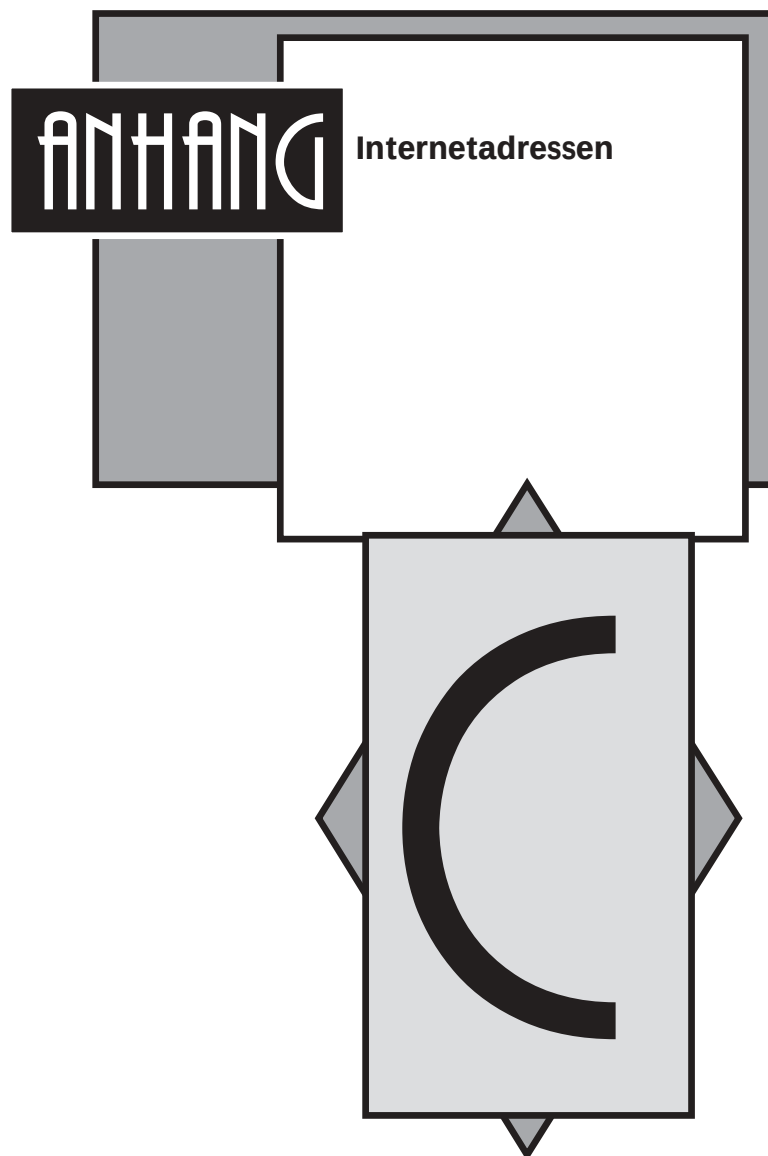


Abbildung B.68: Hauptfenster







Internetadressen

Das Internet ist eine ergiebige Informationsquelle zu Themen rund um die UML. Sie finden dort neben einführenden Online-Tutorials auch zahlreiche Beispiele für die einzelnen Diagrammartentypen. Interessant sind auch viele Aufsätze, die sich kritisch mit der UML auseinandersetzen. Über die Internetseiten der Object Management Group können Sie zudem die Fortschritte bei der Weiterentwicklung der UML verfolgen.

Die folgende Auswahl beschränkt sich auf Adressen, die entweder nach unserer subjektiven Meinung sehr informativ sind oder die sich als Ausgangspunkt für weitere Recherchen in besonderem Maße eignen. Da Internetangebote oft recht kurze Lebenszyklen haben, enthält die folgende Auflistung weitgehend Adressen, die schon länger existieren und bei denen wir davon ausgehen, dass sie auch dauerhaft verfügbar sind.

<http://uml-buch.mindx.de>

Die Webseite zum UML-Taschenbuch. Hier erhalten Sie aktuelle Informationen über neue Versionen der Tools, die sich auf der CD-Rom befinden und weitere nützliche Informationen zur UML.

<http://www.csharp-info.de>

Möchten Sie mehr über die Sprache C# erfahren, die wir in Teil III zur Umsetzung von UML-Modellen in ein objektorientiertes Programm eingesetzt haben, können wir Ihnen unser Einsteigerseminar C# empfehlen. Die Webseite zum Buch zeigt alle Listings und weitere Informationen für Einsteiger zur Sprache C#.

<http://www.objectmind.de>

Die Webseite von Michael Ricken informiert über Workshops, Coaching und Beratung rund um die objektorientierte Systementwicklung.

lung mit UML, C# und Java sowie die objektorientierte Gestaltung von Geschäftsprozessen und Organisationsstrukturen.

<http://www.mindx.de>

Studenten, die an Universitäten, Fachhochschulen und Akademien Michael Rickens Vorlesungen und Praktika besuchen, finden hier Informationen zu den aktuellen Veranstaltungen. Die Seite enthält neben Seminar-Informationen regelmäßig auch virtuelle Ausstellungen von Programmen, Ideen und Konzepten, die Studenten in den Veranstaltungen von Michael Ricken entwickelt haben.

<http://www.stratim.de>

Thomas Erler stellt hier seine Dienstleistungen rund um die Objektorientierte Systementwicklung dar. Darüber hinaus finden Sie auf diesen Seiten Informationen und Links zu den Themen Objektorientierung, UML, Java und C#.

<http://www.oose.de>

Auf den Seiten des Autors und Beraters Bernd Österreich, einem der bekanntesten Fachleute für die objektorientierte Analyse mit UML im deutschsprachigen Raum, finden Sie Glossare, Notationsübersichten Aufsätze und mehr.

<http://www.uml.sig.de>

Eines der wenigen Portale, die für den deutschsprachigen Raum wertvolle Links und Informationen zur UML zusammenstellt. Die Informationen sind hervorragend recherchiert und gut strukturiert präsentiert.

<http://www.omg.org/uml/>

Dies sind die UML-Seiten der Object Management Group (OMG). Sie enthalten Informationen zum aktuellen Stand der UML-Weiter-

entwicklung. Hier können Sie die Diskussionen über neue Versionen der UML verfolgen. Außerdem finden Sie einige kurze, einführende Artikel über die Grundlagen der UML. Die Linkliste ist nicht gerade umfangreich. Dafür verweist sie jedoch auf inhaltlich wertvolle Seiten.

http://www.omg.org/technology/documents/modeling_spec_catalog.htm

Wenn Sie sich intensiver mit der UML beschäftigen, sind die UML-Spezifikationen der OMG eine unverzichtbare Informationsquelle. Unter dieser Adresse finden Sie einen Download-Katalog der OMG, der u.a. die aktuelle UML-Spezifikation im PDF-Format enthält.

<http://www.embarcadero.com/products/describe/UMLposter.pdf>

Über diesen direkten Link können Sie ein UML-Poster der Firma Embarcadero herunterladen. Es enthält alle wichtigen UML-Diagramme in einer Übersicht. Die Datei ist etwa 720 KB groß.

http://www.holub.com/class/oo_design/uml.pdf

Ebenfalls eine Übersicht der UML-Diagramme enthält die Referenzkarte, die Sie über diesen direkten Link herunterladen können. Die Karte fasst die Diagramme mit Beispielen auf zwei DIN-A4 Seiten zusammen. Dateigröße: 32 KB.

<http://www.martinfowler.com/>

Martin Fowler ist im Bereich der objektorientierten Analyse schon seit einigen Jahren recht bekannt. Auf seiner Homepage finden Sie zahlreiche Artikel u.a. zu den Themen UML, Analyse und Design sowie Software Engineering.

<http://www.cs.york.ac.uk/puml/>

The precise UML group bringt internationale Forscher und Interessenten aus der Praxis zusammen, um die Weiterentwicklung der UML voranzutreiben. Ihre Mitglieder sind an dem laufenden Standardisierungsprozess beteiligt und veranstalten zahlreiche Workshops und Konferenzen zu diesem Thema. Die Webseite enthält daher insbesondere Informationen über den aktuellen Stand und die Entwicklung des Standardisierungsprozesses. Sie können aber auch zahlreiche Artikel herunterladen und finden Links zu weiteren UML-bezogenen Seiten.

<http://www.kabira.com/bock/index.html>

Hier finden Sie verschiedene Artikel von Conrad Bock zu weitergehenden Einsatzmöglichkeiten der UML. Die Navigation ist zwar etwas beschwerlich, es sind aber dennoch viele weiterführende Informationen und Links zu finden.

<http://www-4.ibm.com/software/ad/standards/ocl.html>

Diese Seite von IBM informiert über die Object Constraint Language (OCL), mit der in der UML Zusicherungen näher definiert werden können.

<http://softdocwiz.com/UML.htm>

Auf dieser Seite ist u.a. ein umfangreiches englischsprachiges Glossar mit Begriffen der UML zu finden.

<http://www.umlconference.org>

Dies ist die Homepage der *Annual International Conference on the Unified Modeling Language*. Sie enthält inhaltliche und organisatorische Informationen über die nächste Tagung. Links verweisen auf die vergangenen UML-Konferenzen.

<http://ivs.cs.uni-magdeburg.de/~dumke/UML/index.htm>

Hier finden Sie ein deutschsprachiges UML-Tutorial der Uni Magdeburg.

<http://www.celigent.com/omg/umlrtf/tutorials.htm?#Tutorial%20overview>

Diese Adresse verweist auf verschiedene englischsprachige Tutorials der OMG und enthält viele weitere Informationen (u.a. Mailingliste, Arbeitsgruppen).

http://pigseye.kennesaw.edu/~dbraun/csis4650/A&D/UML_tutorial/

In einem Tutorial der Kennesaw State University können Sie hier online blättern.

http://www.objectsbydesign.com/tools/umltools_byCompany.html

Diese Seite von *Objects by Design* enthält einen Überblick kommerziell verfügbarer UML-Tools. Die Darstellung ist sehr übersichtlich und enthält neben Informationen zu Preisen, Versionsnummern und benötigter Plattform auch Links zu den Herstellern der Tools.

http://www.objectsbydesign.com/tools/modeling_tools.html

Ebenfalls von *Objects by Design* ist diese Sammlung von Auswahlkriterien für UML-Tools.

<http://www.jeckle.de/umltools.htm>

Die deutschsprachige Seite von Mario Jeckle gibt ebenfalls einen Überblick verfügbarer UML-Tools. Eine Tabelle stellt den Funktionsumfang der Tools sehr übersichtlich gegenüber.

http://www.cetus-links.org/oo_uml.html

Die wohl umfangreichste Linksammlung (nicht nur) zur UML. Auch zu nahezu allen anderen Bereichen der Objektorientierung sollten Sie hier unbedingt einmal vorbeischaun. Die Sammlung enthält auch zahlreiche Verweise auf deutschsprachige Seiten und Bücher.

<http://dmoz.org/Computers/Programming/Methodologies/Object-Oriented/UML/>

Hier finden Sie ein Linkverzeichnis zum Thema UML von Open Directory.

http://www.microgold.com/Stage/UML_FAQ.html

Microgold und Rational beantworten auf dieser Seite einige häufig gestellte Fragen zur UML.

<http://www.gooal.net>

Unter dieser Adresse finden Sie eine deutschsprachige Mailingliste zur Objekttechnologie.

<http://forums.objectsbydesign.com/>

Hier bietet *Objects by Design* eine Liste verschiedener englischsprachiger Foren mit den Themen UML und Objektorientierung an.

<http://cafe.rational.com/HyperNews/get/hn/umlcafe.html>

Dies ist ein englischsprachiges Forum der Firma Rational.

<http://www.celigent.com/uml/>

Celigent bietet hier ein Forum und aktuelle News zum Thema UML an.

<http://groups.yahoo.com/group/uml-forum/>

Auch bei Yahoo gibt es ein UML-Forum.

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

Dies ist die Homepage der Firma Rational Software, die u.a. das Tool Rational Rose zur objektorientierten Modellierung entwickelt hat.

<http://www.nomagic.com>

Das Entwicklungswerkzeug Magic Draw wird von der Firma No Magic hergestellt.

<http://www.popkin.com>

Die Firma Popkin Software ist der Hersteller des Tools System Architect.

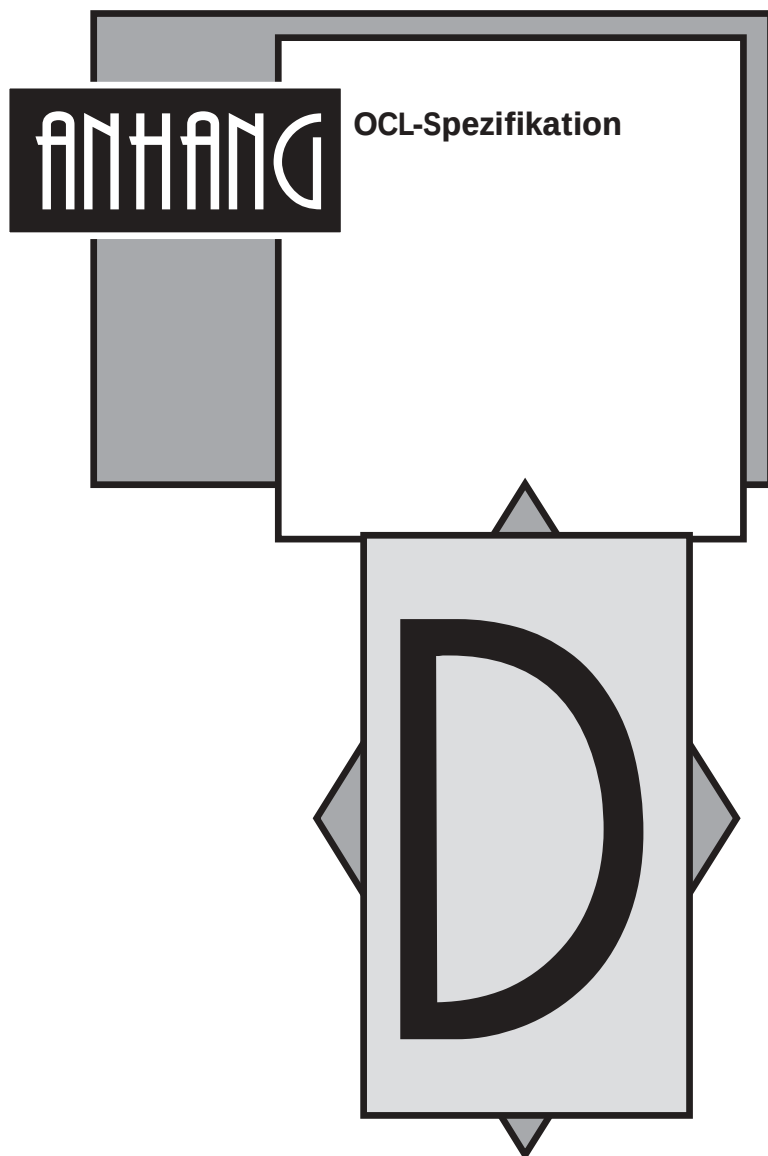
<http://www.togethersoft.de>

TogetherSoft entwickelt das Werkzeug Together, das Sie ebenfalls auf der beiliegenden CD-ROM finden. TogetherSoft bietet die aktuelle Version als Evaluations-Version zum Download an.

<http://www.gentleware.de>

Gentleware ist der deutsche Hersteller des UML-Tools Poseidon, das Sie über diese Adresse in der Community Edition als Freeware herunterladen können. Sie finden das Werkzeug auch auf der CD-ROM.







OCL-Spezifikation

In Teil II dieses Buches wurde die UML-Notation mit ihren Standardelementen vorgestellt. Sie dienen dazu, Modellelemente, Beziehungen und Zusicherungen zu verfeinern und zu verdeutlichen. Häufig ist es mit den Standardelementen der UML allerdings nicht oder nur schwer möglich, eine Problemstellung ausreichend detailliert und zugleich einfach zu beschreiben. In diesen Fällen kann mit der *Object Constraint Language (OCL)* eine formale Sprache eingesetzt werden, die Objektmodelle der UML ergänzt und verfeinert.

OCL als Ergänzung der UML

Mit der OCL können Constraints (Zusicherungen) spezifiziert werden, die durch Stereotypen und Kommentare nicht oder nur sehr umständlich zu beschreiben sind. OCL ist eine formale Sprache, die ursprünglich unter dem Namen *IBEL (Integrated Business Engineering Language)* von der Firma IBM entwickelt wurde, um Geschäftsprozesse zu modellieren.



Die OCL dient ausschließlich der Auswertung von Ausdrücken. Das Ergebnis ist immer ein Wert und führt zu keinerlei Veränderungen im Objektmodell. Es handelt sich nicht um eine Programmiersprache, mit der Prozesse oder Aktivitäten angestoßen werden können.

Außer den Zusicherungen lassen sich mit der OCL Invarianten, Vor- und Nachbedingungen (invariants, pre- und postconditions) für Operationen sowie Navigationspfade für Objektnetze beschreiben. Es handelt sich um eine rein formale Spezifikationssprache, die in Ergänzung zur UML eingesetzt wird. Implementierungsaspekte können mit der OCL nicht dargestellt werden.

Kontext und Invarianten

OCL-Ausdrücke werden in UML-Diagrammen als Text in der Schriftart Courier dargestellt:

```
'Dies ist ein OCL-Ausdruck.'
```

Der *Kontext* eines Ausdrucks wird durch das Schlüsselwort `context` angegeben und leitet jeden OCL-Ausdruck ein. Die UML-Stereotypen «invariant», «precondition» und «postcondition» werden in der OCL durch die Schlüsselwörter `inv`, `pre` und `post` repräsentiert. Nach einem Doppelpunkt steht dann der eigentliche OCL-Ausdruck. Ein OCL-Ausdruck hat also die Form:

```
context TypName inv:  
'Dies ist ein OCL-Ausdruck mit dem Stereotyp «invariant» im  
Kontext von TypName' = 'String'
```

Invarianten sind Zusicherungen, die über einen bestimmten Zeitraum (in der Regel für die Lebensdauer einer Instanz) Gültigkeit besitzen. Sie werden durch das Schlüsselwort `inv` gekennzeichnet. Zur Hervorhebung können die Schlüsselwörter **fett gedruckt** werden. Nachfolgend einige Beispiele aus dem Kontext einer Unternehmung.

Einfache Zusicherungen beziehen sich auf die Werte von Attributen. Der Ausdruck

```
context Unternehmung inv:  
anzahlMitarbeiter > 100
```

definiert die Zusicherung, dass die beschriebene Unternehmung stets mehr als 100 Mitarbeiter hat. Diese Zusicherung bezieht sich auf Instanzen des Typs `Unternehmung`. Um dies zu verdeutlichen, wird das Schlüsselwort `self` hinzugefügt. Der OCL-Ausdruck lautet dann (mit derselben Bedeutung):

```
context Unternehmung inv:  
self.anzahlMitarbeiter > 100
```

`self` bezeichnet hier die Instanz, auf die sich der Ausdruck bezieht. In Fällen wie diesem Beispiel ist der Bezug eindeutig, so dass das Schlüsselwort `self` hier auch entfallen kann. Eine alternative Schreibweise bei Verzicht auf `self` ist die Einführung einer Konstanten als Name für den Kontext:

```
context u : Unternehmung inv:  
u.anzahlMitarbeiter > 100
```

`u` steht in dieser Variante als Platzhalter für den Kontext Unternehmung.

Vor- und Nachbedingungen

Vorbedingungen (preconditions) sind Bedingungen, die beim Aufruf einer Methode gelten müssen. *Nachbedingungen* (postconditions) bleiben auch nach dem Aufruf einer Methode bestehen. Auch diese beiden Standardelemente können Teil eines OCL-Ausdrucks sein.

Die Schlüsselwörter `pre` und `post` werden dabei vor die jeweilige Bedingung gesetzt. Die Syntax eines Ausdrucks mit Vor- bzw. Nachbedingung und einer Operation sieht etwa wie folgt aus:

```
context TypName::operationName (param1:Typ1, ...):RückgabeTyp  
pre: param1 > 7  
post: result = param1 + 5
```

`TypName` bezeichnet wiederum den Kontext, `operationName` ist die Bezeichnung der Operation und `RückgabeTyp` der Datentyp, den die Auswertung des Ausdrucks annimmt. Die an die Operation übergebenen Parameter besitzen jeweils einen bestimmten Datentyp und können im Rahmen der Auswertung des Ausdrucks verarbeitet werden. `result` bezeichnet das Ergebnis der Auswertung und enthält den Wert, den der Ausdruck ermittelt hat und zurückliefert. In dem Beispiel lautet die Vorbedingung, dass der Parameter `param1` beim Aufruf des Ausdrucks einen Wert größer als 7 haben muss. Die Nachbedingung verlangt als Ergebnis des Ausdrucks (`result`) die Summe von `param1` und 5.

Eine Nachbedingung kann also beispielsweise lauten:

```
context Mitarbeiter::sollArbeitszeit():Integer
post: result = 35
```

Die Instanz `Mitarbeiter` ist Kontext dieses Ausdrucks. Mit dem Ausdruck ist eine Operation verbunden, die den Namen `sollArbeitszeit` trägt und vom Typ `Integer` (ganzzahlig) ist. Die Nachbedingung legt als Ergebnis des Ausdrucks den Wert 35 (`result = 35`) fest. Statt der Schlüsselwörter `pre` und `post` kann den Bedingungen auch ein Name zugewiesen werden.

Basistypen und -operationen

Die OCL besitzt einige *Basistypen*, die bereits vordefiniert und somit jederzeit einsetzbar sind. Mit `Integer` haben Sie schon einen dieser Basistypen kennen gelernt. Die folgende Tabelle enthält eine Übersicht der verfügbaren Basistypen.

Typ	Werte	Beispiel
Integer	Positive und negative ganze Zahlen	-12, 0, 5, 9580
Real	Reelle Zahlen	-18522, -12.5, 0, 335.0987
String	Zeichenketten	'Objektorientierung'
Boolean	Wahr oder falsch (0 oder 1)	True, false
Set	Mengen ohne doppelte Werte	{5, 27, 133}, {-0.5, 1.3, 4.0}, {'eins', 'zwei', 'drei'}
Bag	Mengen; doppelte Werte erlaubt	{5, 33, 1}, {5, 1, 5, 33, 1}
Sequence	Wie Bag, jedoch geordnet	{1, 5, 33}, {1, 1, 5, 5, 33,}

Tabelle D.1: Vordefinierte Basistypen

Integer, Real, String und Boolean sind auch in Programmiersprachen häufig verwendete Datentypen. Oft werden diese Typen nach ihren Wertebereichen weiter differenziert (z.B. Short Integer, Long Integer). Auch die Verarbeitung von Mengen ist in vielen Programmiersprachen gängig. Die OCL unterscheidet drei verschiedene Arten von Mengen, deren Operationen jedoch weitgehend gleich sind, wie Sie später noch erfahren werden. Im Unterschied zum Datentyp *Set* erlauben *Bag* und *Sequence* das mehrfache Auftreten eines Elements in einer Menge. Mengen vom Typ *Sequence* sind darüber hinaus auch noch geordnet.

Operationen der Basistypen Integer, Real, String und Boolean

Neben Datentypen definiert die OCL auch Operationen, die mit den Basistypen durchführbar sind. Die folgende Tabelle führt die wichtigsten Operationen für die Typen Integer, Real, String und Boolean auf.

Typ	Operationen
Integer	+, -, *, /, =, div, mod, abs
Real	+, -, *, /, =, abs, round
String	+, -, =, toUpper, toLower, substring, size
Boolean	=, and, or, xor, not

Tabelle D.2: Operationen der Basistypen *Integer*, *Real*, *String* und *Boolean*

Der Tabelle ist zu entnehmen, dass es sich bei den Operationen im Wesentlichen um diejenigen handelt, die Ihnen von den meisten Programmiersprachen her bekannt sind.

- ✓ `div` und `mod` liefern die Ergebnisse der ganzzahligen Division.
- ✓ `abs` berechnet den absoluten Wert einer Zahl.
- ✓ `round` rundet einen Wert vom Typ `Real` auf den nächsten ganzzahligen Wert.
- ✓ `toUpper` und `toLower` wandeln eine Zeichenkette in Groß- bzw. Kleinbuchstaben um.
- ✓ `substring` liefert einen bestimmten Ausschnitt aus einer Zeichenkette.
- ✓ `size` ermittelt die Länge einer Zeichenkette. In OCL-Ausdrücken wird die Operation `size` mit einem Punkt an den Namen der Zeichenkette angehängt, z.B.: `result = s.size`.
- ✓ Der Ausdruck `A and B` liefert das Ergebnis `true`, wenn sowohl A als auch B wahr sind.
- ✓ `A or B` ergibt `true`, wenn A, B oder beide wahr sind.
- ✓ `A xor B` ist `true`, wenn entweder A oder B wahr sind (exklusives oder).
- ✓ `not A` ist die Umkehrung von A. Ist A falsch, dann lautet das Ergebnis von `not A`: `true`.

Achten Sie bei der Verwendung der Operationen auf die korrekte Kombination der Datentypen. Gültige Ausdrücke sind `3.54 + 17 + 4.083` und `'Tisch' + 'Bein'`. Das Ergebnis des ersten Ausdrucks hat den Datentyp `Real` (Ergebnis: 24.623), das zweite Ergebnis ist vom Typ `String` (Ergebnis: 'TischBein'). Nicht zulässig ist hingegen ein Ausdruck wie beispielsweise `7 * true`, da der Typ `Integer` nicht mit dem Typ `Boolean` verarbeitet werden kann bzw. die Datentypen nicht zu der Operation passen.

Operationen der Basistypen `Set`, `Bag` und `Sequence`

`Set`, `Bag` und `Sequence` sind Unterklassen der Klasse `Collection`. Sie besitzen daher einige gemeinsame Operationen und erweitern die

Definition von Collection um zusätzliche eigene Methoden. Es handelt sich bei allen drei Basistypen um Mengen. Eine Auswahl der Operationen für die Basistypen Set, Bag und Sequence zeigt die folgende Tabelle.

Typ	Operationen
Set	size, includes, excludes, count, isEmpty, notEmpty, sum, exists, forAll, intersection, select, reject, collect, asSequence, asBag
Bag	size, includes, excludes, count, isEmpty, notEmpty, sum, exists, forAll, intersection, select, reject, collect, asSequence, asSet
Sequence	size, includes, excludes, count, isEmpty, notEmpty, sum, exists, forAll, append, prepend, subsequence, at, first, last, select, reject, collect, iterate, asBag, asSet

Tabelle D.3: Operationen der Basistypen *Set*, *Bag* und *Sequence*

Auch hierzu einige Beispiele zur Verdeutlichung der Operationen, die durch einen Pfeil mit dem Namen der zu verarbeitenden Menge verbunden werden:

- ✓ `size` bestimmt die Anzahl der Elemente in einer Menge. Der entsprechende OCL-Ausdruck der Bedingung, dass eine Unternehmung mehr als 50 Mitarbeiter hat, lautet z.B.:
`self.mitarbeiter -> size > 50`
`mitarbeiter` ist in diesem Beispiel eine Menge vom Typ *Set*, *Bag* oder *Sequence*, die als Elemente die Daten der einzelnen Mitarbeiter enthält.
- ✓ `includes` ist ein Ausdruck mit dem Ergebnistyp *Boolean*. Der Ausdruck liefert den Wert `true`, wenn das genannte Objekt in der angegebenen Menge enthalten ist: `self.mitarbeiter -> includes (Peter)`

- ✓ `excludes` prüft analog zu `includes`, ob ein Objekt *nicht* in der betrachteten Menge enthalten ist. Der Ausdruck hat somit den Wert `false`, wenn das Objekt Teil der Menge ist. Es gilt:
`excludes (x) = not (includes (x))`.
- ✓ `count` zählt, wie oft ein Objekt in einer Menge enthalten ist.
- ✓ Mit der Operation `isEmpty` wird überprüft, ob eine leere Menge vorliegt. `self.kunden -> isEmpty`.
- ✓ Auch hier existiert mit `notEmpty` wieder eine Umkehrung der Operation.
- ✓ `sum` addiert sämtliche Elemente einer Collection (Set, Bag oder Sequence). Voraussetzung ist, dass die einzelnen Elemente einen Datentyp besitzen, der die Addition unterstützt.
- ✓ Die Operation `exists` prüft, ob mindestens ein Element der betrachteten Menge eine bestimmte Bedingung erfüllt. Das Ergebnis ist vom Typ `Boolean`. Mit dieser Operation lässt sich also nicht feststellen, wie viele Elemente die Bedingung erfüllen. Es ist nur eine Aussage darüber möglich, ob es überhaupt (mindestens) ein solches Element gibt. Der Ausdruck für die Abfrage, ob einer der Mitarbeiter mehr als 100.000 € verdient, lautet:
`self.mitarbeiter -> exists (gehalt > 100000)`.
- ✓ `forAll` testet, ob eine Bedingung für *alle* Elemente der Menge gilt. Auch hier ist das Ergebnis wieder `true` oder `false`. Der OCL-Ausdruck zur Überprüfung, ob alle Mitarbeiter mehr als 40.000 € verdienen, heißt:
`self.mitarbeiter -> forAll (gehalt > 40000)`.
- ✓ Die Operation `intersection` ermittelt die Schnittmenge zweier Mengen.
- ✓ Mit `select` wird die Untermenge einer Menge ermittelt, für die eine bestimmte Bedingung gilt. Um etwa die Menge aller Mitarbeiter zu erhalten, die mehr als 40.000 € verdienen, verwendet man den Ausdruck:
`self.mitarbeiter -> select (gehalt > 40000)`.

- ✓ Die Umkehrung der Operation `select` heißt `reject`. Folgender Ausdruck ermittelt somit die Menge der Mitarbeiter, die maximal 40.000 € verdienen:
`self.mitarbeiter -> reject (gehalt > 40000)`.
- ✓ `collect` liefert einen Bag (eine ungeordnete Menge), der die Ergebnisse der Überprüfung des entsprechenden Ausdrucks für alle Elemente der jeweiligen Menge enthält. Angenommen, Sie haben eine Menge mit fünf Mitarbeitern, die folgende Gehälter beziehen: {MA1.gehalt = 35000, MA2.gehalt = 88000, MA3.gehalt = 40000, MA4.gehalt = 37500, MA5.gehalt = 95000}. Der OCL-Ausdruck
`self.mitarbeiter -> collect (gehalt > 40000)`
liefert dann als Ergebnis die Menge: {false, true, false, false, true}.
- ✓ `asSet`, `asSequence` und `asBag` wandeln die verschiedenen Arten von Collections jeweils in eine andere Art um (Typ-Konvertierung).
- ✓ `append` hängt die zweitgenannte Sequenz an die erstgenannte Sequenz an.
- ✓ `prepend` hängt die erstgenannte Sequenz an die zweitgenannte Sequenz an.
- ✓ Mit der Operation `at (i)` lässt sich das Element an der Stelle `i` in einer Sequenz bestimmen.
- ✓ `first` liefert das erste, `last` das letzte Element einer Sequenz.
- ✓ Die Operation `iterate` ermöglicht die aufeinander folgende Auswahl aller Elemente einer Sequenz. Die komplette Sequenz wird in ihrer Reihenfolge, beginnend mit dem ersten Element, bis zum letzten Element durchlaufen.



Auch bei den Operationen mit Mengen ist darauf zu achten, dass die Datentypen korrekt eingesetzt werden. Vergewissern Sie sich daher immer, welche Arten von Elementen Sie mit einem OCL-Ausdruck verarbeiten und welchen Datentyp das Ergebnis des Ausdrucks annimmt.

Die Operationen der OCL ergänzen die UML um zahlreiche Möglichkeiten, Objektmodelle zu verfeinern. Darüber hinaus können OCL-Ausdrücke um einige Funktionen erweitert werden, die auch in Programmiersprachen verfügbar sind.

Eine dieser Funktionen ist der Ausdruck *let*, der die Einführung von Variablen in OCL-Ausdrücken ermöglicht. Auch Kontrollstrukturen, wie etwa bedingte Anweisungen, können verwendet werden. Ein Beispiel für einen solchen Ausdruck bietet die folgende Zusicherung:

```
context Mitarbeiter inv:  
  let stundenzahl : Real = self.stelle.zeit -> sum in  
  if self.alter < 18 then  
    stundenzahl <= 35  
  else stundenzahl > 35  
  endif
```

In diesem Ausdruck wird mit *stundenzahl* eine Variable vom Datentyp *Real* eingeführt, in der die gesamte Arbeitszeit eines Mitarbeiters aufgenommen wird. Die bedingte Anweisung (*if ... endif*) legt in Abhängigkeit vom Alter des Mitarbeiters fest, wie hoch die Zahl der geleisteten Stunden sein darf. Die Zusicherung besagt, dass Mitarbeiter unter 18 Jahren maximal 35 Stunden arbeiten dürfen. Mitarbeiter über 18 Jahren müssen mehr als 35 Stunden Arbeit leisten.

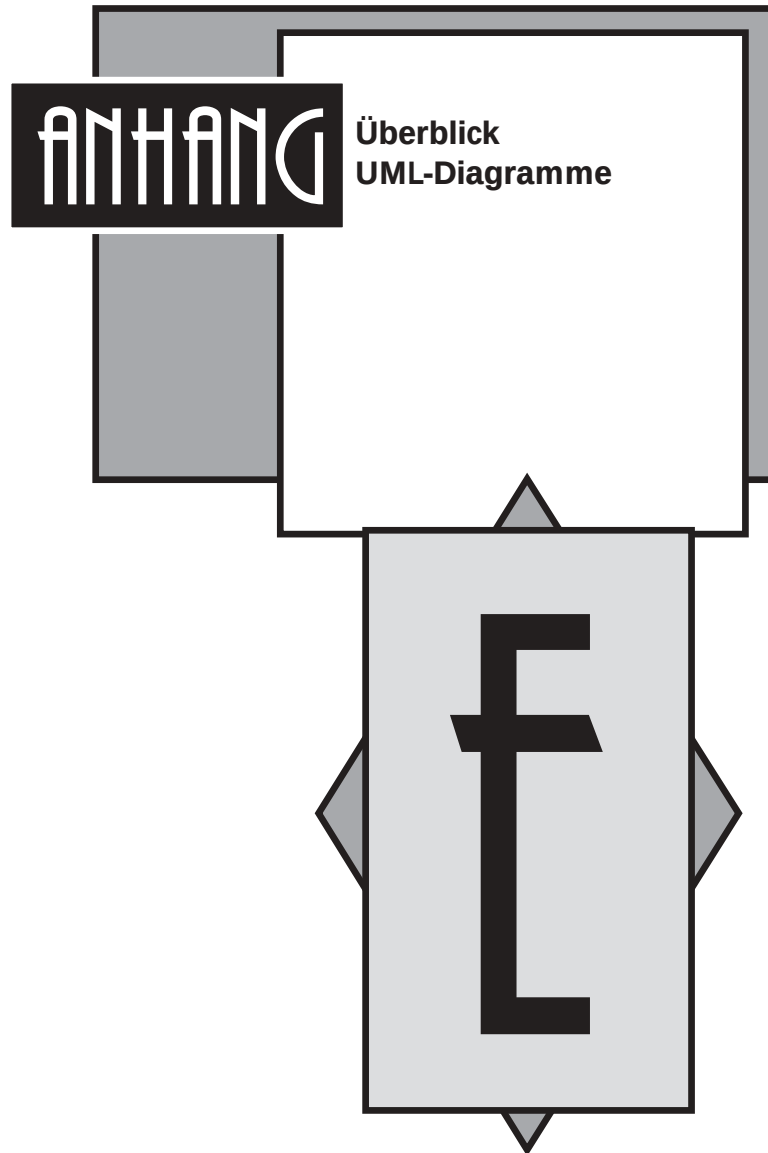


Bei der OCL handelt es sich nicht um eine Programmiersprache. Auch wenn sie einige Konstrukte enthält, die auch aus Programmiersprachen bekannt sind, dient sie ausschließlich der verfeinerten Beschreibung von Constraints. Ein OCL-Ausdruck liefert immer einen Wert als Ergebnis. Insbesondere sind Manipulationen des zugehörigen Objektmodells nicht möglich.

Kommentare in OCL-Ausdrücken werden durch zwei aufeinander folgende Bindestriche gekennzeichnet. Der Kommentar gilt jeweils bis zum Ende der jeweiligen Zeile. In dem letzten Beispiel zu der Arbeitszeit sehen Kommentare wie folgt aus:

```
context Mitarbeiter inv:
let stundenzahl : Real = self.stelle.zeit -> sum in
if self.alter < 18 then
    stundenzahl <= 35    -- unter 18J. max. 35 Std.
else stundenzahl > 35    -- ab 18J. mehr als 35 Std.
endif
```







Überblick UML-Diagramme

Auf den folgenden Seiten geben wir Ihnen einen Überblick über die verschiedenen Diagramme der UML. Er soll Ihnen als optische Hilfe dienen, um sich schnell in den Darstellungsmöglichkeiten der UML zurechtzufinden.

Klassendiagramm

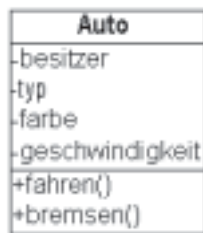


Abbildung E.1: Konkrete Klasse

Ein Klassendiagramm besteht aus den drei Elementen Name, Zustand (Attribute) und Verhalten (Operationen, Methoden).

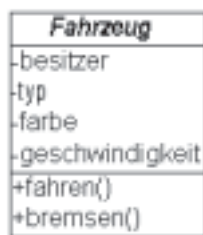


Abbildung E.2: Abstrakte Klasse

Abstrakte Klassen werden durch einen kursiv gedruckten Klassennamen symbolisiert.

Vererbung

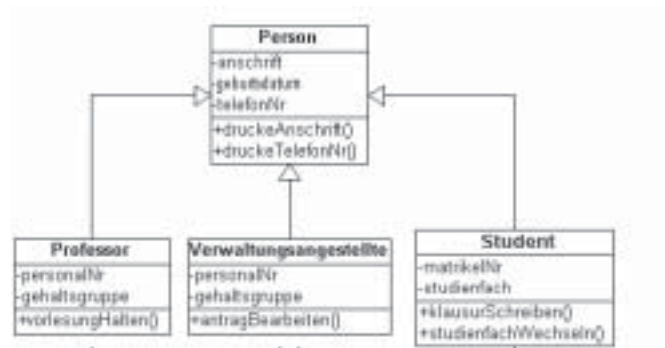


Abbildung E.3: Vererbung

Vererbungsstrukturen werden mit einem durchgezogenen Pfeil von der Unterklasse zur Oberklasse dargestellt. Die geerbten Eigenschaften werden in den Unterklassen nicht noch einmal explizit aufgeführt. Sie gelten als implizit vorhanden.



Abbildung E.4: Überschreiben

Nur beim Überschreiben einer Operation wird diese in der Unterklasse noch einmal aufgeführt. Sie zeigt dann ein anderes (spezielles) Verhalten als in der Oberklasse definiert.

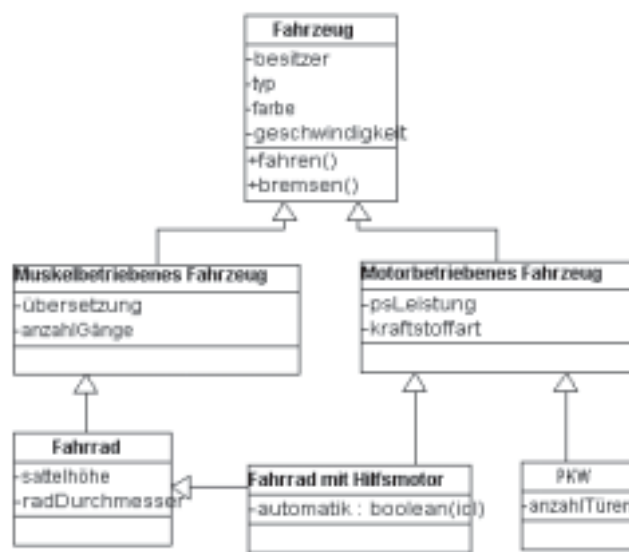


Abbildung E.5: Mehrfachvererbung

Mehrfachvererbung ist in der UML grundsätzlich darstellbar. Aufgrund möglicher Mehrdeutigkeiten bei der Interpretation von Eigenschaften sollte sie jedoch möglichst vermieden werden.

Use Case

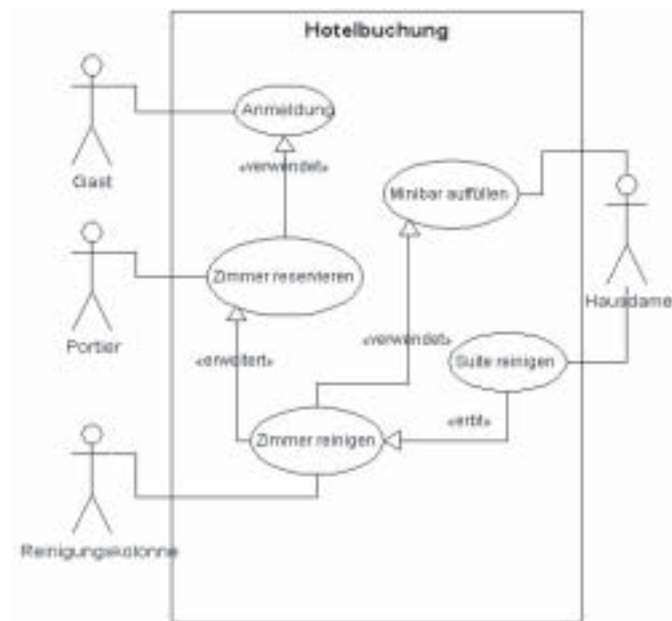


Abbildung E.6: Use-Case-Diagramm

Use-Case-Diagramme bestehen aus Anwendungsfällen, die jeweils einem Akteur zugeordnet sind. Zwischen den Anwendungsfällen können verschiedene Arten von Beziehungen bestehen.

Assoziation



Abbildung E.7: Binäre Assoziation

In einer binären Assoziation stehen genau zwei Klassen zueinander in Beziehung.

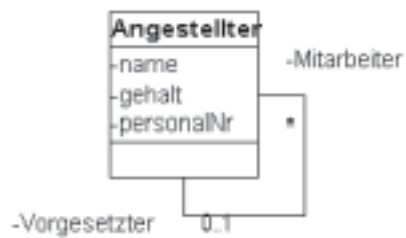


Abbildung E.8: Reflexive Assoziation

Ein Sonderfall der Assoziation ist der Selbstbezug (reflexive Assoziation), der eine Verbindung zwischen zwei Objekten derselben Klasse definiert.

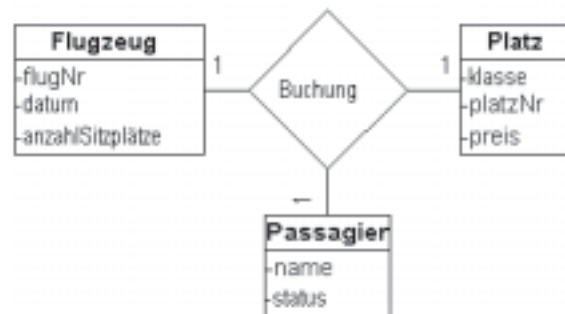


Abbildung E.9: N-äre Assoziation

Mehrgliedrige (n-äre) Assoziationen sind Beziehungen zwischen mehreren Klassen.

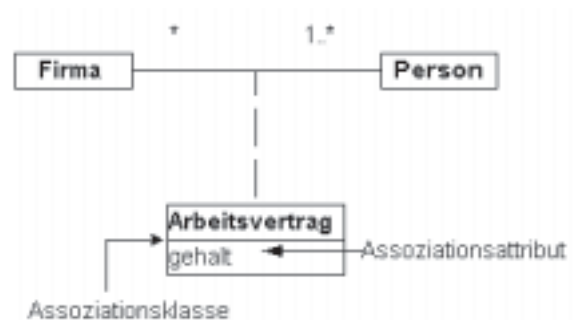


Abbildung E.10: Assoziationsklasse

Assoziationsklassen besitzen in ihrer Definition neben Attributen und Operationen auch Beziehungen zu anderen Klassen.

Zusicherung

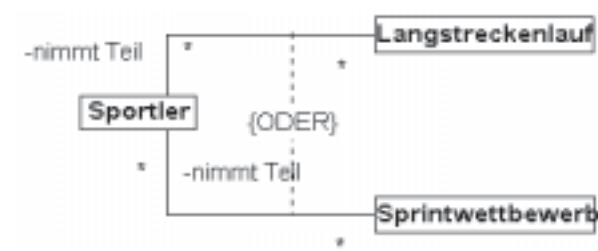


Abbildung E.11: Zusicherung oder



Abbildung E.12: Zusicherung subset

Zusicherungen sind Bedingungen, die Elemente oder Beziehungen in einem Objektmodell stets erfüllen müssen.

Aggregation



Abbildung E.13: Aggregation

Eine Aggregation ist eine besondere Form der Assoziation. Sie stellt eine Beziehung dar, bei der sich die Objekte der einen Klasse aus Objekten einer anderen Klasse zusammensetzen.

Komposition

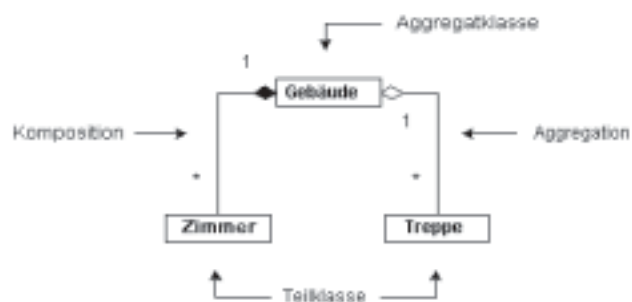
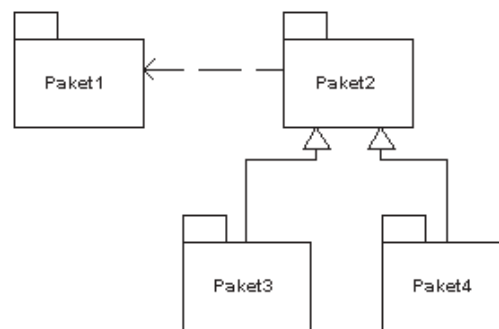


Abbildung E.14: Komposition

Eine Komposition ist eine strenge Form der Aggregation. Dabei können die Teile nicht ohne das Ganze (Aggregat) existieren.

Paket**Abbildung E.15:** Pakete

Mit Hilfe von Paketen lassen sich Problemstellungen gliedern. Sie bilden ein konsistentes Teilmodell und umfassen alle für den jeweiligen Problembereich relevanten Klassen.

Sequenzdiagramm

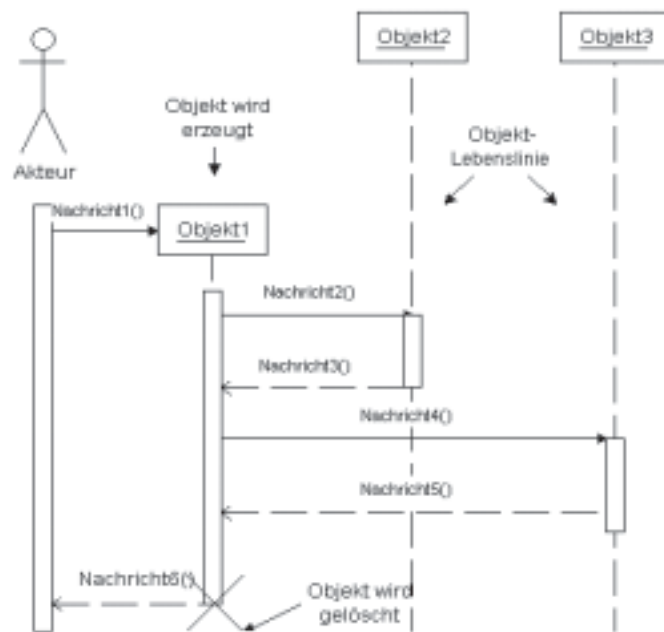
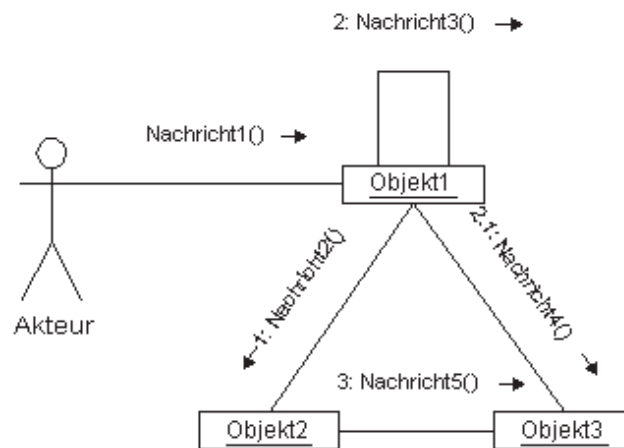


Abbildung E.16: Sequenzdiagramm

Sequenzdiagramme veranschaulichen Interaktionen zwischen Instanzen im Zeitablauf.

Kollaborationsdiagramm**Abbildung E.17:** Kollaborationsdiagramm

Kollaborationsdiagramme definieren die Verbindungen zwischen einer Menge von Objekten in einer bestimmten Situation.

Zustandsdiagramm

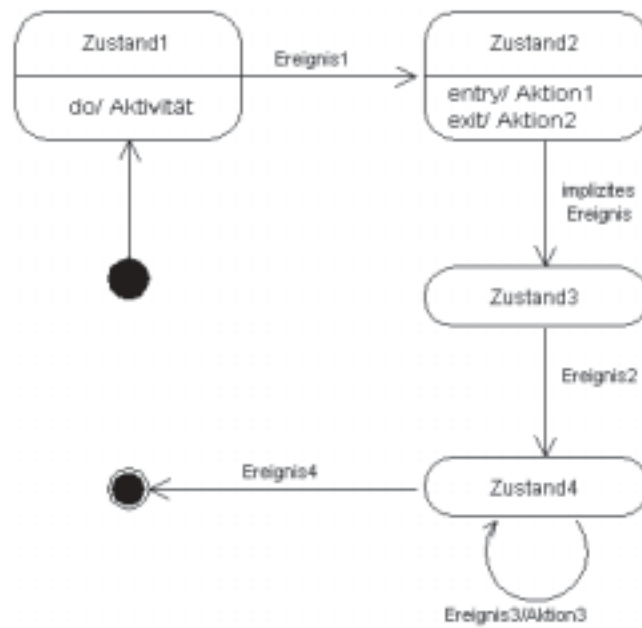


Abbildung E.18: Zustandsdiagramm

Ein Zustandsdiagramm dient der Darstellung der Zustände eines Objekts in seinem Lebenszyklus sowie der Abbildung von Zustandsänderungen und auslösenden Ereignissen.

Aktivitätsdiagramm

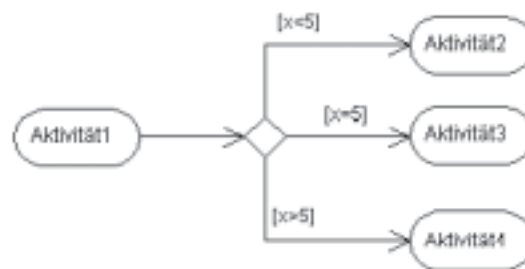


Abbildung E.19: Entscheidungen in Aktivitätsdiagrammen

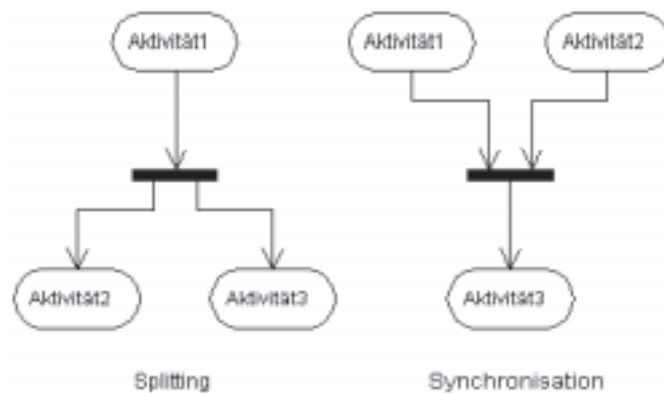


Abbildung E.20: Splitting und Synchronisation von Aktivitäten

Das Aktivitätsdiagramm stellt eine besondere Form des Zustandsdiagramms dar. Die Zustände sind hier mit einer Verarbeitung verknüpft.

Komponentendiagramm

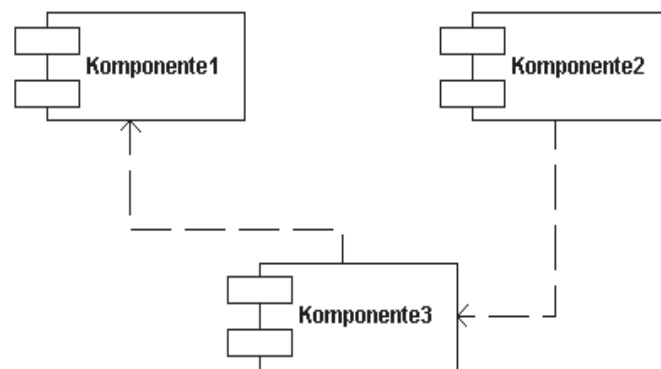
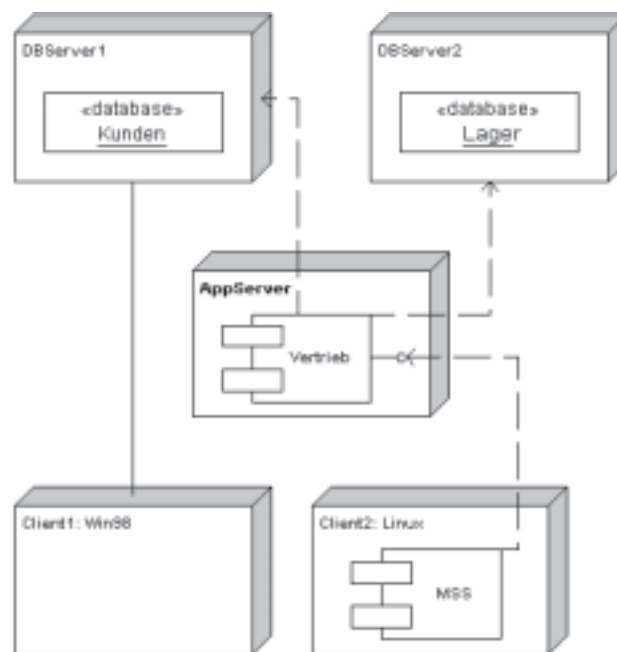


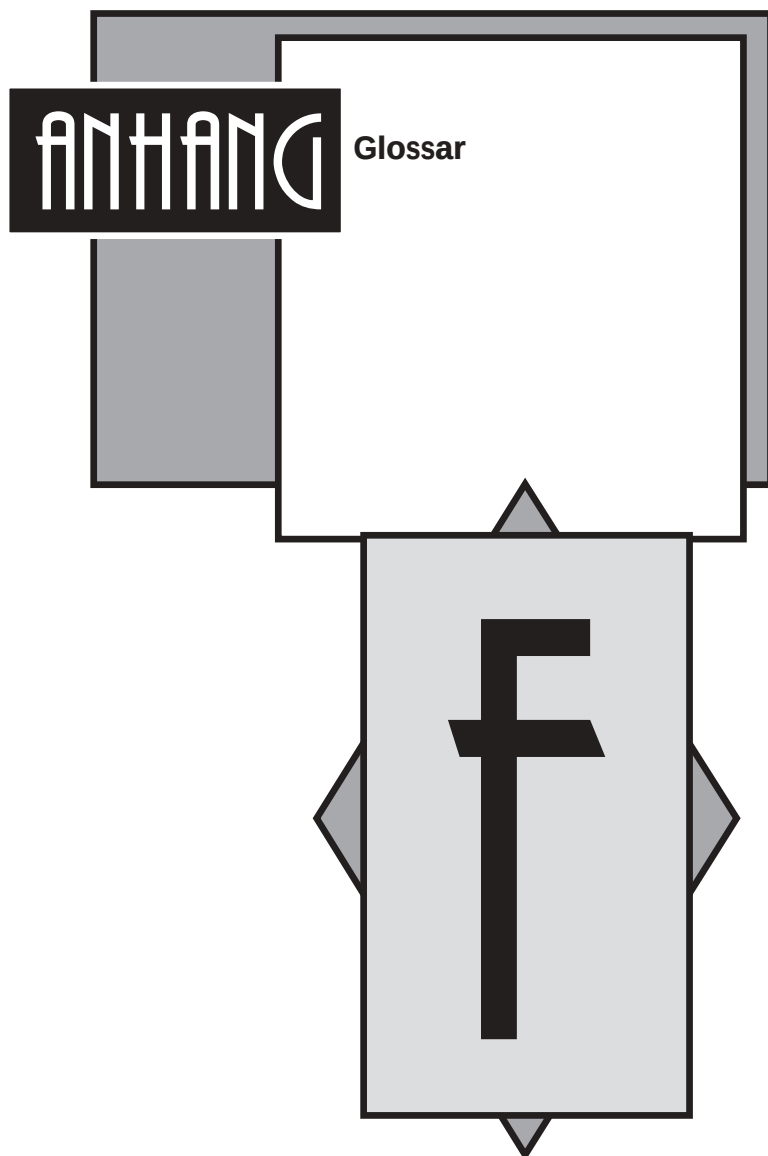
Abbildung E.21: Komponentendiagramm

Komponentendiagramme stellen Komponenten (wiederverwendbarer Programmcode) grafisch dar und verdeutlichen darüber hinaus die Abhängigkeiten zwischen den Software-Komponenten sowie ihren Schnittstellen.

Verteilungsdiagramm**Abbildung E.22:** Verteilungsdiagramm

Verteilungsdiagramme enthalten die Konfiguration einer Hard- und Software-Umgebung.







Glossar

Abhängigkeit

Eine Abhängigkeitsbeziehung zwischen zwei Elementen eines Objektmodells führt dazu, dass Änderungen des unabhängigen möglicherweise auch Änderungen des abhängigen Elements nach sich ziehen.

Abstraktion

Unter einer Abstraktion versteht man die Beschränkung eines Modells auf die für die jeweilige Sichtweise relevanten Modellelemente.

Abstrakte Klasse

Eine abstrakte Klasse bildet selbst keine Objekte. Sie beschreibt lediglich ein Muster für die Ableitung konkreter Klassen.

Aggregation

Die Aggregation ist eine spezielle Form der Assoziation. Sie beschreibt eine Beziehung zwischen zwei Klassen, bei der sich die Instanzen der einen Klasse aus Instanzen der anderen Klasse zusammensetzen.

Akteur

Ein Akteur ist eine Rolle, die von einer Person oder einem System außerhalb des betrachteten Systems eingenommen wird. Akteure stoßen Anwendungsfälle an.

Aktion

Eine Aktion ist ein einzelner Verarbeitungsschritt, der durch ein Ereignis ausgelöst wird. In Zustandsdiagrammen existieren Eintritts- und Austrittsaktionen. Erstere werden beim Eintritt, Letztere beim Verlassen eines Zustands ausgelöst.

Aktivität

Eine Aktivität ist ein Zustand mit einer internen Aktion, dem ein oder mehrere, durch Bedingungen verknüpfte Zustandsübergänge folgen. Einer der Zustandsübergänge wird nach Abschluss der internen Aktion automatisch ausgelöst.

Aktivitätsdiagramm

Aktivitätsdiagramme gehören zu den Verhaltensdiagrammen der UML. Ein Aktivitätsdiagramm ist eine besondere Form des Zustandsdiagramms. Die Zustände sind hier mit einer Verarbeitung verknüpft. In der Regel erfolgen Zustandsübergänge im Unterschied zum Zustandsdiagramm nicht durch Ereignisse, sondern durch die Beendigung eines mit dem Zustand verbundenen Verarbeitungsvorgangs.

Analyse

Die Analysephase dient im Rahmen der objektorientierten Software-Entwicklung der Ermittlung und Definition der Anforderungen an ein Softwaresystem. Ergebnis der Analysephase ist eine Leistungsbeschreibung des zu entwickelnden Systems in Form eines objektorientierten Modells.

Anwendungsfall

Ein Anwendungsfall beschreibt alle Aktivitäten sowie die beteiligten Akteure eines Systems. Anwendungsfälle werden von Akteuren initiiert und besitzen ein eindeutiges Resultat.

Anwendungsfalldiagramm

Anwendungsfalldiagramme beschreiben Anwendungsfälle mit ihren Beziehungen untereinander und zu beteiligten Rollen, Ereignissen oder Prozessen durch eine grafische Darstellung.

Assoziation

Eine Assoziation beschreibt eine Beziehung zwischen den Objekten von Klassen. Neben den beteiligten Klassen enthält die Beschreibung auch Kardinalitäten, Assoziations- und Rollennamen sowie mögliche Zusicherungen.

Assoziationsklasse

Eine Assoziationsklasse ist eine Assoziation, die selbst Attribute und Methoden besitzt. Assoziationsklassen können ihrerseits auch Assoziationen zu anderen Klassen bilden.

Attribut

Attribute beschreiben den Zustand eines Objekts. Sie definieren als statische Komponente die Daten, die Objekte einer Klasse enthalten können. Attribute haben einen bestimmten Datentyp und nehmen Attributwerte an. Alle Objekte einer Klasse besitzen dieselben Attribute.

Botschaft

Objekte kommunizieren über Botschaften (alternative Bezeichnung: Nachrichten). Ein Objekt (Client) sendet eine Botschaft an ein anderes Objekt (Server), um über dessen Methoden eine Dienstleistung nachzufragen. Das Server-Objekt reagiert mit der Ausführung einer Methode und antwortet gegebenenfalls seinerseits mit einer Botschaft.

Constraint

Synonym: Zusicherung.

Destruktor

Ein Destruktor ist eine Methode, die ein Objekt löscht. Sie wird in Objektmodellen meist als implizit vorhanden betrachtet und nicht gesondert aufgeführt (implizite Methode).

Domäne

Als Domäne wird eine reale Problemstellung bezeichnet, die in einem Softwaresystem in Ausschnitten abgebildet werden soll.

Dynamisches Binden

Beim dynamischen (späten) Binden wird einer Nachricht (Botschaft) erst zur Laufzeit des Programms eine bestimmte Methode eines Objekts zugeordnet. Im Gegensatz zum dynamischen Binden erfolgt diese Zuordnung beim frühen Binden bereits während der Kompilation eines Programms.

Dynamisches Modell

Das dynamische Objektmodell beschreibt das Verhalten des betrachteten Systems. Die Dynamik äußert sich in den Methoden der Objekte.

Einfachvererbung

Bei der Einfachvererbung gehört zu jeder Unterklasse maximal eine Oberklasse, deren Eigenschaften sie erbt. Es entsteht eine Baumstruktur.

Ereignis

Ein Ereignis tritt zu einem bestimmten Zeitpunkt auf. Es besitzt keine zeitliche Ausdehnung. Ereignisse lösen Zustandsübergänge aus. Sie können verkörpert werden durch Signale, Botschaften (Nachrichten) sowie als zeitliche Ereignisse durch das Verstreichen eines bestimmten Zeitraums oder das Erreichen eines bestimmten Zeitpunkts.

Exemplar

Der Begriff Exemplar ist eine alternative Bezeichnung für ein Objekt oder eine Instanz.

Geheimnisprinzip

Das Geheimnisprinzip bezeichnet die Unsichtbarkeit von Attributen und Methodenspezifikationen einer Klasse oder eines Objekts für andere Elemente des Objektmodells.

Generalisierung

Bei der Generalisierung werden Klassen mit ähnlichen Eigenschaften zu einer Oberklasse zusammengefasst. Gemeinsame Attribute, Methoden und Assoziationen der Unterklassen werden in der Oberklasse definiert und an die Unterklassen vererbt.

Geschäftsprozess

Ein Geschäftsprozess (Use Case) besteht aus einer Folge von Aktivitäten, die der Erfüllung einer zuvor definierten Aufgabe dienen.

Instanz

Der Begriff Instanz ist eine alternative Bezeichnung für ein Objekt oder ein Exemplar.

Instanziierung

Unter dem Begriff Instanziierung (auch: Instanziierung) versteht man das Erzeugen eines Exemplars einer Klasse.

Interaktionsdiagramm

Interaktionsdiagramm ist in der UML der Oberbegriff für Sequenz-, Kollaborations- und Aktivitätsdiagramme.

Kapselung

Als Kapselung bezeichnet man die Zusammenfassung von Name, Zustand und Verhalten eines Objekts.

Kardinalität

Die Kardinalität bezeichnet die Anzahl der Objekte, die an einer Assoziation beteiligt sind.

Klasse

Eine Klasse definiert die Eigenschaften einer Menge von Objekten mit ihren Attributen (Zustandsraum), Methoden (Verhalten) und Beziehungen. Alle Objekte einer Klasse besitzen dieselben Eigenschaften. Jede Klasse ist in einem Paket durch ihren Namen eindeutig identifizierbar.

Klassenattribut

Klassenattribute beschreiben Eigenschaften einer ganzen Klasse von Objekten.

Klassenmethode

Klassenmethoden beschreiben das Verhalten einer Klasse. Sie operieren nicht auf einem einzelnen Objekt, sondern auf der Klasse selbst.

Klassendiagramm

Klassendiagramme sind eine grafische Darstellung der Klassen, ihrer Assoziationen und Vererbungsstrukturen.

Kollaborationsdiagramm

Kollaborationsdiagramme gehören zu den Verhaltensdiagrammen der UML. Ein Kollaborationsdiagramm ist die grafische Darstellung von Objekten und ihrer Verbindungen. Die Verbindungen symbolisieren den Austausch von Nachrichten zwischen den Objekten.

Komponente

Eine Komponente ist ein wiederverwendbarer Programmcode, der eine abgegrenzte Aufgabe erfüllt.

Komponentendiagramm

Ein Komponentendiagramm ist die grafische Darstellung der Komponenten eines Systems. Es zeigt neben den Komponenten selbst auch deren gegenseitige Abhängigkeiten sowie ihre Schnittstellen.

Komposition

Die Komposition ist eine strenge Form der Aggregation. Die Existenz der Teile ist abhängig von der Existenz des Ganzen. Gleichwohl kann das Ganze ohne (alle) seine Teile existieren.

Konkrete Klasse

Eine konkrete Klasse ist eine Klasse, die Objekte erzeugen kann.

Konstruktor

Ein Konstruktor ist eine Methode, die Objekte einer Klasse erzeugt und initialisiert. Sie wird in Objektmodellen meist als implizit vor-

handen betrachtet und nicht gesondert aufgeführt (implizite Methode).

Mehrfachvererbung

Bei der Mehrfachvererbung (multiple Vererbung) können zu einer Unterklasse mehrere Oberklassen gehören, deren Eigenschaften sie erbt. Es entsteht eine Netzstruktur. Bei dieser Form der Vererbung kann es bei Namensgleichheit von Attributen oder Methoden zu Konflikten kommen, die durch entsprechende Konfliktlösungsmechanismen kanalisiert werden sollten.

Metaklasse

Metaklassen besitzen als Exemplare wiederum Klassen.

Metamodell

Ein Metamodell ist ein Modell zur Beschreibung einer Modellierungssprache.

Methode

Methoden beschreiben ein Vorgehen bei der Lösung einer bestimmten Aufgabe. In der UML ist eine Methode die Implementierung einer Operation. Häufig werden die Begriffe Methode und Operation synonym verwendet.

Multiple Vererbung

Synonym: Mehrfachvererbung.

Nachbedingung

Eine Nachbedingung ist ein Zustand, der nach der Verarbeitung einer Methode erreicht werden muss.

Nachricht

Synonym: Botschaft.

Oberklasse

Eine Oberklasse ist eine Klasse in einer Vererbungsstruktur, die ihre Eigenschaften an mindestens eine andere Klasse (Unterklasse) vererbt.

Object Constraint Language

Die Object Constraint Language (OCL) ist eine formale Sprache, mit der Constraints (Zusicherungen, Bedingungen) in Objektmodellen der UML spezifiziert werden können.

Object Management Group

Die Object Management Group (OMG) ist eine Vereinigung von zurzeit ca. 1000 Entwicklern und Anwendern objektorientierter Technologien, die versuchen, allgemein anerkannte Standards und Spezifikationen für die Objektorientierung zu schaffen, welche die Entwicklung offener objektorientierter Systeme ermöglichen.

Objekt

Ein Objekt ist die Repräsentation einer identifizierbaren, aus einem bestimmten Kontext isolierbaren Einheit, die durch einen Namen, einen Zustand und ein bestimmtes Verhalten beschrieben ist.

Objektdiagramm

Ein Objektdiagramm ist die grafische Darstellung der Objekte einer Problemdomäne und ihrer Beziehungen zu einem bestimmten Zeitpunkt.

Objektidentität

Jedes Objekt ist in einem objektorientierten Modell einzigartig und eindeutig identifizierbar. Auch wenn zwei Objekte zu einem bestimmten Zeitpunkt dieselben Attributwerte besitzen, so hat dennoch jedes seine eigene Identität.

Objektorientierte Analyse

Die objektorientierte Analyse (OOA) dient im Rahmen der objektorientierten Software-Entwicklung der Ermittlung und Definition der Anforderungen an ein Softwaresystem. Ergebnis der objektorientierten Analyse ist eine Leistungsbeschreibung des zu entwickelnden Systems in Form eines objektorientierten Modells.

Objektorientiertes Design

Das objektorientierte Design (OOD) baut auf dem Modell der objektorientierten Analyse auf und verfeinert es aus Sicht der Realisierung. Ergebnis dieser Phase der objektorientierten Software-Entwicklung ist ein Designmodell.

Objektorientierte Programmierung

Die objektorientierte Programmierung (OOP) implementiert die Ergebnisse der Analyse und des Design in einer objektorientierten Programmiersprache.

Objektorientierte Software-Entwicklung

Die objektorientierte Software-Entwicklung ist ein Konzept zur Herstellung von Software, das auf den objektorientierten Paradigma beruht. Die Vorgehensweise gliedert sich dabei in die Phasen objektorientierte Analyse (OOA), objektorientiertes Design (OOD) und objektorientierte Programmierung (OOP).

Operation

Operationen sind Dienstleistungen, die Objekte anderen Objekten nach Aufforderung über eine Nachricht zur Verfügung stellen. In der UML werden Operationen durch Methoden implementiert. Häufig werden die Begriffe Operation und Methode synonym verwendet.

Paket

Ein Paket fasst zusammengehörige Modellelemente unter einem gemeinsamen Namen zusammen. Mit Hilfe von Paketen lassen sich Problemstellungen gliedern. Jedes Paket umfasst alle für den jeweiligen Problembereich relevanten Klassen.

Parameter

Ein Parameter ist eine Variable, die einer Methode bei ihrem Aufruf zur Verarbeitung übergeben wird.

Persistenz

Persistenz bezeichnet die Eigenschaft eines Objekts, über die Laufzeit des Programms hinaus zu existieren. Ein solches persistentes Objekt steht beim Neustart des Programms wieder zur Verfügung, ohne dass Informationen verloren gehen.

Polymorphismus

Polymorphismus bedeutet »Vielgestaltigkeit«. In der Objektorientierung ist Polymorphismus ein Konzept, das es Objekten erlaubt, auf ein und dieselbe Nachricht unterschiedlich zu reagieren.

Problemstellung

Synonym: Domäne.

Reflexive Assoziation

Reflexive Assoziationen beschreiben Beziehungen zwischen Objekten ein und derselben Klasse.

Rolle

Rollen repräsentieren Klassen bzw. Objekte in Assoziationen.

Schnittstelle

Als Schnittstelle bezeichnet man die Menge der Signaturen, die in den Methoden einer Klasse definiert sind.

Sequenzdiagramm

Sequenzdiagramme gehören zu den Verhaltensdiagrammen der UML. Ein Sequenzdiagramm ist die grafische Darstellung der zeitlichen Abfolge von Interaktionen zwischen einer ausgewählten Menge von Objekten.

Sichtbarkeit

Die Sichtbarkeit bestimmt die Zugriffsmöglichkeiten auf Attribute und Methoden. Die UML unterscheidet die Sichtbarkeiten public, protected und private.

Signatur

Die Signatur einer Methode besteht aus dem Namen, den Parametern und dem Rückgabewert der Methode.

Spätes Binden

Synonym: dynamisches Binden.

Spezialisierung

Bei der Spezialisierung werden allgemeine Eigenschaften einer Klasse (Oberklasse) in Unterklassen konkretisiert und erweitert. Die Unterklassen erben die Eigenschaften der Oberklasse, passen sie ihren speziellen Anforderungen an und ergänzen sie um weitere Eigenschaften (Attribute, Methoden, Beziehungen).

Statisches Modell

Das statische Objektmodell beschreibt die Elemente des betrachteten Systems. Es umfasst die Klassen mit ihren Assoziationen und ihrer Vererbungsstruktur. Darüber hinaus definiert es die Attribute mit ihren Datentypen und benennt die Methoden.

Stereotyp

Stereotypen sind spezifische Erweiterungen der Modellelemente der UML.

Subklasse

Synonym: Unterklasse.

Superklasse

Synonym: Oberklasse.

Swimlane

Synonym: Verantwortungsbereich.

Transition

Eine Transition ist ein Zustandsübergang. Sie verbindet einen Ausgangszustand mit einem Folgezustand und wird durch ein Ereignis ausgelöst.

Überladen

Beim Überladen wird der Methodenname innerhalb einer Klassenbeschreibung mehrfach aufgeführt. Jede dieser Methoden besitzt eine andere Parameterliste. Beim Methodenaufwurf erfolgt die Identifikation der »richtigen« (also tatsächlich gemeinten) Methode durch die übergebene Parameterliste.

Überschreiben

Beim Überschreiben wird in einer Unterklasse die aus einer Oberklasse geerbte Methode neu definiert.

Unterklasse

Eine Unterklasse ist eine Klasse in einer Vererbungsstruktur, die Eigenschaften einer anderen Klasse (Oberklasse) erbt, diese ihren Anforderungen anpasst und durch eigene Eigenschaften erweitert. Eine Unterklasse kann die Eigenschaften einer (Einfachvererbung) oder mehrerer Oberklassen (Mehrfachvererbung) erben.

Use Case

Synonym: Anwendungsfall.

Verantwortlichkeitsbereich

Der Verantwortlichkeitsbereich (swimlane) definiert in einem Aktivitätsdiagramm die Zuständigkeit von Rollen für eine bestimmte Menge von Aktivitäten.

Vererbung

Die Vererbung ist eine Beziehung zwischen einer Oberklasse (Generalisierung) und einer Unterklasse (Spezialisierung). Die Unterklasse übernimmt (erbt) die Eigenschaften der Oberklasse, verändert und ergänzt sie um eigene Eigenschaften. Bei der Einfachver-

erbung gehört zu jeder Unterklasse genau eine Oberklasse. Bei der Mehrfachvererbung (multiple Vererbung) hat eine Unterklasse mehr als eine Oberklasse.

Verhalten

Unter dem Verhalten eines Objekts versteht man seine Aktionen und seine Reaktionen auf die Botschaften anderer Objekte. Das Verhalten eines Objekts wird in seinen Methoden definiert.

Verhaltensdiagramm

Ein Verhaltensdiagramm repräsentiert dynamische Sachverhalte innerhalb eines objektorientierten Modells. Es stellt die Auswirkungen der Ausführung von Methoden grafisch dar. Die UML unterscheidet vier Arten von Verhaltensdiagrammen: Sequenzdiagramme, Kollaborationsdiagramme, Zustandsdiagramme und Aktivitätsdiagramme.

Verteilungsdiagramm

Ein Verteilungsdiagramm ist die grafische Darstellung der Konfiguration einer Hard- und Software-Umgebung. Es zeigt, welche Komponenten auf welchen Knoten (physische Objekte, wie Computer, Speicher, Prozessoren) verarbeitet werden und welche Kommunikationsbeziehungen zwischen den Knoten bestehen.

Vorbedingung

Eine Vorbedingung ist ein Zustand, der vor der Verarbeitung einer Methode erreicht werden muss.

Zusicherung

Zusicherungen sind Bedingungen in einem objektorientierten Modell. Sie schränken Inhalte oder Zustände ein und definieren bestimmte Eigenschaften, die in jedem Fall vorhanden sein müssen.

Zustand

Ein Zustand ist eine bestimmte Zeitspanne im Lebenszyklus eines Objekts. Er wird definiert durch die Werte, die den Attributen des Objekts während eines Zeitraums zugeordnet sind. Ein Zustand besteht so lange, bis ein Ereignis eintritt, das zu einem Zustandsübergang führt. Während eines Zustands kann ein Objekt Aktionen ausführen.

Zustandsautomat

Ein Zustandsautomat besteht aus einer Menge von Objektzuständen und Transitionen. Dazu gehört stets ein Anfangszustand und möglicherweise ein Endzustand.

Zustandsdiagramm

Zustandsdiagramme gehören zu den Verhaltensdiagrammen der UML. Ein Zustandsdiagramm stellt die Zustände eines Objekts in seinem Lebenszyklus sowie die Zustandsübergänge und die auslösenden Ereignisse grafisch dar.

Zustandsübergang

Synonym: Transition.

Index

A

Abhängigkeit 181, 187, 745
abstract 292
Abstraktion 52, 83, 745
Activity Graphs Package 81
Aggregation 162, 175, 286, 371, 433, 494, 734, 745
 geteilte 166
 Hierarchie 165
 mehrstufige 164
 mehrteilige 166
Akteur 90, 345, 396, 398, 730, 745
 Aufgaben 399
Aktion 746
 Do- 211
 Entry- 211
 Exit- 211
 Include- 211
Aktivität 220, 339, 374, 387, 388, 453, 507, 746
Aktivitätsdiagramm 192, 220, 456, 508, 739, 746
Analyse 339
 dynamische 374
 grobe System- 340, 342, 357
 statische 355
Anthropozentrik 32
Anwendungsfall 89, 91, 339, 340, 344, 345, 347, 465, 746
Anwendungsfalldiagramm 251, 467, 730, 747

ArrayList 280
Assembly 327
Assoziation 29, 145, 163, 175, 177, 271, 286, 339, 356, 371, 373, 376, 433, 492, 731, 747
 abgeleitete 157
 binäre 146, 731
 gerichtete 158
 mehrgliedrige 153
 n-äre 153, 732
 qualifizierte 155
 reflexive 153, 731, 756
 rekursive 153
Assoziationsklasse 150, 286, 732, 747
Attribut 54, 112, 118, 208, 269, 339, 359, 417, 486, 727, 747
 Klassen- 125, 750
 Meta- 79
 qualifizierendes 155
Aufwandschätzung 338
Außensicht 395

B

Bag 717, 718
base 294, 301
Basistyp 716
Bedingung
 Nachbedingung 715, 752
 Vorbedingung 715, 759
Behavioral Elements Package 81
Benutzungsoberfläche 261

Beziehung 93
Beziehungstyp
 dynamischer 267
Booch 76
Botschaft 747

C

C# 19, 25, 139, 268
C++ 19, 21, 25, 44, 46, 139
Cast-Operator 302
CD-ROM 641
Collaborations Package 81
Collection 280, 718
Common Behavior Package 81
complete 143
Constraint 143, 713, 748
Context 714
Core Package 80
Count 281

D

Data Types Package 81
Datenkapselung 57
Datentyp 119, 717
Denkstil 19
Design 339
Design-Prinzipien 75
Destruktor 748
Diagramm 82
 Aktivitäts- 84
 Anwendungsfall- 83, 344
 Implementierungs- 84
 Klassen- 84
 Kollaborations- 84
 Komponenten- 84

Sequenz- 84
Verhaltens- 84
Verteilungs- 84
Zustands- 84
disjoint 143
Domäne 748
Dynamisches Binden 748

E

Eiffel 45
Eigenschaft 21
Einfachvererbung 748
Entity-Relationship-Modellierung 148
Ereignis 90, 339, 346, 375, 385, 402, 450,
 506, 749
Exemplar 115, 749
Extension Mechanisms Package 80

F

Foundation Package 80

G

Geheimnisprinzip 23, 749
Generalisierung 63, 93, 130, 365, 488,
 749
Geschäftsprozess 395, 749
 Merkmale 398
 Struktur 403
Geschäftsvorfall 468
Global Declaration Space 326
Granularität 99

H

Hashtable 284

I

IBEL 713
IComparable 284
Implementierung 339
Implementierungsdiagramm 186
incomplete 144
Indizierer 309
Informationsquelle 356
Instanz 60, 749
Instanzierung 750
Interaktion 339, 374, 376, 378, 438
Interaktionsdiagramm 498, 750
Interface 118, 127, 304
internal 327
Invarianten 714
is-Operator 315

J

Java 19, 25, 45, 139, 231

K

Kardinalität 146, 148, 163, 169, 279, 750
Klasse 21, 36, 59, 84, 103, 112, 182, 268, 338, 339, 354, 356, 359, 365, 414, 480, 486, 750
 abstrakte 61, 144, 728, 745
 assoziative 150, 747
 Basis- 289
 Beschreibung 362, 376, 428
 konkrete 59, 727, 751
 Meta- 79, 118
 Name 363
 Ober- 130, 289, 728
 Ober- (Superklasse) 753, 757
 parametrisierbare 118
 Sub- 289

 Super- 289

 Unter- 130, 289, 728

 Unter- (Subklasse) 757, 758

Klassenattribut 750

Klassendiagramm 103, 111, 253, 727, 751

Klassenmethode 750

Kollaborationsdiagramm 191, 202, 444, 737, 751

Komponente 751

Komponentendiagramm 187, 740, 751

Komposition 168, 175, 286, 371, 433, 494, 734, 751

Konstruktor 293, 751

Kontext 714

L

Let 722

M

MagicDraw 578, 680

 Systemanforderungen 681

Maschinenzentrik 32

Mehrfachvererbung (multiple Vererbung) 729, 752

Metaklasse 752

Metamodell 752

Methode 20, 57, 486, 510, 752

 Klassenmethode 750

 Überladen 758

 Überschreiben 758

Model Management Package 82

Modell 51, 74, 342

 dynamisches 78, 111, 341

 fachliches 262

 Meta- 78

statisches 78, 111, 341
 Struktur- 78
 Verhaltens- 78
 Modellierung 89, 243
 Modellierungssprache 74
 Modifizierer 327, 331

N

Nachbedingung 715, 752
 Nachricht 193, 753
 Name 52
 Namespace 267, 320

O

object 297
 Object Constraint Language (OCL) 119, 713, 753
 Object Management Group (OMG) 77, 753
 Objekt 23, 49, 79, 84, 112, 115, 228, 338, 753
 Destruktion 194
 Konstruktion 193
 Objektdiagramm 111, 753
 Objektidentität 754
 Objektlebenszyklus 192
 Objektmodell
 dynamisches 438, 497, 748
 statisches 413, 477, 757
 Objektorientierte Analyse (OOA) 69, 746, 754
 Objektorientierte Programmierung (OOP) 69, 754
 Objektorientierte Softwareentwicklung 754
 Objektorientiertes Design (OOD) 69,

754

Objektorientierung 19, 31, 49
 Objektzustand 228
 OMT 76
 OOSE 76
 Operation 57, 112, 121, 182, 210, 339, 359, 375, 389, 419, 457, 727, 755
 Klassen- 125
 Meta- 79
 Overlapping 143
 Overloading 300
 Override 303, 312

P

Paket 80, 179, 255, 339, 340, 350, 354, 363, 408, 410, 471, 474, 735, 755
 Interdependenzen 411
 Schnittstellen 412
 Parameter 755
 Persistenz 755
 Perspektive 99
 Phasenmodell 240
 Polymorphie 66, 137, 299
 Polymorphismus 755
 Poseidon 609, 690
 Installation 691
 Systemanforderungen 690
 Post 715
 Pre 715
 Problemabgrenzung 396, 405
 Problemausschnitt 339, 340, 348
 Abgrenzung 337, 349, 376
 Problemstellung 468, 755
 Projektmanagement 240

R

Rational Rose 548, 658
 Installation 661
 Systemanforderungen 659
Redefinition 300
Reflection 277
Remove 281
Restriktion 346, 401
 finanzielle 343
 personelle 343
 Sachmittel- 343
 zeitliche 343
Reverse-Engineering 243
Rolle 90, 193, 756

S

Schnittstelle 118, 127, 182, 260, 304,
 331, 339, 340, 346, 353, 400, 474, 756
Self 714
Sequence 717, 718
Sequenzdiagramm 106, 191, 192, 257,
 443, 499, 736, 756
Set 717, 718
Sicht
 Außen- 82, 89, 340
 dynamische 83
 externe 89
 Innen- 83, 89, 340
 interne 89
 statische 83
Sichtbarkeit 123, 326, 756
 package 124
 private 124, 329
 protected 124, 327, 329
 protected internal 327
 public 124, 327, 329

Signatur 756
Simula 32, 45
Smalltalk 21, 36, 45, 262
Software Engineering 429
Software-Entwicklung 235
Softwareentwicklungszyklus 240
Spätes Binden 756
Spezialisierung 63, 130, 365, 367, 488,
 757
Splitting 223
Spracharchitektur 78
Sprachelemente 78
State Machines Package 81
static 293
Stereotyp 129, 182, 206, 757
 access 182
 association 206
 friend 183
 global 206
 import 183
 lokal 206
 parameter 206
 self 206
Steuerungsfokus 193
Strukturmodell 111
subset 162
Swimlane 226
Synchronisation 223
Synchronisationsbedingung 204
System Architect 568, 671
 Installation 672
 Systemanforderungen 671
Systemanalyse
 grobe 395, 465

T

Together 517, 645
 Installation 647
 Systemanforderungen 645
Toolübersicht 632
Transition 208, 220, 757

U

Überladen 67, 300, 758
Überschreiben 67, 137, 300, 729, 758
Unified Method 77
Use Case 81, 84, 89, 344, 347, 398, 404,
 730, 749, 758
using 320, 327

V

Verantwortlichkeitsbereich 226, 758
Vererbung 21, 33, 42, 63, 93, 130, 177,
 180, 287, 339, 363, 365, 369, 430, 488,
 728, 758
 Einfach- 134, 748
 Mehrfach- 134, 752
 mehrstufige 136
 multiple 370, 752
Vererbungshierarchie 136
Verhalten 23, 36, 57, 103, 112, 121, 727,
 759

Verhaltensdiagramm 191, 759
Verhaltensmodell 111
Verteilungsdiagramm 189, 741, 759
virtual 303
Vorbedingung 715, 759
Vorgehen
 evolutionäres 240
Vorgehensmodell 76, 101
Vorgehensmuster 240

W

Wiederverwendung 59

Z

Zusicherung 143, 160, 713, 733, 759
Zustand 23, 36, 45, 52, 54, 104, 112, 208,
 220, 339, 374, 381, 445, 727, 760
 End- 209
 Objekt- 228
 Start- 209
 Unter- 215
Zustandsautomat 760
Zustandsdiagramm 191, 208, 448, 500,
 738, 760
Zustandsübergang 208, 211, 339, 375,
 381, 445, 760