Peter Zöller-Greer

Software
Analyse
und
Design

Software Analyse und Design

beschreibt den Prozess von der Auftragserteilung über die Entwicklung bis zur Installation einer Anwendungssoftware. Der Buchtitel deutet schon an, dass das eigentliche Programmieren völlig in den Hintergrund tritt; durch leistungsfähige Softwarewerkzeuge kann die eigentliche Kodierung einer Software zum großen Teil automatisiert werden. Voraussetzung dafür ist allerdings, dass eine detallierte Planung vorhanden ist. Der Entwurf der Software ist daher heute mehr denn je das Gütesiegel dafür, kostengünstig und effizient Software zu entwicklen.

In der gleichen Reihe erschienen:

Prof. Dr. rer. nat. Peter Zöller-Greer
ist Mathematiker und unterrichtet die Fächer Künstliche Intelligenz, Software-Engineering und Multi Media Systeme an der FH Frankfurt am Main-University of Applied Sciences.

9 783981 163919
Preis: 22,- Euro
ISBN 978-3-9811639-1-9

Software Analyse und Design

Zöller-Greer

Die Reihe *Wissen & Praxis >kompakt<* nimmt sich komplexer Themen an und versucht diese so einfach wie möglich, beschränkt auf das Wesentliche, für Studium und Praxis gleichermaßen geeignet darzustellen.

Composia Verlag
www.composia.de

Composia Verlag

Peter Zöller-Greer

Software Analyse und Design

Grundlagen und Anwendungen

Mit einer Einführung in Phasenmodelle, Software-Architekturen, Projektplanung, MDA, UML, Entwurfsmuster und Maskenentwurfstechniken

Reihe *Wissen & Praxis >kompakt<*

Composia Verlag

Peter Zöller-Greer

Software Analyse und Design

Grundlagen und Anwendungen

Mit einer Einführung in Phasenmodelle, Software-Architekturen, Projektplanung, MDA, UML, Entwurfsmuster und Maskenentwurfstechniken

Prof. Dr. Peter Zöller-Greer
studierte nach Abschluss einer Berufsausbildung als Physiklaborant (BASF-AG Ludwigshafen/Rh.) von 1975-1981 Mathematik (Diplom) und Theoretische Physik an den Universitäten Siegen und Heidelberg. Er promovierte an der Universität Mannheim über eine approximationstheoretische Lösung eines Problems aus der Quantenmechanik und arbeitete zunächst als Systemanalytiker bei Brown Boveri Reaktor GmbH und danach als DV-Referent bei ABB Mannheim, bevor er 1989 die Geschäftsführung der Firma Composia GmbH in Mannheim übernahm. Bereits während dieser Tätigkeiten arbeitete er als freier Dozent an verschiedenen Hochschulen. Seit 1993 ist er Professor am Fachbereich Informatik und Ingenieurwissenschaften der FH Frankfurt am Main – University of Applied Sciences. Seine Lehr- und Arbeitsgebiete sind Künstliche Intelligenz, Software-Engineering und Multimedia-Systeme.

Bibliographische Information der Deutschen Bibliothek:
Die Deutsche Bibliothek verzeichnet diese Publikation in der
Deutschen Nationalbibliographie; detaillierte bibliographische
Daten sind im Internet über http://dnb.ddb.de abrufbar.

2. Auflage

Alle Rechte vorbehalten.

© Composia Verlag, Wächtersbach, 2010

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

ISBN 978-3-9811639-1-9

Inhalt:

Vorwort		**4**
1	**Einführung**	**5**
2	**Methodische Ansätze**	**10**
3	**Software-Architekturen**	**20**
4	**Projektplanung und Pflichtenheft**	**36**
5	**Datenmodellierung**	**54**
6	**Entwurfsmuster**	**90**
7	**Anwendungsentwurf**	**100**
Anhang	**Lösungen der Übungsaufgaben**	**125**
	Weiterführende Literatur	**138**
	Index	**139**

Vorwort

Kaum ein Gebiet ist in den letzten Jahren so stark dem Wandel unterlegen wie das Software-Engineering. Während dieser Begriff in den Anfangsjahren der Programmentwicklung gar nicht existierte, ist er heute zum Zentrum des Informatikstudiums geworden. Und das mit gutem Grund: die eigentliche Programmierarbeit ist heute der kleinste Teil der Softwareentwicklung geworden. Viel wichtiger sind dagegen die richtige Analyse und das Design des zu entwickelnden Systems. Alles scheint darauf abzuzielen, dass in Zukunft gar keine Programmierarbeiten von Nöten sein werden. Es gibt heute schon Werkzeuge, die eine Softwareentwicklung „fast" ohne Programmierung ermöglichen. Die meiste Energie muss daher im Vorfeld, bei der Planung und dem Entwurf, eingesetzt werden. Wird hier alles „richtig" gemacht, kann die eigentliche Kodierung des Programms fast vollständig automatisiert werden oder es können zumindest auf vorhandene Bausteine (z.B. Entwurfsmuster) zurückgegriffen werden, welche dann leicht zur gewünschten Anwendung „zusammenkomponiert" werden können. Nach einer kurzen Einführung in die Thematik (Kapitel 1) werden einige populäre Phasenmodelle besprochen (Kapitel 2) und wichtige Software-Architekturen erläutert (Kapitel 3). Mit der Kenntnis dieser Dinge kann dann das Projekt geplant und eine semantische Datenmodellierung in einem Pflichtenheft festgehalten werden (Kapitel 4). Der Softwareentwickler beginnt dann mit der abstrakten Datenmodellierung (Kapitel 5). Kapitel 6 gibt eine kurze Einführung in Entwurfsmuster. Danach ist der Entwickler in der Lage, Programmoberflächen und Funktionen zu entwerfen bzw. zu implementieren (Kapitel 7).

Zu jedem dieser Kapitel gibt es eigene, z.T. sehr umfangreiche Lehrbücher, daher können die hier präsentierten Kapitel das jeweilige Gebiet natürlich nur einführend behandeln.

Ich möchte an dieser Stelle meiner lieben Frau Diana für ihre unermüdliche Geduld mit mir danken. Dank auch unseren „Cratchits", die mir einen neuen Blick auf das Wesentliche gaben.

Im Februar 2010

Peter Zöller-Greer

1. Einführung

Als die ersten Computer in den 40er und 50er Jahren gebaut wurden, hat sich schnell gezeigt, dass die Programmierung solcher Rechner vom System weg hin zu allgemeineren (Sprach-)Konzepten notwendig wird. Nicht nur die Rechnergenerationen entwickelten sich fort, auch die „Sprachgenerationen“. Nachfolgende Tabelle gibt einen Überblick über deren Verlauf, wobei die jeweiligen Sprachen nicht unbedingt an die gleiche Rechnergeneration gebunden sind (z.B. laufen Sprachen der 5. Generation auf Rechnern der 3. Generation etc.).

Nr.	Rechnergeneration	Sprachgeneration
1	Röhren- und Relais (bis 1958)	Maschinensprache
2	Transistoren (1958-1966)	Assemblersprachen
3	Integrierte Schaltkreise (1966-1975)	„Höhere“ Programmiersprachen
4	Microprozessoren (seit 1975)	Abfragesprachen (SQL etc.)
5	Reduktions- und Parallelrechner (seit 1980)	KI-Sprachen[1]
6	Bio-Chip-Rechner (??)	??
7	Quantenrechner (??)	??

Diese Einteilung variiert allerdings in der Literatur (wie vieles im Software-Engineering). In den 60er Jahren des 20. Jahrhunderts verbreiteten sich Computer in vielen Industrieunternehmen. Dabei steckte jedoch die systematische Entwicklung darauf laufender Software noch in den Kinderschuhen. Software wurde normalerweise in einem zentralen Rechenzentrum für umliegende Fachabteilungen von einem Spezialistenteam entwickelt. Wobei hier der Ausdruck „Team“ nicht unbedingt meinte, dass mehrere Personen am gleichen Projekt zusammenarbeiteten, sondern es war häufig so, dass jeder Programmierer komplett ein Programm oder zumindest einen abgegrenzten Programmteil entwickelte, der relativ isoliert gewesen ist. Hinzu kam die Tatsache, dass solche Programme nicht in großem Umfang geplant wurden, sondern die Programmierer begannen häufig einfach mit der Programmierung loszulegen. Und dies zudem noch, bevor genau klar war, was überhaupt der spätere Benutzer wollte. Dieses relativ unsystematische Vorgehen erzeugte natürlich hohen Entwicklungsaufwand und damit auch Unsicherheit hinsichtlich der Kosten und der Entwicklungsdauer.

Es wurde schon bald die Notwendigkeit einer besseren Planung von Software erkannt und begonnen, sog. Pflichtenhefte zu schreiben. Diese enthielten mehr oder weniger präzise die jeweiligen Anforderungen an die zu entwickelnde Software. Aufbau und Form solcher Pflichtenhefte unterlagen allerdings keinem

[1] vgl. z.B. Zöller-Greer, P.: *Künstliche Intelligenz*, Composia Verlag, 2007

bestimmten Standard, und es war dem jeweiligen Autor überlassen, wie genau das Problem und evtl. Lösungen beschrieben wurden.
Man ging zu dieser Zeit auch dazu über, die Planung der Software von ihrer Kodierung zu trennen. Sogenannte Systemanalytiker hatten die Aufgabe, das Problem zu analysieren und eine algorithmische Lösung zu entwickeln, die dann einem Programmierer zum Zwecke der Kodierung übergeben wurde. Im kommerziell-administrativen Bereich waren dies auch tatsächlich verschiedene Personen. Im naturwissenschaftlich-technischen Bereich dagegen wurden Systemanalyse und Programmierung häufig von ein- und derselben Person durchgeführt. Dies machte insofern Sinn, als hier die Probleme oft so komplex waren, dass die Kommunikation zwischen Analytiker und Programmierer sehr schwierig gewesen wäre.
Obwohl also einige Anstrengungen unternommen wurden, war der gesamte Entwicklungszyklus einer Software noch von vielen Unsicherheitsfaktoren begleitet. Abgeschätzte Entwicklungskosten wurden regelmäßig erheblich überschritten, ebenso wie die geplanten Entwicklungszeiten. Man fand heraus, dass dies in erster Linie an mangelhafter Planung lag. Planungsfehler wurden oft erst sehr spät entdeckt und waren nur mit erheblichem Aufwand zu korrigieren. Außerdem wiesen die Pflichtenhefte häufig Inkonsistenzen (Widersprüche) auf, die zunächst nicht bemerkt wurden. Diese Situation bezeichnet man heute als die Softwarekrise der 60er Jahre.
Es wurden viele Anstrengungen unternommen, um aus dieser Krise herauszukommen. So wurde zunächst z.B. versucht, die Planungsphase zu standardisieren, wobei ingenieurwissenschaftliche Methoden als Vorbild dienen sollten. Die Grundidee war, besagte Planungsphase systematisch zu erarbeiten und methodische Ansätze dafür zu entwickeln. Man erhoffte sich dadurch weniger Planungsfehler und eine Kodierung zu ermöglichen, welche Programme liefern sollte, die die geforderten Spezifikationen so gut wie möglich erfüllen.

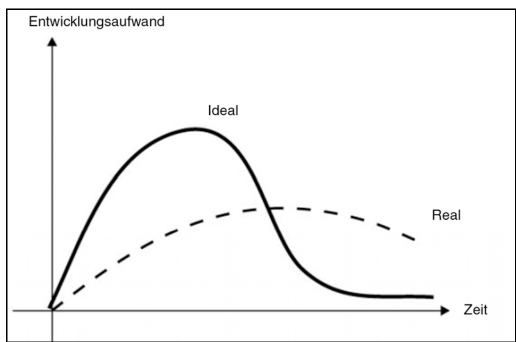


Abb. 1.1 Lebenszyklus einer Software

Abb. 1.1 stellt den idealen Lebenszyklus einer Softwareentwicklung dem seinerzeit realen gegenüber. Im Idealfall steckt man den meisten Aufwand in die Planungsphase, so dass die sich anschließende Kodierungsphase ohne großen Aufwand vonstatten geht und auch keine wesentlichen Korrekturen erforderlich macht. Im Realfall dagegen war die Planungsphase nicht sonderlich intensiv durchdacht worden und die Kodierung war demzufolge fehlerhaft und korrekturanfällig. Die wohlbekannten Gesetzte von Edward A. Murphy finden leider auch hier ihre Anwendung.

Murphys Gesetze:
Alles dauert länger als man denkt
Alles ist teurer als man denkt
Alles ist komplizierter als man denkt
Wenn ein Fehler passieren kann, dann passiert er auch.
(→ Murphy war Optimist)

Diese etwas scherzhafte Formulierung von Murphys Gesetzen weist auf die Probleme hin, die mit der Entwicklung von Software verbunden sein können. Es führte die Softwarekrise der 60er Jahre also zu der Notwendigkeit, Softwareapplikationen systematisch zu planen und zu entwickeln. Dies wiederum führte zu der wissenschaftlichen Disziplin des Software Engineering. Um allerdings effiziente Software zu entwickeln, bedarf es einer systematischen Lösung von Problemen der Art:

- Komplexe Anforderungen sind oft schwer zu verstehen
- Softwareentwicklung erfordert absolute Präzision, einen hohes Maß an Kreativität, Erfahrung im Entwickeln komplexer Programme und echtes Teamwork
- Unter Stressbedingungen kann die Fehlerrate ansteigen
- Es ist eine hohe Qualifikation der Mitarbeiter ist erforderlich

An einen Entwickler werden zudem noch gewisse spezifische Anforderungen gestellt, wie z.B. die Fähigkeit, Wichtiges von Unwichtigem trennen zu können, oder die richtige Einschätzung der Komplexität einer Anforderung.
Um komplexe und aufwendige Anforderungen einer systematischen Lösung zuführen zu können, sind drei wichtige Prinzipien einzuhalten:

- Die Modellierung der Realität durch Abstraktion,
- die Reduktion der Komplexität durch Strukturierung und
- Fokussierung auf die Lösung von Teilproblemen.

Es gibt computergestützte Hilfsmittel zur Durchführung dieser Aufgaben, die man allgemein CASE-Tools nennt (CASE = Computer Aided Software Engineering). Es wird dringend empfohlen, solche Tools wann immer möglich einzusetzen. Diese Werkzeuge sind in der Lage, schon bei der Planung und dem

Entwurf der Anwendung Inkonsistenzen zu entdecken. Dadurch reduziert sich die Fehlerrate beträchtlich. Außerdem können viele dieser Tools das Planungsergebnis direkt in ein relationales oder objektorientiertes Datenbankschema umsetzen und erzeugen so z.B. physikalische Tabellenstrukturen.
Nachfolgend soll nun eine allgemeine Definition des Begriffs Software Engineering gegeben werden.

Definition 1.1 (**Software Engineering**)
Unter *Software Engineering* versteht man die Wissenschaft, effiziente Software auf der Basis ingenieurmäßiger Methoden und ökonomischer Gesichtspunkte zu entwickeln. Sie umfasst die strukturierte Analyse, den Entwurf und die Realisierung sowie das Testen und Warten eines Programms mittels methodischer Ansätze.

Bis zu den achtziger Jahren des 20. Jahrhunderts unterteilte man die Entwicklung eines Softwaresystems zunächst nur in die zwei Phasen *Systemanalyse* und *Programmierung*. Die Systemanalyse umfasste damals die Problembeschreibung, Beschreibung des Ist- und Sollzustandes, Input/Output-Beschreibung des gewünschten Systems, einen Projektplan, ein semantisches Datenmodell, ein logisches Datenmodell, Datenfluss-Diagramme/Programm-Struktogramme und ein Lösungskonzept. Die sich daran anschließende Phase Programmierung beschäftigte sich mit der eigentlichen Kodeerzeugung, ggf. einem „Tuning" des Kodes sowie schließlich dem Test und der Wartung der Software.
Diese beiden Phasen sind heute allerdings nur Teile detaillierterer Phasenmodelle und umfassen demzufolge auch nicht mehr alle oben genannten Komponenten, sondern diese sind ausgelagert und in andere, eigene Entwicklungsphasen integriert. Gelegentlich wird eine Unterteilung des Software-Engineering in zwei Komponenten vorgeschlagen:

Konstruktionssystematik
Darunter versteht man die Ansammlung der benutzen Methoden, Werkzeuge und Ergebnisse

Produktionssystematik
Diese beschreibt die Entwicklungsschritte, deren Reihenfolge, gesetzte Termine sowie die anfallenden Kosten.

Es sei darauf hingewiesen, dass diese zwei Komponenten nichts mit den Phasen des Software-Engineering zu tun haben! Die Phasen fließen eher in die 2. Komponente ein, während die erste einfach die Betriebsmittel zu ihrer Durchführung bereit stellt.

Die Lösung der Probleme der Softwarekrise des letzten Jahrhunderts erhoffte man sich also durch Phasenkonzepte. Der Entwicklungszyklus einer Anwendung wird in einzelne Phasen unterteilt und jede Phase enthält genau definierte Eingaben, Ausgaben und Tätigkeiten. Man unterscheidet dabei zwischen linearen und nichtlinearen Phasenmodellen. Bei linearen Phasenmodellen muss eine Phase vollständig abgearbeitet sein, bevor die nächste Phase begonnen werden kann. Bei nichtlinearen Phasenmodellen ist das nicht so; hier können einzelne Phasen auch mehrmals durchlaufen werden, ohne immer vollständig abgeschlossen zu sein. Solche Phasenmodelle sind Inhalt des nächsten Kapitels.

2. Methodische Ansätze

Die Komplexität einer Anwendungsentwicklung legt nahe, das Projekt in einzelne Phasen zu unterteilen. Auf anderen Gebieten ist dies ja schon länger erfolgreich gang und gäbe. Möchte beispielsweise jemand ein Haus bauen, so beauftragt er ja auch nicht direkt einen Maurer, welcher dann drauf losmauert, ohne Plan, und dauernd den Kunden danach fragt, ob es so oder so gerade recht ist. Würde dem Kunden dann nämlich etwas nicht gefallen, so müssten Mauern wieder eingerissen und von Neuem gebaut werden. Bis ein Haus schließlich den Vorstellungen des Kunden entspricht, würde sicher viel Zeit und Geld verbraucht sein. Aus diesem Grund gibt es Architekten, die zuerst einmal mit dem Kunden zusammen eine Planung ausarbeiten. Erst wenn diese zur Zufriedenheit des Kunden abgeschlossen ist, wird normalerweise ein Auftrag zum Bau eines Hauses erteilt. Was aber beim Hausbau selbstverständlich erscheint, setzte sich erst langsam bei der Softwareentwicklung durch. Wie in Kapitel 1 angedeutet, gab es zunächst nur die beiden Phasen *Systemanalyse* und *Programmierung* (wobei der Begriff Systemanalyse heute eine etwas andere Bedeutung hat). Der Systemanalytiker erledigte alles „Planerische“, wobei darunter auch die Programmablaufpläne verstanden wurden, und der Programmierer setzte die Programmablaufpläne in eine Programmiersprache um. Es zeigte sich nach und nach die Notwendigkeit, diese beiden Phasen feiner zu unterteilen, insbesondere mit der seit den 80er Jahren des 20. Jahrhunderts neu entstandenen Möglichkeit der „Eingreifens“ eines Benutzers während des Programmablaufs. War nämlich im bisherigen „Batch-Betrieb“ (Lochkarten) das EVA-Prinzip vorherrschend (Eingabe-Verarbeitung-Ausgabe), so konnten jetzt durch „Computer-Terminals für jeden“ auch die Anwender den Programmablauf beeinflussen (durch Bildschirmeingaben, von denen der weitere Programmablauf abhing etc.).

Das klassische Wasserfallmodell

Die ursprüngliche grobe Einteilung des Entwicklungsprozesses in nur zwei Phasen entspricht der Einteilung bei der Errichtung eines Hauses in die zwei Phasen Planung und Realisierung. Eine subtilere Unterteilung wurde jedoch aus den beschriebenen Gründen erforderlich. Es zeigte sich außerdem, dass innerhalb dieser beiden groben Phasen noch zu viele Fehlerquellen auftraten, welche dann aber durch weitere Systematisierung der einzelnen Arbeitsschritte beseitigt werden konnten. Eine bis heute noch oft erfolgreich anwendbare Systematisierung ist das klassische Wasserfallmodell. In der Literatur findet man manchmal Abweichungen hinsichtlich der Anzahl und des Inhalts der einzelnen Phasen, doch im Prinzip handelt es sich immer um folgende Einteilung:

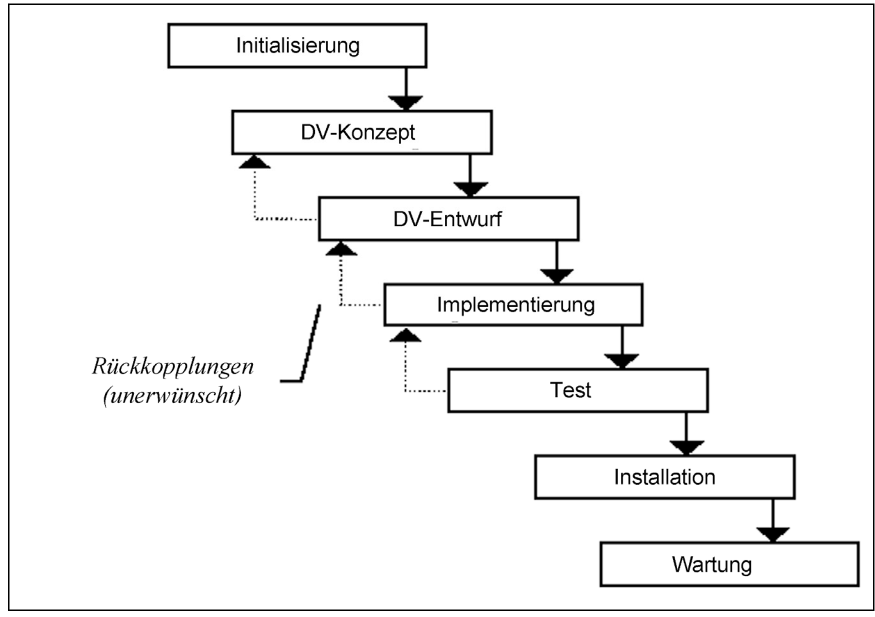


Abb. 2.1 Wasserfallmodell

Das Wasserfallmodell zählt zu den linearen Phasenmodellen. Das heißt, eine Phase kann erst begonnen werden, wenn die davor liegende vollständig beendet ist. Wir schauen uns jetzt die einzelnen Phasen genauer an.

Phase 1: Initialisierung

Die Initialisierungsphase startet das geplante Entwicklungsprojekt. In der Regel wird damit begonnen, dass ein Unternehmen das geplante Projekt ausschreibt oder sich direkt an ein Software-Entwicklungsunternehmen wendet mit der Bitte um eine Angebotserstellung. Zur Erstellung eines Angebotes ist es erforderlich, dass eine erste Analyse des Problems erfolgen muss, aus dem die notwendige Information für das Angebot ableitbar ist. Ein Problem des angeboterstellenden Unternehmens ist dabei, dass einerseits ein detailliertes Angebot erst erfolgen kann, wenn hinreichende Information über das Projekt vorliegt und andererseits eben eine solche erste Problemanalyse bereits recht arbeitsaufwendig werden kann, je nach dem, welchen Umfang das geplante Projekt haben soll. Lehnt der Kunde das Angebot ab, ist die für die erste Analyse investierte Zeit verloren. Deshalb wird ein Entwickler zunächst eine recht grobe Analyse beim Kunden

vornehmen, die aber trotzdem eine erste Kostenabschätzung sowie eine erste zeitliche Abschätzung des Projektverlaufs möglich macht.
Eine mittlerweile sich immer mehr verbreitende Variante ist, dass sich ein Softwareentwicklungsunternehmen bereits die Angebotserstellung bezahlen lässt. Dies ist insbesondere bei komplexen Projekten der Fall, wo bereits ein großer Aufwand getrieben werden muss, um überhaupt ein einigermaßen realistisches Angebot abgeben zu können. Deshalb wird schon für eine Angebotserstellung oft eine umfangreiche Detailanalyse erforderlich, und dieser Aufwand muss dann auch honoriert werden. Der Kunde hat aber auch einen Vorteil von dieser Vorgehensweise, denn er hat dann eine ausführliche Analyse des Entwicklers vorliegen, welche er, selbst wenn er das Angebot ablehnt, weiter verwerten kann (diese kann er z.B. einem anderen Systementwickler zur Verfügung stellen).
Die Initialisierungsphase enthält in jedem Fall mindestens folgende Punkte:

- *Problembeschreibung*, in der Zweck und Rechtfertigung für das geplante Unterfangen genannt wird
- *Projektziele* (gewünschtes Ergebnis des Projekts)
- *Grobe Projektbeschreibung*; welche bereits installierte Hard- und Software ist vorhanden, welche zusätzliche Hard- und Software muss angeschafft werden, sind Netzwerkkomponenten erforderlich etc.
- *Grober Projektplan*, welcher eine personale und zeitliche Aufwandsplanung und erste grobe Projektorganisation enthält
- *Kostenabschätzung*
- *Ein Angebot an den Kunden*

Eine zuverlässige Aufwands- und Kostenabschätzung birgt jedoch die größte Problematik. Kunden bevorzugen in der Regel Festpreise, aber gerade die setzen eine realistische Kostenabschätzung voraus, und, wie beschrieben, würde das eine intensive Problemanalyse erfordern. Erfahrene Entwickler habe hier deutliche Vorteile, da sie oft ein recht sicheres Gefühl für Art und Umfang anstehender Entwicklungsaufgaben haben.
Falls Festpreise gewünscht sind, dann ist es in jedem Fall ratsam, einen geeigneten Puffer für nicht vorhersehbare Fälle (vgl. Murphys Gesetze aus Kapitel 1) einzuplanen. Manche Entwickler setzen z.B. einen Faktor 2 für die Entwicklungskosten im Angebot an den Kunden gegenüber des zunächst tatsächlich abgeschätzten Aufwands an, vor allem dann, wenn viele Unsicherheiten bestehen hinsichtlich der Problemanalyse.
Wenn keine Festpreise erforderlich oder wenigstens nicht verbindlich sind, so kann vereinbart werden, dass eine genaue Aufwandschätzung erst nach Abschluss der nächsten Phase vorgelegt werden muss, also wenn das DV-Konzept vorliegt. In diesem Fall möchte der Kunde jedoch häufig wenigstens einen Kos-

tenvoranschlag für die Erstellung des DV-Konzepts haben, welcher dann im Angebot angegeben wird. Die zeitliche Dauer der Initialisierungsphase sollte so gewählt werden, dass für den Softwareentwickler kein zu hohes Risiko für den Fall entsteht, dass der Kunde das Angebot ablehnt.

Phase 2: DV-Konzept (Fachkonzept)

Diese Phase dient u.a. hauptsächlich zur Erstellung des „klassischen" Pflichtenhefts. Die in der Initialisierungsphase noch relativ groben Analysen müssen jetzt verfeinert werden, so dass ein detailliertes Konzept entsteht, mittels dessen ein Programmentwickler später seine Datenmodelle und Programmablaufpläne entwickeln kann (das DV-Konzept wird auch manchmal *Grobkonzept* genannt).
In Kapitel 4 wird diese Phase im Einzelnen genau beschrieben, so dass hier nicht weiter darauf eingegangen werden muss. Es bleibt hier allenfalls anzumerken, dass das Pflichtenheft von Entwickler und Auftraggeber *gemeinsam* geplant werden muss (manchmal basieren Pflichtenhefte auf einem Anforderungskatalog, den der Auftraggeber vorab allein erstellte; dieser Anforderungskatalog wird auch Lastenheft genannt). Ein Pflichtenheft stellt im juristischen Sinne einen Vertrag dar und muss daher auch von beiden Parteien verstanden sein. Dies führt in der Regel dazu, dass ein Pflichtenheft eher *verbal* die Datenstrukturen und Abläufe beschreibt. So eine Darstellung heißt auch *semantisches Datenmodell*.

Phase 3: DV-Entwurf

Der eigentliche Entwurf des Anwendungssystems geschieht hier. Das Ergebnis wird auch manchmal *Feinkonzept* genannt. Dabei spielt diese Phase die Rolle, die z.B. beim Hausbau die Erstellung eines Bauplans, des „Blueprints", spielt. So wie ein Architekt auf dem Papier das zukünftige Haus plant (nach den Wünschen des Auftraggebers, bei uns im Pflichtenheft niedergeschrieben), so plant der Entwickler in der Phase DV-Entwurf die Einzelheiten der DV-Anwendung. Und ebenso wie beim Architekten ist das Ergebnis auch in der Datenverarbeitung ein Bauplan im wahrsten Sinn des Wortes, der zum großen Teil tatsächlich aus grafischen Elementen auf Papier, man könnte es einen Programm-Blueprint nennen, besteht. Dazu zählen Darstellungen wie das Entity-Relationship-Diagramm (ERD) oder auch die UML (Unified Modelling Language). Das Ergebnis dieser Phase ist also eine Programmspezifikation, in der alle Einzelheiten beschrieben sind, die zur eigentlichen Programmierung des Systems erforderlich sind. Dazu gehört übrigens auch eine detaillierte Planung von Teststrategien zwecks Verifikation der entwickelten Programme.

Phase 4: Implementierung
Wenn der Entwurf erfolgreich beendet ist, dann kann die Anwendung programmiert werden. Diesen Vorgang nennt man Implementierung, da jetzt der Entwurf in kodierter Form auf den Computer gebracht wird. Je nach Anwendungsgebiet kann es sich dabei um die berühmte echte „Knochenarbeit“ handeln, wenn z.B. ein Echtzeitsystem in C^{++} geschrieben werden muss. Es kann aber auch sein, dass gar nicht viel im herkömmlichen Sinne programmiert werden muss, sondern dass durch den geschickten Einsatz von Werkzeugen vieles ohne Programmierarbeit erledigt werden kann. Gerade bei Datenbankanwendungen z.B. für betriebliche Informationssysteme sind sehr mächtige Tools vorhanden. Im Idealfall kann der Programmcode sogar automatisch generiert werden. Dazu ist es natürlich erforderlich, dass der DV-Entwurf hinreichend detailliert und formalisiert ist. Werkzeuge, welche (fast) keine Programmierung mehr erforderlich machen, sind ein hochaktuelles Forschungsgebiet des Software-Engineering. Die Vorgehensweise für die Implementierung ist im Allgemeinen „top-down“, d.h. es werden zuerst die einzelnen Module geplant und diese dann jeweils verfeinert und implementiert. Das Ergebnis der Phase Implementierung ist also ein ausführbares Programm mit einer evtl. physikalisch implementierten Datenbasis.

Phase 5: Test
Der Implementierung des Programms folgt ein ausführlicher Test. Dabei sind zwei Testverfahren zu unterscheiden.

Der Programmtest
Hierunter versteht man den Test der einzelnen Programm-Module auf logische Konsistenz und Übereinstimmung mit dem DV-Design. Dieser Test wird i.d.R. vom Programmierer selbst durchgeführt. Normalerweise beginnen diese Tests nicht erst am Ende der Implementierungsphase, sondern bereits während der Kodierung, d.h. während an den entsprechenden Modulen programmiert wird, werden auch gleich die dazu gehörigen Tests gemacht. Selbstverständlich wird nach Abschluss der Implementierung noch mal ein größerer Gesamttest durchgeführt, um das Zusammenspiel der einzelnen Module zu überprüfen und die Konsistenz mit dem DV-Design festzustellen. Für diese Art des Testens gibt es kaum formale Methoden, da die verschiedenen Möglichkeiten des Programmierstils einfach zu vielfältig sind. Als Faustregel kann man allenfalls sagen, dass die Module jeweils für sich so weit wie möglich ausgetestet werden. Ist dies erfolgt, so sollten dann einige zusammengehörige Module gemeinsam, d.h. deren Zusammenspiel, getestet werden. Man testet also Gruppen von Modulen, bis schließlich das ganze Programm ausgetestet ist. Handelt es sich z.B. um wissenschaftliche Anwendungen, so sollte eine verifizierbare Methode im voraus wohl überlegt werden. Soll ein Programm z.B. die numerische Lösung eines Differen-

tialgleichungssystems ermöglichen und sind dafür numerische Stützstellen für eine Interpolations- oder Approximationsfunktion vorgesehen, so kann ein sehr effektiver Test dadurch geschehen, dass man einige der Stützstellen innerhalb ihres Konvergenzintervalls leicht verschiebt. Damit hat man numerisch gesehen völlig andere Eingabe-Zahlen als zuvor, aber das Ergebnis der Lösung müsste davon unabhängig sein. Auch hilft es häufig, Extremfälle zu betrachten, deren Ergebnis im Voraus bekannt ist (z.B. triviale Fälle, wo alle Eingaben Null sind etc.).

Der Benutzertest
Ist das Programm durch den Programmtest als logisch richtig und fehlerfrei verifiziert, so muss ein Test unter Produktionsbedingungen erfolgen. Dieser Test kann zwar auch vom Programmierer durchgeführt werden, es ist aber besser, wenn hier ausgewählte Benutzer testen. Wichtig ist dabei, dass keinesfalls schon echte Produktionsprozesse gefahren werden, sondern es soll ja nur getestet werden in einer *produktionsidentischen* Umgebung, aber nicht in der Produktion selbst. Es kann nämlich hier noch zu folgenreichen Fehlern im Programm kommen, und das darf natürlich nicht den Produktionsprozess behindern. Auch müssen die Benutzer, welche diese Tests durchführen, wissen, dass es sich hier um einen Test handelt, welcher u.U. zu unerwarteten Ergebnissen führen kann. Die Planung über die Art und Weise dieser Tests sowie der beteiligten Personen sollte bereits im Fachkonzept festgelegt werden.

Phase 6: Installation
Nach Abschluss der Testphase erfolgt die Installation der Anwendung in der Produktionsumgebung. Es sollte dafür ein Zeitpunkt gewählt werden, welcher unkritisch für die laufende Produktion ist. Das heißt, dass gleichzeitig keine kritischen Anwendungen dabei laufen dürfen. Auch muss eine Datensicherung des Zielrechners vorhanden sein, so dass ein Recovery möglich ist.
Mit der Übergabe des Anwendungsprogramms wird normalerweise eine Installationsroutine mitgeliefert, welche es einem unbedarften Anwender ermöglicht, das Programm nach starten einer Setup-Routine gemäß den Anweisungen auf dem Bildschirm zu installieren.
Bei kritischen Produktionsumgebungen oder sehr komplizierten Programmsystemen kann es sinnvoll sein, einen Installationstest auf einem produktionsidentischen System zu machen, bevor man auf dem tatsächlichen Zielrechner installiert.

Phase 7: Wartung
Normalerweise ist die Wartungsphase -jedenfalls bei hinreichend getesteten Programmen- wenig arbeitsintensiv. Manchmal werden gewisse Schulungsmaß-

nahmen (Anwenderschulungen etc.) vorgenommen, welche aber gewöhnlich zeitlich begrenzt sind. Läuft das Programm erst einmal richtig, so bleibt hier nicht mehr viel zu tun. Manchmal kann es vorkommen, dass kleinere Fehler erst nach längerer Zeit entdeckt werden oder dass gewisse Programmerweiterungen gewünscht werden. Solche Fälle können als Teil eines Wartungsvertrages im Voraus abgedeckt werden. Erfolgte dies nicht, dann muss in diesen Fällen ein neues Angebot erstellt und das ganze wie ein neues Projekt behandelt werden.

Rückkopplungen im Wasserfallmodell
Angenommen, beim Benutzertest stellt sich heraus, dass eine Ausgabe nicht das gewünschte Ergebnis liefert. Der Benutzer geht damit dann zum Programmierer. Wenn es sich nun um keinen Programmierfehler handelt, so wird der Programmierer den „Planer", d.h. denjenigen, der den DV-Entwurf getätigt hatte, ansprechen. Hat auch dieser keinen Fehler bei der Planung gemacht, so wird der schließlich im Fachkonzept nachschauen und dort den Fehler lokalisieren können. Das Fachkonzept aber wurde zusammen mit dem Auftraggeber nach Erstellung „abgesegnet", d.h. beide, Entwickler und Auftraggeber hatten das Fachkonzept als verbindlich abgezeichnet. Jetzt muss also das Fachkonzept korrigiert werden. Daraufhin müssen die Phasen DV-Entwurf, Implementierung und Test erneut durchlaufen werden. Dieser Sachverhalt stellt die unerwünschten Rückkopplungen dar.
Wenn man Pech hat, dann kann dies alles sogar mehrmals geschehen. Es gibt Aussagen, dass ein Fachkonzept mit mehr als 20 Seiten immer an irgendeiner Stelle widersprüchlich ist. Aus diesem Grund ist es gerade bei umfangreicheren Projekten sinnvoll, schon frühzeitig computergestützte Werkzeuge einzusetzen. Es gibt bereits hilfreiche Tools für die Erstellung von Fachkonzepten, welche gewisse Inkonsistenzen frühzeitig entdecken können. Solche Tools setzen voraus, dass die Spezifikation allerdings sehr genau durchgeführt wird. Es ist klar, dass bedingt durch den linearen Verlauf der Phasen im Wasserfallmodell Fehler erst sehr spät entdeckt werden. Zudem bekommt ein Anwender erst zu einem sehr späten Zeitpunkt das erste Mal etwas von dem Anwendungsprogramm zu sehen, nämlich frühestens beim Benutzertest. Bis dahin ist aber schon der Hauptanteil der Entwicklungszeit verbraucht. Bei größeren Entwicklungsprojekten kann es durchaus sein, dass ein Anwender erst ein Jahr nach Auftragserteilung zum ersten Mal etwas auf seinem Computer zu sehen bekommt. Das ideale Wasserfallmodell (ohne Rückkopplungen) setzt also etwas voraus, was es eigentlich nicht gibt: Ideale Auftraggeber, die zum Zeitpunkt der Auftragsvergabe haargenau wissen, was sie wollen und was das Programm später können soll, und ideale DV-Entwickler, die völlig fehlerfrei arbeiten und ebenfalls alle möglichen Probleme vorausgedacht haben. So etwas gibt es natürlich nicht. Deswegen hat sich nach und nach eine weitere Planungsmethode etabliert, in welcher

die Not des realen (rückgekoppelten) Wasserfallmodells (nämlich dessen reale Nichtlinearität) zur Tugend gemacht wurde: Das Spiralmodell. Diese Vorgehensweise sei als nächstes näher betrachtet.

Das Spiralmodell

Die unerwünschten Rückkopplungen im Wasserfallmodell und die mittlerweile weit verbreiteten Werkzeuge zur relativ schnellen Gestaltung von Programmoberflächen haben dazu geführt, dass sich neben dem klassischen Wasserfallmodell noch ein weiteres etabliert hat: Das Spiralmodell. Die Grundgedanke ist, dass nicht die einzelnen Phasen jeweils beendet sein müssen, bevor die nächste Phasen begonnen wird, sondern dass die Phasen in kleinere Teilphasen zerlegt werden und diese zunächst begonnen werden. Es werden also die großen Phasen immer nur teilweise bearbeitet und danach gleich zur nächsten Phase übergegangen. Das Ergebnis ist ein mehrmaliges, stückweises Durchlaufen aller Phasen, bis ein gewisser Reifegrad erreicht ist. Betrachtet man z.B. nur die 4 Hauptphasen Analyse, Entwurf, Implementierung und Test, so könnte eine einfache Version des Spiralmodells wie in Abb. 2.2 skizziert aussehen.

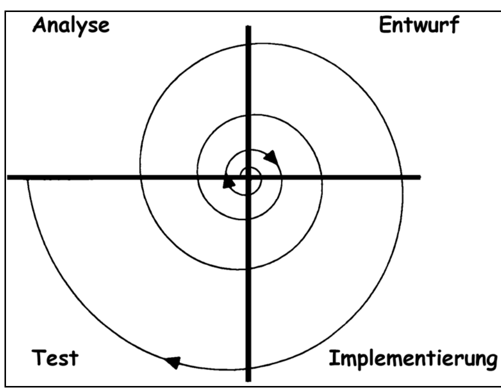


Abb. 2.2 Vereinfachtes Spiralmodell

Boehm hat diese Idee seiner verfeinerten Version des Spiralmodells zugrunde gelegt. Der Vorteil einer solchen Vorgehensweise liegt auf der Hand: Durch die sukzessive Entwicklung des Systems bekommt der Anwender schon relativ früh Teile des zukünftigen Programms zu sehen und es können so Fehler auch früher erkannt und damit rechtzeitig korrigiert werden. Die ersten Programmteile, welche durch die ersten, inneren Spiralläufe im Quadranten der Implementierung repräsentiert werden, nennt man auch (Implementierungs-)*Prototypen*. Sie stel-

len eine Art Programmhülle dar, wobei hier hauptsächlich nur die Benutzeroberflächen realisiert sind ohne wirkliche Anwendungen oder Funktionen hinter den Schaltflächen.

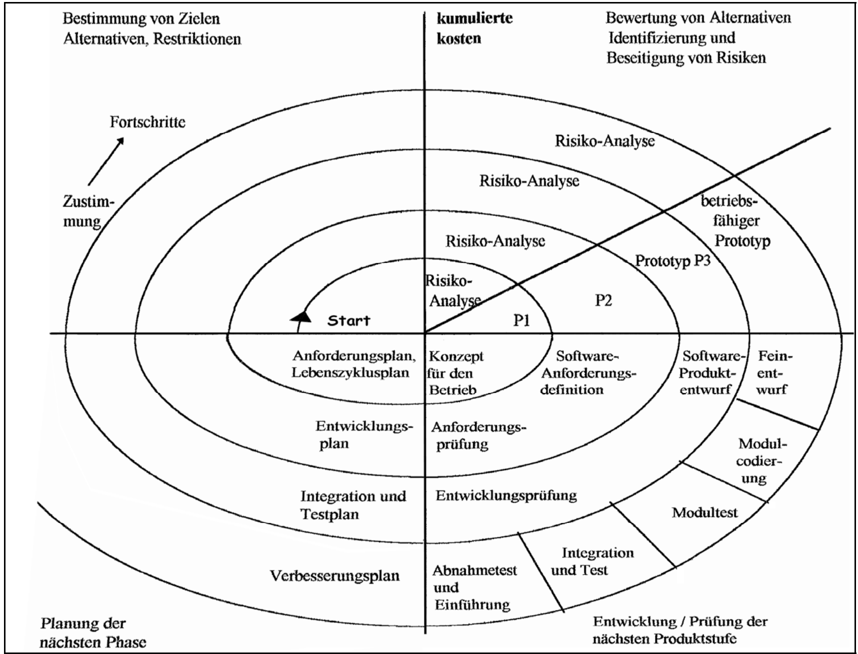


Abb. 2.3 Spiralmodell nach Boehm

Eine alte Faustregel besagt, dass man für die endgültige Programmierung des Systems alle Prototypen am besten wegschmeißt und den Zielzustand neu programmiert. Diese etwas radikale Methode muss aber nicht immer angewendet werden, denn häufig liefert das Prototyping recht brauchbare Programmmodule, die zumindest teilweise wiederverwendet werden können.
Das Spiralmodell birgt allerdings auch Nachteile. Insbesondere kann es dazu führen, dass der Gesamtaufwand des Entwicklungsprojekts stark in die Höhe geht. Festpreise sind bei so einer Vorgehensweise daher sehr gefährlich für den Entwickler. Leicht kann es z.B. dazu kommen, dass der Anwender, wenn er die ersten Versionen der Prototypen zu sehen bekommt, in Euphorie ausbricht und etliche Zusatzwünsche äußert, die so zunächst im Budget gar nicht vorgesehen waren. Die Entwicklungsspirale kann so ins unermessliche wachsen, das Programm kommt nicht in der geplanten Zeit zum Abschluss. In der Praxis erweist

es sich daher häufig als sinnvoll, eine Kombination des klassischen Wasserfallmodells mit dem Spiralmodell zu praktizieren. Die groben Planungen und das Gerüst der Entwicklung sollten mit dem Wasserfallmodell gemacht werden, während die Details dann nach dem Spiralmodell realisiert werden können.
Nun gibt es natürlich noch viele andere Phasenmodelle, es seinen hier z.B., das V-Modell[2] (und seine Erweiterung V-Model XP), das Ontogenese-Modell[3] oder das Diamant-Modell[4] genannt. Dort finden sich Parallelen zum Wasserfallmodell, wobei allerdings die Schwerpunkte z.T. andere sind.

Bevor wir zur praktischen Umsetzung der beschrieben Phasenmodelle kommen, wird im nächsten Kapitel auf ein in jüngster Zeit immer mehr an Bedeutung gewinnendes Thema eingegangen: Software-Architekturen.

Übungen zum Selbsttest

1. In welcher Phase des Wasserfallmodells sind folgende Sachverhalte angesiedelt:
   a) Angebotserstellung
   b) Projektplan
   c) Semantisches Datenmodell
   d) ER-Diagramm
   e) Anwendertest

[2] http://h90761.serverkompetenz.net/v-modell-xt/Release-1.1/Dokumentation/html
[3] Zöller-Greer, P.: *Softwareengineering für Ingenieure und Informatiker*, Vieweg 2002, S. 28
[4] ibid., Seite 16ff.

3. Software-Architekturen

In diesem Abschnitt sollen einige grundlegende Architekturen und Verfahren, die in neuerer Zeit eine immer wichtigere Rolle spielen, übersichtsartig beschrieben werden. Wir beschränken uns dabei nur auf 2 Repräsentanten, eine „wirkliche" Architektur (SOA) und die Anwendung einer Architektur (MDA) [5]. Wir folgen zunächst inhaltlich u.a. den Ausführungen des Fraunhofer Instituts für Software Engineering (IESE)[6], welches im Rahmen des vom Bundesministerium für Bildung und Forschung (BMBF) geförderten ViSEK-Projekts entsprechende Untersuchung angestellt und „Quasi-Normen" definiert hat. Das IESE legt fest:

Definition 3.1 (**Software Architektur**):
Unter einer *Software-Architektur* verstehen wir eine Spezifikation über die Teile des zu entwickelnden Systems, welche ihre Konnektoren sowie die Regeln für die Interaktion dieser Konnektoren mit den Systemteilen enthält. Ein Konnektor umfasst dabei z.B. alle Schnittstellen (auch auf verschiedenen Abstraktionsebenen).

Service Oriented Architecture (SOA)
Die Service-orientierte Architektur (Service-Oriented Architecture, SOA) ist ein Architekturkonzept, welches sich im Wesentlichen aus Diensten zusammensetzt. Dabei ist die Architektur eines Softwaresystems durch eine Konfiguration von Komponenten und Konnektoren beschreibbar. Eine bestimmte Konfiguration bildet dabei eine konkrete Architektur. Werden diese Vorstellungen auf das Konzept der SOA überführt, so entsprechen die Komponenten einer (dienstorientierten) Architektur den Diensten, die Konnektoren den verschiedenen möglichen Interaktionen zwischen Diensten.
Das Konzept der Service-Oriented Architecture sieht im Wesentlichen drei Komponenten vor (siehe Abbildung 3.1):

- *Service Providers*
- *Service Registries*
- *Service Requestors*

[5] Für eine ausführliche Behandlung siehe z.B. Zöller-Greer, P.: *Software-Architekturen: Grundlagen und Anwendungen*, Verlag composia, 2010
[6] http://www.software-kompetenz.de, auch nachfolgende Bildquellen

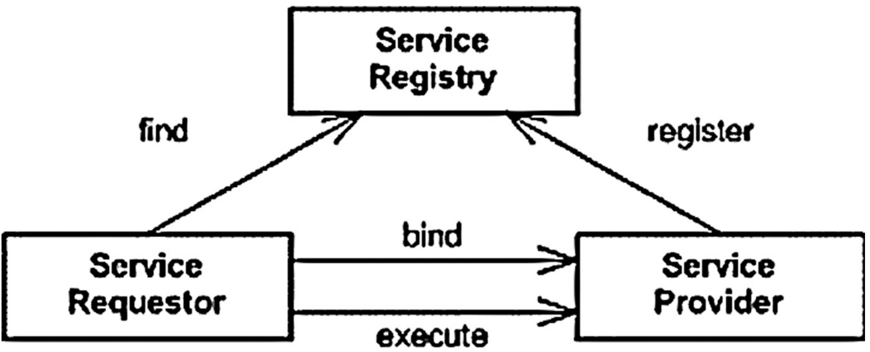


Abb. 3.1 SOA

Die im SOA-Konzept vorhandenen *Basisinteraktionen* entsprechen auf der Konzeptebene den Konnektoren einer Service-orientierten Architektur:

- Registrierung eines Dienstes (*register*, RegisterConnector)
- Suchen und Finden eines Dienstes (*find*, FindConnector)
- Herstellen der Bindung zu einem Dienst (*bind*, BindConnector)
- Stellen einer Anfrage an einen Dienst (*execute*, ExecuteConnector)

Eine der wichtigsten Eigenschaften von SOAs ist, dass sie weitgehend transparent sind. In SOAs sind immer die Realisierungen von Diensten strikt von ihren Beschreibungen getrennt ist. Konkret bedeutet dies, dass zu jedem Dienst eine separate Schnittstelle (service interface) existiert, die den Dienst eindeutig beschreiben kann. Unter Zuhilfenahme von plattformunabhängigen Beschreibungstechniken wie der Web Service Description Language (WSDL), kann somit nicht nur der Dienst, sondern sogar die zur Realisierung verwendete Plattform abstrahiert werden. Die Realisierung der einzelnen Dienste ist somit transparent und das Gesamtsystem heterogen und interoperabel.
Es besteht beispielsweise die Möglichkeit, Dienste in Java, *CORBA* oder Microsoft .NET zu realisieren. Sofern zur Beschreibung WSDL (oder etwas Äquivalentes) verwendet wird, ist es möglich, die Dienste auf anderen Basismaschinen zu nutzen. Dies zieht sich von der Nutzung eines Java-Web Services in einer .NET-basierten Anwendung bis hin zu komplexen Einsatzszenarien im Bereich der *EAI (Enterprise Application Integration)*.
Wie wir in Abb. 3.1 sahen, besteht die Service-Oriented Architecture aus den drei Komponenten Service Provider, Service Registries und Service Requestoren. Diese Komponenten werden nachfolgend genauer beschrieben.

Service Provider

Service Provider entsprechen den Dienstleistern. Ein Dienstleister kann entweder eine Softwarekomponente oder ein Dienst-anbietendes Unternehmen sein. Dies hängt hauptsächlich von der aktuellen Sicht auf die Architektur ab. Ein Service Provider stellt damit eine Funktionalität oder eine Ressource und eine passende Beschreibung zur Verfügung. Diese Beschreibung entspricht einem Vertrag. Dieser beschreibt die Interaktion mit dem Dienst (Schnittstellenbeschreibung) und kann weitere Informationen enthalten, die nicht an die konkrete Funktionalität gebunden sind (sog. „nichtfunktionale Aspekte").

Service Registry

Die Registry stellt eine zentrale Registrierung für die angebotenen Dienste dar, während die *Service Registry* eine dezentrale Registrierung beschreibt. Service Provider registrieren ihre Dienste bei einer Registrierung mithilfe eines Vertrags (service interface). Eine Registrierung kann dabei unterschiedlich ausgelegt sein: es gibt z.B. verzeichnisartige Registrierungen wie UDDI, welche gerade im Bereich der Web Services weit verbreitet sind (UDDI steht für Universal Description, Discovery and Integration; eine Organisation, die Webdienste anbieten möchte, kann eine UDDI –Registrierung, auch Business Registration genant, erstellen, welche aus einem XML-Dokument besteht, dessen Format in einem von UDDI definierten Schema festgelegt ist). Eine solche Registrierung ist passiv, sie kann nur Aufträge von Service Providern entgegennehmen (registrieren, updaten, löschen eines Dienstes) oder Anfragen von potenziellen Clients (Suche nach Dienstangebot) entgegennehmen.
Aktive Registrierungen stellen demgegenüber eine Möglichkeit zur Verfügung, weitere komplexe Aufgaben wahrnehmen können. Zu nennen sind hier Konzepte, wie sie beispielsweise in ODP (Open Distributed Processing) mithilfe von Tradern beschrieben werden. Weitere Aufgaben der Registrierungen sind Kategorisierung und Katalogisierung sowie Priorisierung und Ranking.

Service Requestor/Client

Service Requestoren sind mit Clients üblicher Client/Server-Architekturen vergleichbar. Sie kennen nur Registrierungen, an die sie Suchanfragen nach bestimmten Dienstleistungen stellen können. Die Dienstvermittlung erfolgt dann auf der Basis der registrierten Verträge. Analog zu Client/Server-Architekturen ist die Diensterbringung für den Client transparent und hinter einer Schnittstelle versteckt.

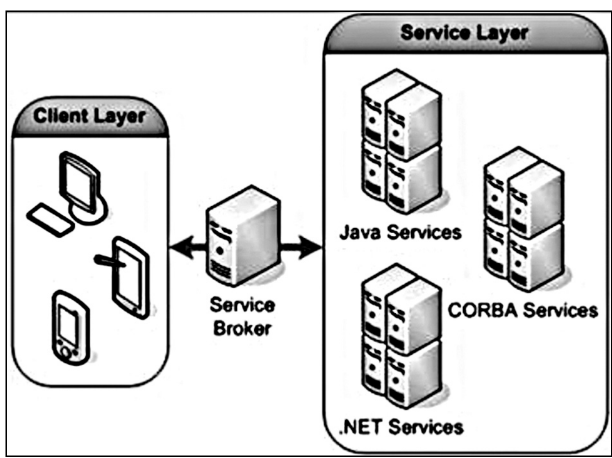


Abb. 3.2 Servicenutzung unter SOA

Die Komponenten einer SOA können nun verwendet werden, um eine dienstorientierte Architektur in drei Teilsysteme zu gruppieren:

1. Dienstnutzung (Client Layer)
2. Dienstvermittlung (Service Broker)
3. Diensterbringung (Service Layer)

Gezeigt ist hier ein heterogenes Szenario, das einige Besonderheiten der SOA verdeutlicht. Dienste können nahezu beliebig realisiert sein (rechter Teil der Abbildung). Sie müssen lediglich der Anforderung genügen, ihre Funktionalität über eine normierte Schnittstelle, z.B. eine WSDL-Schnittstelle, anzubieten.
Eine Registry (Service Broker, Mitte der Abbildung 3.2) kann administrative Aufgaben wahrnehmen (Katalogisierung, Kategorisierung) und, weitaus wichtiger, die Funktion eines Vermittlers ausüben. Der Service Broker ermöglicht sowohl Plattform-, als auch Ortstransparenz. Diese „Indirektionsstufe" ist gerade für den Dienstnutzer interessant, da sie es ihm ermöglicht, beliebige Dienste an beliebigen Orten zu nutzen. SOAs werden deshalb meistens als lose gekoppelte Architekturen betrachtet.
SOA erfährt derzeit gerade durch den vermehrten Einsatz von Internetapplikationen eine immer größere Relevanz.

Model Driven Architecture (MDA)
Die Model Driven Architecture ist von der Object Management Group (OMG) eingeführt und wird als nächster Schritt zur Automation der Erstellung von Software und somit in Richtung einer ingenieurmäßigen Softwareentwicklung aufgefasst. Genau genommen handelt es sich bei MDA um keine Architektur im Sinne unserer Definition, sonder eher um eine Anwendung davon.
Um MDA besser zu verstehen, sollen zunächst einige kurze Ausführung zur Modellbildung allgemein im Software-Engineering folgen.
Die Modellbildung ist ein Abstraktionsprozess in der Informatik und der Mathematik, mit dem man versucht, die Komplexität eines Systems auf ein „bewältigbares" Maß zu reduzieren. Im Modell sollen dabei die entscheidenden Eigenschaften und Funktionalitäten des Systems erhalten bleiben. Dabei kommen üblicherweise zwei Varianten in Betracht (siehe auch Kapitel 4 und 5).

Funktionale Dekomposition

Bei der funktionalen Dekomposition werden die einzelnen Funktionen des Systems in immer feingranularere Einheiten unterteilt, bis diese schließlich durch entsprechende Verhaltensdiagramme in ihrem Ablauf modelliert werden können. Typische Strukturmodellierungskonstrukte sind z.B. sog. "State-Charts", typische Ablaufmodellierungskontrukture sog. "Activity-Charts", welche auf Zustandsautomaten basieren, die das reaktive Verhalten des Systems bzw. einer Komponente darstellen. Dabei lassen sich Komponenten hierarchisch oder parallel zu übergeordneten Komponenten bis hin zu einem kompletten System zusammensetzen.

Objektorientierte Modellierung

Für eine objektorientierte Modellierung können z.B. UML-Diagramme benutzt werden. In diesem Ansatz stehen verschiedene Diagrammtypen zur Verfügung, durch die jeweils ein Teil des Gesamtmodells modelliert werden kann. Man unterscheidet

- **Anwendungsfalldiagramme**
  Dienen der einfachen Darstellung von Prozessen. In diesem Diagrammtyp werden die Anwendungsfälle, ihre Beziehungen untereinander oder mit konkreten Personen, Ereignissen oder Prozessen beschrieben.
- **Klassendiagramme**
  Beschreiben die statische Struktur eines Objektmodells und enthalten die Attribute wie die Beziehungen von Mengen einer Klasse von Objekten.

- **Verhaltensdiagramme**
  Enthalten im Gegensatz zu den Klassendiagrammen die dynamischen Vorgänge innerhalb des objektorientierten Modells. Hier wird wieder weiter in verschiedene Diagramme, wie Sequenz- , Kollaborations- , Zustands- und Aktivitätsdiagramme untergliedert, je nach den betrachteten Modellelementen (z.B. Objekte oder Zustände) und dem Abstraktionsgrad.
- **Implementierungsdiagramme**
  Bieten die Möglichkeit für die Unterstützung objektorientierter Modelle in Programmcode. Hier wird zwischen Komponenten- und Verteilungsdiagrammen differenziert.

Nach diesen Erläuterungen zurück zu MDA.
MDA ist eine Strategie zur modellgetriebenen (im obigen Sinne) und generativen (d.h. automatisch generierten) Soft- und Hardwareentwicklung. Durch automatische Generierung von Quellcode aus den Modellen wird der Automatisierungsgrad der Entwicklung erhöht (und somit Fehlerquellen minimiert). MDA etabliert sich zunehmend als ein Standard, der Software schneller, effizienter, kostengünstiger und qualitativ hochwertiger zu erstellen vermag.
Der Kern der MDA besteht in der Trennung von fachlichen und technischen Teilen eines Softwaresystems in Form von plattformunabhängigen Modellen (PIMs) und plattformspezifischen Modellen (PSMs), die beide unabhängig voneinander wieder verwendet werden können.
In den PIMs wird das fachliche Wissen (Fachlogik) des Softwaresystems/Anwendung, wie z. B. Geschäftsprozesse oder Fachverfahren, technologieunabhängig erfasst und modelliert. Im Gegensatz dazu wird in den plattformspezifischen Modellen die Implementierungstechnologie definiert, d. h. die technischen Aspekte bezogen auf eine konkrete Plattform. Hierbei kommen häufig Frameworks zum Einsatz. Die explizite Trennung der Fachlogik von der Implementierungstechnologie erlaubt eine leichtere Wiederverwendung der Fachlogik, z. B. im Rahmen einer Migration auf eine neue Plattform.
Ziel der MDA ist unter anderem, die Lücke zwischen Modellen (bzw. Modellierern) und Quelltext (bzw. Programmierern) zu schließen. Im allgemeinen bewirkt ja jede Änderung im Entwicklungsprozess auch Änderungen auf den einzelnen Abstraktionsebenen. Grundsätzlich gilt die Faustregel, dass je höher der Abstraktionsgrad, desto invarianter ist er gegenüber Änderungen tieferliegender Ebenen. Zudem sollen sich die abstrakteren Modelle (PIM) durch Mapping (zumindest teilweise) automatisch in weniger abstrakte Modelle (PSM) überführen lassen. Dies scheint mit MDA in scheinbare Nähe gerückt. Eine Standardisierung in diesem Bereich verspricht stabilere Softwaren und kürzere, besser kalkulierbare Entwicklungszeiten.

Motivation und Nutzen
Man kann feststellen, dass die mit Software zu lösenden Probleme im Prinzip immer die gleichen sind, nur die Technologien ändern sich. Früher wurde z.B. ein Bestellsystem mit COBOL, später in VBA und heute in Java programmiert. Doch die wichtigsten Anforderungen an solche Systeme und auch die Funktionen der Software blieben über Jahrzehnte die gleichen (von einigen Details wie Netzwerkfähigkeit etc. abgesehen). Dies legt nahe, eine Architektur zu finden, die sich ebenfalls nicht verändert und sogar die neuen Gegebenheiten integriert. MDA ist so ein Versuch dies zu bewerkstelligen.
Man hofft neben der Reduzierung der Risiken und Schwächen der traditionellen Softwareentwicklung folgende Problemfelder besser in den Griff zu bekommen:

- Aufwand/Kosten
- Fehleranfälligkeit und Code-Qualität
- Konsistenz zwischen Code und Modellen
- Integration und Interoperabilität
- Portabilität
- Wartung.

MDA versucht dies durch plattformunabhängige Modelle zu realisieren, so dass dadurch entstehende Konsistenzprobleme zwischen Modellen und Quellcodes vermieden werden. Das Problem ist nämlich, dass zwar häufig Systeme z.B. in UML beschrieben werden, doch bei späteren Programmanpassungen werden diese i.d.R nicht nachgeführt. Somit werden diese später auch nicht mehr genutzt und deren Vorteil ist dahin.
MDA integriert auch etablierte Standards wie UML und andere.

Konzepte und Basistechnologien
Die OMG veröffentlicht in ihrem MDA-Guide[7] die grundlegenden Konzepte und Technologien. Wir folgenden im wesentlichen diesen Ausführungen.
Es gibt 4 wichtige Konzepte in MDA:

- **Computation Independent Model (CIM)**
  Das CIM repräsentiert die Geschäfts- oder Domänensicht des zu entwickelnden Softwaresystems und ist der Modelltyp mit der höchsten Abstraktionsebene.

- **Platform Independent Model (PIM)**
  Mit Hilfe des PIMs wird die Struktur und das Verhalten des zu entwickelnden Softwaresystems spezifiziert. Dies erfolgt unabhängig von

[7] siehe http://www.omg.org/docs/omg/03-06-01.pdf

den technischen Aspekten (Software- und Hardwareplattformen) der Implementierung.

- **Platform Specific Model (PSM)**
  Das PSM erweitert das PIM um die konkreten technischen Details, die für die Implementierung des Softwaresystems in der jeweiligen Zielplattform notwendig sind.

- **Modell-Mapping**
  Mit Hilfe so genannter Mappings (Abbildungen) wird das PIM gegenüber einer konkreten Zielplattform basierend auf den Plattformprofilen in ein entsprechendes PSM übersetzt.

MDA unterstützt folgende Basistechnologien:

- **UML**
  Die Unified Modeling Language (UML, vgl. Kapitel 5) ist integraler Bestandteil der MDA-Konzepte. In der aktuellen Version UML 2.0 wird durch die Erweiterung hinsichtlich Metamodellierung und Profilmechanismus der MDA-Ansatz unterstützt. Die UML dient der (graphischen) Beschreibung von Modellen, die je nach Modellart z. B. die Struktur oder das Verhalten festlegen. Hierzu werden (bzw. wurden bereits) diverse Erweiterungen in Form von Plattformprofilen (wie z.B. UML/Realtime für Echtzeitanwendungen) konzipiert, um der UML immer mehr Plattformen zu erschließen.

- **UML-Profil**
  UML-Profile sind der Standardmechanismus zur Erweiterung des Sprachumfangs der UML, um sie an spezifische Einsatzbedingungen (z. B. fachliche oder technische Domänen) anzupassen.

- **MOF**
  Die Meta Object Facility (MOF) ist ein Standard der OMG zur Definition von Metamodell-Sprachen. Die MOF ist auf der Meta-Metamodellebene angesiedelt und definiert den Aufbau der Metamodellebene auf der z. B. UML und CWM liegen.

- **XMI**
  eXtensible Markup Language Metadata Interchange (XMI) ist ein XML-basiertes Austauschformat für UML Modelle, was die Interoperabilität von Modellen zwischen verschiedenen Werkzeugen unterschiedlicher Hersteller ermöglicht. So lassen sich mittels der MDA die Modelle zwischen verschiedenen Werkzeugen und Werkzeugkategorien unterschiedlicher Hersteller austauschen.

- **CWM**
  Das Common Warehouse Metamodel (CWM) ist ein von der OMG entwickeltes und standardisiertes Referenzmodell für den formalen und herstellerunabhängigen Zugriff und Austausch von Metadaten in der Domäne Data Warehousing.

Die OMG legt in ihrem grundlegenden MDA-Guide viel Wert auf das Mapping von PIM nach PSM. Dies ist in der Tat auch der wichtigste Aspekt der ganzen Angelegenheit: Hat man nämlich z.B. ein UML-Klassendiagramm (innerhalb des PIM), so ist das Problem gerade, daraus „automatisch“ eine lauffähige Software-Anwendung zu erzeugen (welche alle Angaben des konkreten PSM benötigt).

Es muss also möglich sein, dass für ein vorhandenes PIM als „Input“ ein konkretes PSM als „Output“ geliefert werden kann, wobei natürlich die plattfomkonkreten Parameter (Betriebssystem, Hardware etc.) ebenfalls mitgegebenen werden müssen. Dies soll nachfolgend näher spezifiziert werden.

Mapping PIM → PSM

Grundsätzlich unterscheidet die OMG verschiedene Formen des Mappings. So gibt es das „Type Mapping“, welches ein Mapping zwischen Metamodellen (wie MOFs) ermöglicht, und dem „Instance Mapping“, wo konkrete Elemente eines PIM in konkrete Elemente eines PSM umgesetzt werden. Über sogenannte „Marks“ (Markierungen) werden diese Elemente ausgewählt. Marks können allgemein beschrieben und wiederverwendet werden. Beispielsweise kann die Mark „Entity“ ein PSM-Element sein, was allgemein Klassen aus dem PIM zugeordnet wird. Die Marks werden also dazu benutzt, geeignete Information für das Mapping zur Verfügung zu stellen, wie nachfolgende Abbildung zeigt.

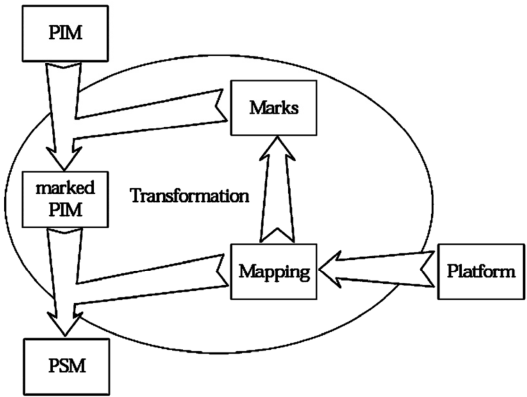


Abb. 3.3 Mapping mit Marks

Wie man in Abb. 3.3 erkennt, fließen also die Marks in die Mapping-Beschreibung ein, sodass das PIM zu einem „Marked PIM“ wird. Zusammen mit der Plattform-Information kann daraus das PSM erzeugt werden (und daraus schließlich der Code für die Anwendung). Nun sind Transformationsprozesse nicht nur auf diese Ebene beschränkt. Man kann weiter abstrahieren und eine „Meta-Ebene“ einführen, wo die jeweiligen Entsprechungen zu finden sind. Diese sind auf der Meta-Ebene in Abb. 3.4 zu finden.

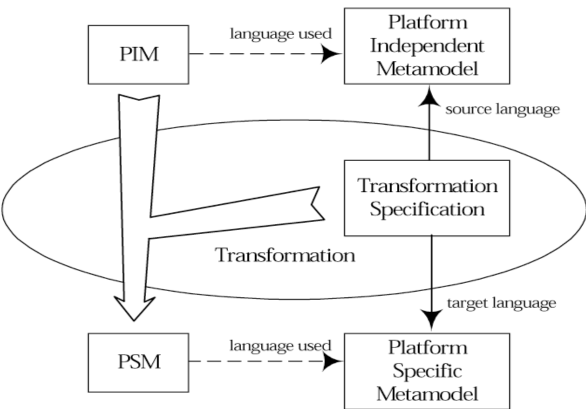


Abb. 3.4 Meta-Modell-Mapping

Man erkennt in Abb. 3.4, dass jetzt Angaben über die benutzen Sprachen einfließen, was dazu führt, dass die allg. Quellsprache in eine konkrete Zielsprache überführt werden muss.

Diese Transformationen geschehen über bestimme Beschreibungssprachen innerhalb eins MOFs. Nun gibt es natürlich auch entsprechende Transformationen für die Datentypen (Abb. 3.5).

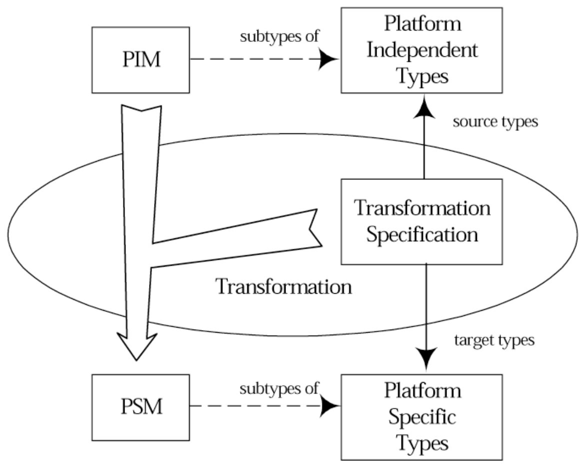


Abb. 3.5 Modell-Mapping

Hier werden jetzt Objekttypen einander zugeordnet und entsprechend transformiert. Dabei müssen diese auf Instanzenebene genau beschrieben sein, damit in dem Transformationsschema die jeweilige Überführung gewährleistet ist. Ähnlich wie bei den Sprachen geschieht dies auch hier so, dass sowohl PIM als auch PSM gewisse Subtypes ihren zugeordneten Plattformspezifikationen zur Verfügung stellen. Über die Transformationsspezifikation können dann die plattformunabhängigen Objektdatentypen konkreten plattformabhängigen zugewiesen werden.

Natürlich ist es sinnvoll, wenn gewissen Pattern (z.B. Entwurfsmuster, vgl. Kapitel 6) in den Transformationsprozess einfließen. Die geschieht dann wie in Abb 3.6 gezeigt:

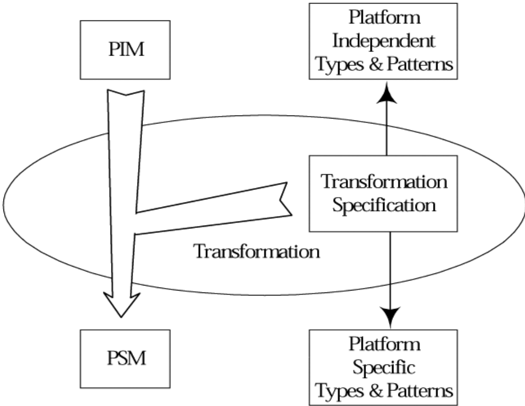


Abb. 3.6 Mapping mit Pattern

Grundsätzlich können Pattern auch als Marks mit eingebunden werden, sodass hier wie in Abb. 3.3 vorgegangen werden kann.

Werkzeuge
Viele Hersteller nehmen für ihre Werkzeuge in Anspruch, MDA-kompatibel zu sein. Jedoch werden die MDA-Konzepte der OMG gegenwärtig von sehr wenigen Werkzeugen überzeugend unterstützt. Insbesondere die explizite Darstellung des PSM in den Werkzeugen erfolgt nur selten. Der Großteil der gegenwärtig am Markt verfügbaren MDA-Werkzeuge verstecken das PSM hinter Einstellungen und Erweiterungen des PIMs.

Die OMG stellt auf ihrer Website eine umfassende Aufstellung von Firmen und ihren Produkten[8] im Umfeld der MDA zur Verfügung. Diese Liste beinhaltet neben bekannten Softwareherstellern und Open-Source-Produkten auch Unternehmen, die weniger mit MDA-Werkzeugen zu tun haben, sondern vielmehr Dienstleistungen rund um das Thema MDA anbieten.

Zur Zeit sind MDA-Werkzeuge wie „ArcStyler“ (von Interactive Objects Software GmbH) oder das Plug-In „Codagen Architect“ (für bestehende UML-Werkzeuge wie RationalRose etc.) relativ gut einsetzbar.

[8] unter http://www.omg.org/mda/committed-products.htm

Nachfolgend drei Abbildungen, die die Vorgehensweise in ArcStyler 5.5 demonstrieren[9]:

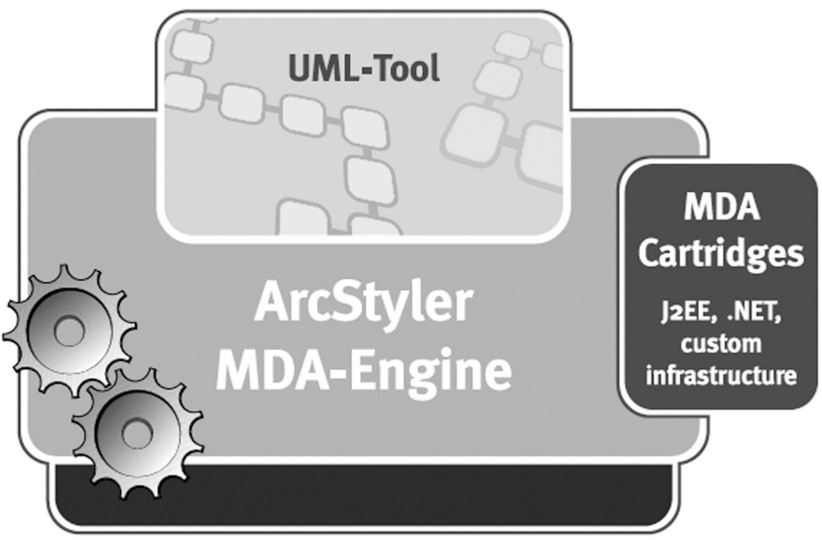


Abb. 3.7 ArcStyler Komponenten

ArcStyler besitzt ein UML-Tool zum Erstellen der UML-Klassendiagramme im PIM. Über auswählbare MDA-Cartridges werden die transformationsspezifischen Angaben für das PSM gemacht. Die MDA-Engine schließlich führt das Mapping durch und erzeugt auch den Quellcode der Applikation.

Abb. 3.8 gibt einen Einblick wie ein UML-Klassendiagramm mit dem ArcSyler erzeugt werden kann. Auch wenn aufgrund der technischen Verkleinerung hier keine Details zu erkennen sind, so sieht man immerhin, dass gewisse Zeichenwerkzeuge verfügbar sind, mit deren Hilfe die Assoziationen zwischen den Klassen hergestellt werden können. Es kann außerdem auf die Instanzenebenen gegangen werden und die jeweiligen Links dazu sind sichtbar.

Schließlich werden die plattformspezifischen Angaben als Marks vorgenommen (Abb. 3.9), so dass das Mapping von PIM nach PSM durchgeführt werden kann.

[9] Quelle: http://www.interactive-objects.com/

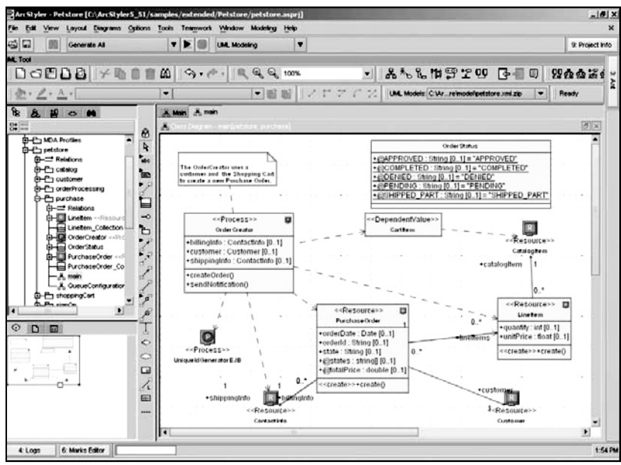


Abb. 3.8 UML-Diagramm-Erstellung im ArcStyler

Experten sind sich allerdings darüber einig, dass nach dem heutigen Stand der Technik allenfalls 70% - 85% des automatisch generierten Codes wirklich das Problem abdecken, d.h. es muss noch immer „von Hand" hier eingegriffen werden, um eine wirklich den Kundenwünschen gemäße Applikation zu erzeugen. Doch es gibt noch andere Problemfelder.

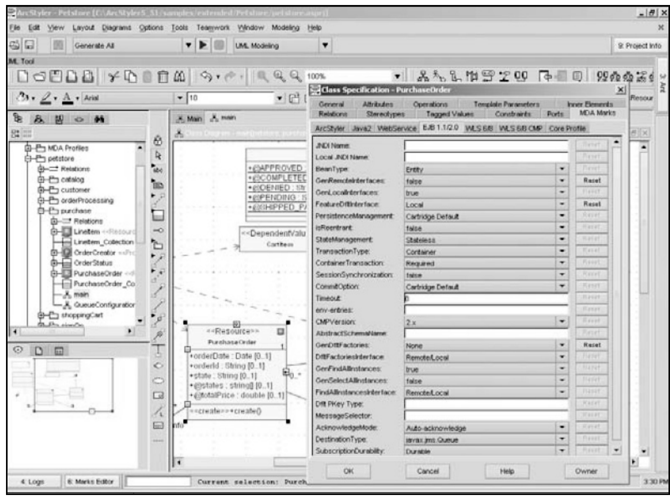


Abb. 3.9 Marks im ArcStyler

Praktische Problemfelder

Was in MDA theoretisch recht gut klingt, birgt in der Praxis aber noch einige Probleme. Dazu zählen laut IESE unter anderem:

- Stabilität des Datenmodells
  Änderungen am Datenmodell können großen Aufwand nach sich ziehen, der bei der Änderung des Modells nicht sofort sichtbar ist.

- Entfremdung des Entwicklers vom Code
  Der generierte Code ist für den Entwickler nur noch schwer verständlich und damit schwer nachvollziehbar.

- Unzulänglichkeiten in den Tools
  Die Erwartungshaltungen der Entwickler sind derzeit nur schwer zu erfüllen.

- Reverse-Engineering/Reengineering/Wartung
  MDA ist ursprünglich für Neuentwicklungen gedacht. Wie können aber bestehende Softwaresyteme in MDA übernommen werden, so dass die Weiterentwicklung mit Hilfe dieser Methodik erfolgen kann? Ein weiterer Aspekt ist die Erzeugung von Releases, Patches oder Updates, welches ein wichtiger Bestandteil der laufenden Wartung von Anwendungen ist. Wie geht MDA damit um? Jedes Mal das komplette System neu installieren?

- Plattform-Begriff
  Die OMG versteht im engeren Sinne unter diesem Begriff die Middleware und hier insbesondere das ebenfalls von der OMG entwickelte Komponentenmodell CORBA, welches einen Standard darstellt zur Entwicklung von serverseitigen, skalierbaren, sprachunabhängigen, transaktionalen Mehrbenutzersystemen. Plattformunabhängigkeit meint also eher Unabhängigkeit von Middleware, im weiteren Sinne aber auch z. B. Programmiersprachen. Letztlich muss sich der MDA-Anwender im Vorfeld darüber klar werden, wie er den Begriff Plattform definiert.

- ROI (Return-on-Investment)
  Bisher gibt es nur sehr wenig Überlegungen dazu, ab wann sich der Einsatz von MDA lohnt. Für den Einsatz von MDA sind viele Vorbereitungen und Vorarbeiten notwendig. Hierzu gehören nicht nur Schu-

lungen der Entwickler, sondern auch konzeptionelle Arbeiten (z. B. Anpassung des Softwareentwicklungsprozesses). Dies sind nicht unerhebliche Aufwände.

- EAI (Enterprise Application Integration)
  Modellierung auf hoher Ebene erscheint auf den ersten Blick sinnvoll. Neue Anwendungen laufen aber selten für sich isoliert, sondern müssen in eine vorhandene Anwendungslandschaft integriert werden. Wie aber werden sie in der MDA-basierten Entwicklung berücksichtigt?

- Herstellerabhängigkeit
  Da der MDA-Standard derzeit noch nicht vollständig und einheitlich von den Toolherstellern umgesetzt wird, bedeutet die Festlegung auf ein Werkzeug damit auch eine gewisse Abhängigkeit vom Hersteller. Solche Abhängigkeiten haben sich in der Vergangenheit jedoch oftmals als schlecht herausgestellt.

Grundsätzlich kann jedoch gesagt werden, dass MDA richtungsweisend ist und in Zukunft das Software-Engineering völlig weg vom Programmieren kommen wird. Die Planung und Modellierung in MDA wird dann ausreichen, um den Programmcode (und damit die eigentliche Applikation) automatisch generieren zu lassen.

Übungen zum Selbsttest

1. Wozu dienen Software-Architekturen?

2. Was ist das Ziel von MDA?

3. Was versteht man in MDA unter „Mappng“?

4. Projektplanung und Pflichtenheft

Ist ein Softwareprojekt einmal gestartet, so muss man sich überlegen, mit welchen Werkzeugen man es durchführen will. Dabei ist immer noch die Planungsphase gemeint. Man muss sich überlegen, ob man modellgetrieben vorgehen will (MDA), ob man eine serviceorientierte Anwendung hat (SOA), ob man nach dem Wasserfallmodel, dem Spiralmodell, einer Kombination davon oder nach einem ganz anderen Modell vorgehen möchte und so weiter. Natürlich kann im Rahmen dieses Kompaktkurses nicht jede dieser Möglichkeiten durchgespielt werden. Wir beschränken uns daher auf die mehr oder weniger „klassische" Vorgehensweise, die einen guten Ausgangspunkt auch für die neuen Architektonischen Ansätze darstellt und zudem in der Industrie noch sehr verbreitet ist (und wohl auch noch eine Weile bleibt).
Nach dem klassischen Wasserfallmodell findet das hier nachfolgend Beschriebene in der Phase „DV-Konzept" statt (vgl. Abb. 2.1). Diese Phase folgt der offiziellen Auftragsvergabe, wo schon eine grobe Problembeschreibung erfolgte. Nun muss diese Problemanalyse verfeinert und ein Lösungskonzept erarbeitet werden. Am DV-Konzept sind daher sowohl Auftraggeber wie auch Systementwickler (Auftragnehmer) beteiligt. Das DV-Konzept muss in einer Sprache erfolgen, die von beiden verstanden wird. Daher ist das DV-Konzept meistens verbal formuliert, manchmal auch schon mit einfachen sog. U-Case- und/oder ER-Diagrammen angereichert (immer vorausgesetzt, dass auch der Auftraggeber diese verstehen kann). Wir werden daher in diesem Kapitel u.a. diese beiden grafischen Hilfsmittel zur Problem- und Ablaufbeschreibung vorstellen.
Das Ziel der Phase „DV-Konzept" ist also eine Spezifikation (das Pflichtenheft), mit der der Softwareentwickler dann danach ohne den Auftraggeber weiter zu Rate ziehen zu müssen (Idealfall!) selbstständig die weitere DV-Planung bis zum fertigen Programm erstellen kann. In der Praxis ist das natürlich weder realistisch noch gewollt. Es ist vielmehr so, dass versucht wird, das Pflichtenheft so zu erstellen, als sollte es den beschriebenen Anforderungen genügen, doch bei der weiteren Planung, dem Software-Design, soll nach dem Spiralmodell vorgegangen werden, so dass der zukünftige Anwender bereits frühzeitig die Entstehung der Software (u.a. durch Ausprobieren von Prototypen) und evtl. Planungsfehler oder falsche Funktionsabläufe korrigieren kann. Damit fließt natürlich auch die Grenze, wo die Software-Analyse endet und das Software-Design beginnt. Nach dem Spiralmodell wechseln sich diese Phase naturgemäß sogar andauernd ab. Auch die Literatur ist sich diesbezüglich nicht völlig einig. Die Erstellung des Pflichtenhefts jedenfalls würde ich vollständig der Software-Analyse zuordnen, während die Phasen DV-Entwurf und Implementierung im „spiralförmigen" Wechsel Teile aus der Analyse und dem Design enthalten.

Zunächst sei die Definition eines Pflichtenheftes gemäß DIN 69901 angegeben. Demnach muss ein Pflichtenheft folgendes enthalten (jedenfalls wenn es dieser DIN-Norm entsprechen will):

1. Unternehmenscharakteristik

1. Name und Adresse des Unternehmens
2. Branche, Produktgruppe, Dienstleistungen
3. Unternehmensstruktur, Zahl der Betriebsstätten, Niederlassungen
4. Unternehmensgröße, Wachstumsrate
5. Organisation und Datenverarbeitung
6. Weitere wesentliche Angaben zum Unternehmen

2. Istzustand der Arbeitsgebiete

1. Überblick und Zusammenhänge; evtl. Strukturorganisation
2. Bisherige Verfahren für die Arbeitsgebiete 1 ... n
3. Bisherige Hilfsmittel
4. Vorhandenes Fachwissen, Computer-Wissen, Organisationsniveau
5. Unternehmensspezifische Besonderheiten
6. Bewertung des Istzustandes - Aufwand und Nutzen,- Stärken und Schwächen

3. Zielsetzungen

1. Erwarteter quantifizierbarer Nutzen
2. Sonstige erwartete Vorteile

4. Anforderungen an die geplante Anwendungssoftware

1. Fachliche Anforderungen
    1. Überblick und Zusammenhänge
    2. Detaillierte Anforderungen an die Arbeitsgebiete 1 ... n
        - wesentliche Verfahren
        - wesentliche Ein- und Ausgabeinformationen
        - Verarbeitungsarten und -häufigkeit
        - unternehmensspezifische Besonderheiten
2. Technische Anforderungen
    1. Qualitätsanforderungen - Integration
        - Dateiorganisation
        - Zugriffsberechtigung, Datensicherheit und -rekonstruktion
        - Form der Programmauslieferung
    2. Dokumentation und Schulung

       [Differenzierung in funktional und nichtfunktional]

5. Mengengerüst

1. Kartei/Stammdaten
2. Bestandssätze, Belege / Zahl der Bewegungen
3. sonstige Mengen- und Häufigkeitsangaben

6. Anforderungen an Hardware und Systemsoftware

7. Mitarbeiter für die Umstellung

8. Zeitlicher Rahmen

In der Praxis ist es nur selten so, dass ein Pflichtenheft wirklich alle Anforderungen dieser DIN-Norm erfüllt. Das Deutsche Institut für Normierung (DIN), welches diese Empfehlungen herausbringt, ist natürlich um möglichst maximale Vollständigkeit und Allgemeinheit bemüht. Wenn nun aber ein Unternehmen beispielsweise einen Softwareentwickler mit der Erstellung eines Adressprogramms beauftragt, ist sicher z.B. die Wachstumsrate des Unternehmen (Punkt 1.4) von wenig Relevanz.
Uns geht es hier im Rahmen des Software-Engineerings ohnehin mehr um die DV-relevanten Dinge. Deswegen werden wir uns auf die dafür wichtigen Punkte beschränken.
Die Phase DV-Konzept (manchmal auch Fachkonzept oder Grobkonzept genannt) wird, wie schon angedeutet, zumindest teilweise von dem Systemanalytiker zusammen mit dem Kunden durchgeführt. Der Kunde beschreibt dabei die Ausgangsverhältnisse (manchmal vorab in einem sog. Lastenheft) und was das gewünschte Programm später können soll. Dem Systemanalytiker kommt u.a. die Aufgabe zu, die „richtige“ Information aus dem Kunden herauszuholen. Meistens kennt der Systemanalytiker das Problemfeld des Kunden kaum, während der Kunde normaler Weise nicht weiß, welche Information für die genaue Systemanalyse wichtig ist. DV-Fachmann und Kunde müssen daher einen gemeinsamen Ansatzpunkt finden. Grob unterteilt besteht ein Fachkonzept aus zwei Teilen, nämlich der *Anforderungsanalyse*, wo der Ist- und Sollzustand von Hard- und Software erarbeitet wird, und der *Systemanalyse*, wo eine erste semantische Datenmodellierung vorgenommen wird. Letzte beschreibt, was die Anwendung leisten soll, wie sie prinzipiell aufgebaut sein muss und wie das erreicht werden kann. Das Ergebnis ist dann das Papier, welches Pflichtenheft oder allgemeiner einfach nur Spezifikation genannt wird.
Im DV-Konzept wird damit also im Prinzip der Weg vom Ist-Zustand zu dem Soll-Zustand in einer für Kunde und Analytiker gleichermaßen verständlichen Sprache beschrieben. Daher ist das Pflichtenheft überwiegend verbal abgefasst und enthält wenige Abstraktionen. Dies birgt jedoch die Gefahr, dass hier Inkon-

sistenzen entstehen können. Das heißt, es könnte z.B. auf Seite 10 des Pflichtenheftes etwas stehen, was einer Anforderung auf Seite 25 direkt widerspricht. Solche Widersprüche werden u.U. erst sehr spät bemerkt, im ungünstigsten Fall erst beim Testen der Software. Dieses führt dann zu den unerwünschten Rückkopplungen im Wasserfallmodell, denn es muss im Pflichtenheft eine Korrektur gemacht werden, die alle nachfolgenden Phasen betreffen kann. So etwas ist dann meistens auch relativ zeit- und kostenaufwendig.
Es sollte das Pflichtenheft daher sehr genau und umsichtig entworfen werden. Manchmal kann es hilfreich sein, hier sogar schon Elemente des DV-Entwurfs zu verwenden, sofern der Kunde in der Lage ist, dies zu verstehen.
Im Pflichtenheft sollten mindestens enthalten sein:

- (semantische) *Modellierung* der projektrelevanten Teilausschnitte der Realität
- Beschreibung des *Istzustandes*, ggf. unter Zuhilfenahme von Organigrammen, Funktionsabläufen und/oder bereits vorhandener Datenmodellen
- Beschreibung des *Sollzustandes*, ggf. unter Zuhilfenahme von Organigrammen, Funktionsabläufen und/oder bereits vorhandener Datenmodellen
- Beschreibung der *Ein- und Ausgaben* des zu entwickelnden Programms
- Beschreibung von *Datenstrukturen* (Länge und Art der Daten, z.B. Gleit- oder Festkommazahlen, wie lang können maximal die jeweils abzuspeichernden Zeichenketten sein)
- Beschreibung des *Datenaufkommens*, d.h. der Datenmengen, Datenherkünfte etc.
- Beschreibung des *Wegs* von der Dateneingabe zur Datenausgabe (macht beispielsweise dies im Istzustand bereits jemand „von Hand", so ist diese Person zu befragen und der genaue Funktionsablauf zu beschreiben; evtl. Berechnungsformeln etc. sind anzugeben)
- Beschreibung der geplanten *Datenmanipulationsmöglichkeiten*
- Angaben über die voraussichtlichen *Zugriffshäufigkeiten* und den Benutzerkreis (wer greift wie oft ggf. mit welchen anderen Benutzern gleichzeitig auf welche Datenbestände zu)
- Angaben zu *Sicherheitsaspekten* (ist z.B. ein nach Hierarchien abgestufter Zugriff einzelnen Benutzer erforderlich, dürfen manche Benutzer nur bestimmte Daten ansehen/ändern, Passwörter, Administratorzugänge etc.)
- Angaben zum *Datenschutz* (ist z.B. der Betriebsrat mit einzubeziehen)
- Beschreibung von möglichen *Validierungsprozeduren* (Planung von Validierungstests, wer testet wann was wie lange etc.)

- Detaillierter *Projektplan* (wer macht wann was), z.B. in der Form eines Balkendiagramms
- Art und Umfang evtl. *neu anzuschaffender Hardware/ Software/ Personal*
- Festlegung von *Review-Terminen* und evtl. Zahlungsbedingungen
- Festlegung von *Projektpersonal* und Verantwortlichkeiten auf Kundenseite

Im Hinblick auf eine möglichst effiziente Systementwicklung sollte das Pflichtenheft so ausführlich wie möglich sein. Hier zu sparen wäre ein großer Fehler (wegen der u.U. zu erwartenden Rückkopplungen). Diese Phase zusammen mit der nächsten Phase des Wasserfallmodells (DV-Entwurf) sollten mind. 70% der geplanten Entwicklungszeit des gesamten Projekts betragen. Was hier versäumt wird, kann ein Vielfaches an Zeit bei der Implementierung kosten.
Das fertig gestellte Pflichtenheft sollte am Besten sowohl vom Kunden als auch vom Systemanalytiker unterschrieben und damit von beiden Seiten als verbindlich anerkannt werden. Sollte es später zu einem Rechtsstreit vor Gericht kommen, so kann dadurch eine Beweispflicht (egal von welcher Seite) u.U. erleichtert werden. Außerdem kann damit einer möglichen „Salamitaktik" des Kunden vorgebeugt werden: Wenn der Kunde so nach und nach immer weitere Wünsche äußert, kann der Systementwickler sich auf das gemeinsam festgelegte Pflichtenheft berufen und für darüber hinaus gehende Kundenwünsche zusätzliche Aufwendungen in Rechnung stellen. Der Umfang des Pflichtenhefts kann bis zu mehreren hundert Seiten betragen, je nach Projektgröße.
Das Pflichtenheft sollte so gestaltet sein, dass der Softwareentwickler im Prinzip keine weiteren Angaben vom Kunden benötigt um das Anwendungsprogramm zu entwerfen und zu programmieren. In der Praxis kann es natürlich trotzdem sein, dass auch in späteren Phasen bei evtl. Unklarheiten oder Widersprüchen der Kunde konsultiert werden muss, um die Probleme beseitigen zu können.
Nachfolgendes Beispiel enthält bereits eine recht detaillierte semantische Datenbeschreibung, wie sie manchmal erst in der Nachfolgephase „DV-Entwurf" zu finden ist. Häufig schwimmen die Grenzen zwischen diesen beiden Phasen. Spätestens jedoch in der Entwurfsphase müssen alle Angaben genau spezifiziert sein, aber je mehr Angaben im Pflichtenheft gemacht werden, und umso genauer diese sind, desto besser und präziser kann die Entwurfsphase durchgeführt werden. Hier also ein Beispiel für einen Ausschnitt aus einem Pflichtenheft, welches der Autor dieses Buches mit einem Kunden der Großindustrie erarbeitete. Wie man sieht, handelt es sich dabei um eine detaillierte verbale Beschreibung eines relativ komplizierter Sachverhalts.

... Zunächst werden die Felder mit den Inhalten aus der Tabelle A90UMW gefüllt, und zwar alle Datensätze, bei denen die Felder KU_ZE und/oder KENZZGES gefüllt sind. Dazu die Folgedatensätze, bei denen die Felder TEL noch gefüllt sind, aber nur bis zum nächsten gefüllten Feld KU_ZE und/oder zum nächsten Telefoneintrag. Nun wird von der Tabelle BUERO_90 aus gearbeitet. Alle Datensätze die in den Feldern BURO_ART, GB_KURZ, KFZ_KENN und/oder ANSPRECH ein Kurzzeichen enthalten, werden mit den Feldern KU_ZE und KENZZGES auf Übereinstimmung geprüft, und zwar jedes Feld, das gefüllt ist. Ist Übereinstimmung vorhanden, muss noch der Ort überprüft werden. Dazu Feld ORT_ aus Tabelle BUERO_90 mit dem Ortsnamen im Feld NAMSITZ ab der Zeile mit dem gefundenen Code im Feld KU_ZE bis zum nächsten Eintrag im Feld KU_ZE und innerhalb derselben Blocknummer prüfen. Wird auch hier Übereinstimmung gefunden, dann werden die Felder aus BUERO_90 in die Tabelle ANSPRE übernommen und zwar in die Zeile, in der der Code steht. Wird zu einem Datensatz in der Tabelle BUERO_90 keine Übereinstimmung gefunden, wird der Datensatz an das Ende der Tabelle ANSPRE angehängt, und zwar für jedes ausgefüllte Codefeld ein Datensatz, hinter jedem hinzugefügten Datensatz werden drei weitere Zeilen mit dem Firmenkurzzeichen, einer Block- und Zeilennummer gefüllt und der relevante Ansprechpartnercode wird in das Feld KU_ZE eingetragen. Nach diesem Abgleich werden alle Datensätze, in denen die Felder KU_ZE und /oder KENZZGES gefüllt sind, aber welchen noch keine Adresse zugeordnet ist, mit der Adresse der jeweiligen Gesellschaft gefüllt (Straße, PLZ und Ort, Postfach, PLZ zum Postfach und Ort aus Tabelle gesell). Da es in der Gesellschaftstabelle zwei Datensätze geben kann, ist zu prüfen, ob unter NAMSITZ innerhalb des Blocks ein Ort aufgeführt ist (Feld NAME=1). Ist kein Ort aufgeführt, dann ist immer die Adresse aus dem Datensatz 1/2 zu nehmen. Ist ein Ort aufgeführt und er stimmt mit der Adresse von Satz 2/2 überein, dann ist diese Adresse zu nehmen, sonst wieder die Adresse von Satz 1/2. Nun ist jedem Datensatz der mit einer Adresse gefüllt ist, aber noch keinen Eintrag im A90UMW-Teil der Tabelle hat (außer dem Gesellschaftskurzzeichen) über die Beziehung zur vertansprech-Tabelle ein Ansprechpartner aus dieser Tabelle zuzuordnen. Ist nur ein Ansprechpartner vorhanden, wird dieser genommen. Sind mehrere Ansprechpartner vorhanden, wird der Ansprechpartnercode (Feld abt) in Tabelle vertansprech mit dem Code in KU_ZE verglichen. Bei Übereinstimmung erfolgt die Übernahme der Inhalte der Felder funktion, ansprech, telefon (telefax, funktel, email jeweils in die nächsten Zeilen, die bereits angelegt wurden). Ist keine Übereinstimmung vorhanden, wird der Datensatz mit der niedrigsten Zeilennummer genommen in dem ein Name eingetragen ist. Ganz zum Schluss müssen noch die Zeilen, die nur Gesellschaftskurzzeichen, Block- und Zeilennummern enthalten, gelöscht werden...

Dieses semantische Datenmodell ist schon recht ausführlich und gibt bereits einige Vorschläge zur Implementierung. Das ist allerdings nicht immer so. Oft werden hier nur Andeutungen gemacht, die dann erst im DV-Entwurf in dieser Detaillierungstiefe umgesetzt werden. Doch die Faustregel, nämlich: "je früher ins Detail gegangen wird, umso besser", sollte angewendet werden wann immer möglich.
Das obige Beispiel eines Ausschnittes aus einer semantischen Datenmodellierung ist –wie gesagt- schon sehr spezifisch. Am Allgemeinen beginnt ein Pflichtenheft mit der Beschreibung der vorhandenen „DV-Landschaft“, so dass der Entwickler weiß, was an Hard- und Software schon vorhanden ist. Es muss dann hier schon abgeklärt werden, ob und evtl. welche zusätzliche Hard- und Software angeschafft werden muss, damit das Projekt durchgeführt werden kann. Auch evtl. zusätzliches Personal sollte hier schon berücksichtigt werden.
Zur Beschreibung eines „IST-Ablaufs“ (wenn schon vorhanden) sowie des „SOLL-Ablaufs“ wird neben der verbalen Beschreibung häufig noch eine grafische Darstellung benutzt, die sich User-Case-Diagrammtechnik (oder einfach nur U-Cases) nennt (in UML nennt man dies auch Anwendungsfall-Diagramm).
U-Case-Diagramme bestehen im Wesentlichen aus nur 4 Komponenten: Strichmännchen, welche beteiligte Personen repräsentieren, Ellipsen, welche Aktionen anzeigen, Verbindungslinien, welche die beiden Letztgenannten in Verbindung bringen, sowie erläuternde Beschriftungen.
Wir betrachten hier zunächst ein Beispiel für den ISTZUSTAND in einem Unternehmen der Musikindustrie. Ein Tonträgerhersteller (früher „Schallplattenfirma“ genannt) will den Ablauf der Vertragsabwicklung und Lizenzabrechung mit dem Musikverlag, seinen Künstlern und Produzenten („Lizenzgeber“ genannt) zukünftig mit einer Software erledigen. Momentan wird das alles noch „von Hand“ durchgeführt. Bei der Istzustandsbeschreibung im Pflichtenheft kann also z.B. das nachfolgende U-Case-Diagramm helfen, den Sachverhalt und den Ablauf, wir er *vor* der Softwareentwicklung aussieht, zu klären.
Es ist allerdings wichtig zu erwähnen, dass ein U-Case-Diagramm alleine selten aussagekräftig genug ist, um daraus wirklich einen präzisen Ablauf erkennen zu können. In der Regel ergänzen U-Case-Diagramme nur eine verbale Beschreibung, die also auf jeden Fall auch im Pflichtenheft enthalten sein muss. In unserem Beispiel wird deshalb auch eine solche verbale Beschreibung dem Diagramm noch hinzugefügt.

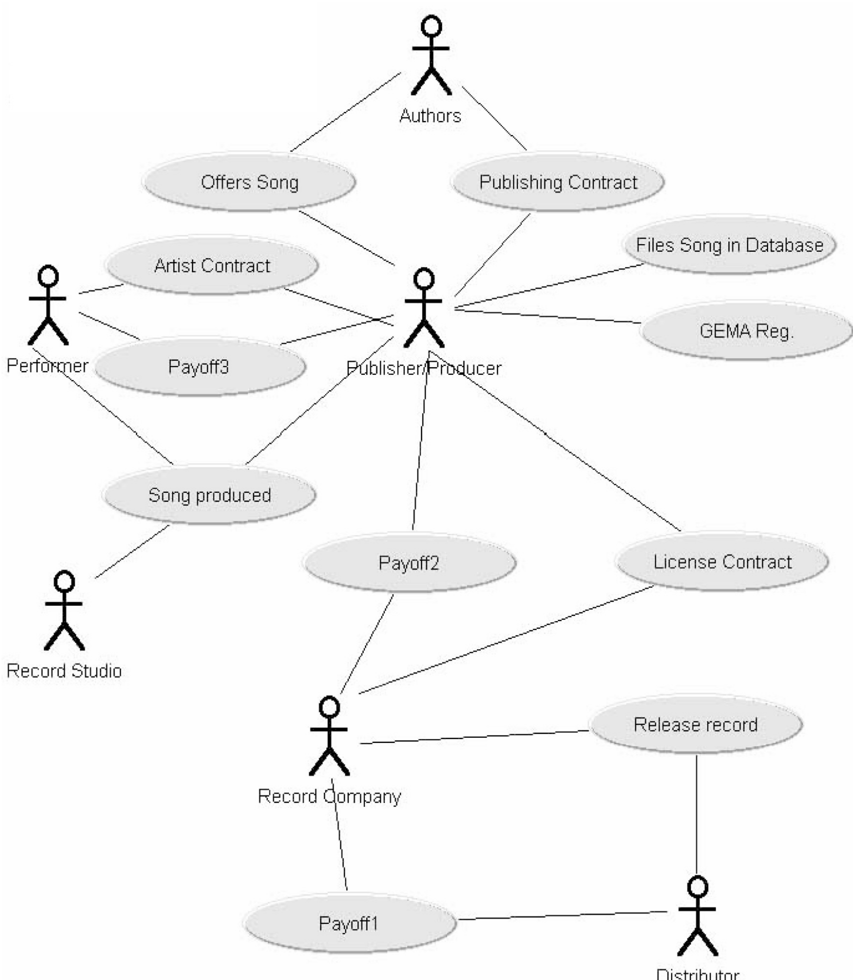


Abb. 4.1 U-Case-Diagramm ISTZUSTAND

Die zugehörige verbale Erläuterung könnte z.B. so lauten:

Ein Autor (in Abb. 4.1 Strichmännchen oben) offeriert eine Komposition (Song) einem Musikverleger oder Produzenten (Publisher/Producer, Strichmännchen oben Mitte). Gefällt dem Verleger das Lied, so wird ein Verlagsvertrag gemacht (Publishing Contract, oben rechts) und die relevanten Daten in einer Kartei abgelegt (Files Song in Database, oben rechts)

sowie eine Registrierung bei der GEMA durchgeführt. Jetzt beauftragt der Verleger einen Produzenten (bzw. manchmal sind Verleger und Produzent identisch) mit der Suche nach einem Interpreten (Performer/Artist, Strichmännchen mitte links). Wurde ein solcher gefunden, so wird mit demselben ein Künstlervertrag abgeschlossen (Artist Contract, mitte links). Danach geht der Produzent mit dem Künstler in ein Tonstudio (Record Studio, Strichmännchen links unten) und produziert (mind.) einen Song. Das Ergebnis ist ein sog. Master-Tape, also eine Aufnahme des fertig produzierten Liedes. Der Produzent macht sich jetzt auf die Suche nach einem Tonträgerhersteller, also der eigentlichen Plattenfirma (Record Company, Strichmännchen mitte unten). Hat er diese gefunden, so wird jetzt zwischen dem Produzenten und dem Tonträgerhersteller ein weiterer Vertrag abgeschlossen, der sog. Lizenzvertrag (License Contract, mitte rechts). Die Plattenfirma lässt daraufhin die Tonträger mit dem Song vervielfältigen und veröffentlicht diesen (Release Record, rechts unten), in dem sie einen Distributor (Strichmännchen rechts unten) mit dem Vertrieb der Tonträger beauftragt (welcher die Tonträger schließlich den Schallplattengeschäften verkauft). In regelmäßigem Turnus rechnet der Distributor mit der Plattenfirma über die verkauften Tonträger ab, d.h. er zahlt nach Abzug seiner Provision einen bestimmten Betrag (Payoff1, ganz unten mitte) an die Plattenfirma. Diese wiederum zahlt an den Produzenten die im Lizenzvertrag vereinbarte Provision (Payoff2, Bildmitte). Der Produzent schließlich zahlt daraufhin an den Künstler die im Künstlervertrag vereinbarte Beteiligung (Payoff3, oben rechts). Zu beachten ist dabei, dass zwar eine Abrechnung nach Anzahl verkaufter Tonträger erfolgt, jedoch sind ggf. nicht alle Songs eines Tonträger abrechnungsfähig, sondern nur diejenigen, welche dem entsprechenden Lizenzvertrag zugrunde liegen. Es ist also bei der Lizenzabrechnung nur der tatsächliche Anteil an Liedern eines Tonträger zu berücksichtigen. Der Autor erhält seine Zahlungen entweder direkt von der GEMA (an welche Rundfunkunternehmen ihre Sendetantieme entrichten müssen und der Tonträgerhersteller für jede gepresste Schallplatte eine Schutzgebühr zahlen muss), oder, falls er selbst nicht GEMA-Mitglied ist, vom Musikverlag (gemäß den Vereinbarungen im Verlagsvertrag), der in der Regel dann GEMA-Mitglied ist und von dort die entsprechenden Anteile erhält.

So eine Ablaufbeschreibung kann für den Softwareentwickler sehr hilfreich sein, auch wenn hier jetzt noch keine Datenstrukturen etc. definiert wurde. Aber es

soll die zu entwickelnde Software ja einen Teil der o.a. Realität wiederspiegeln, d.h. die beschriebenen Abläufe computerseitig abdecken.
Ist der Ablauf des Istzustands hinreichend beschrieben, dann geht es bei der Sollzustandsbeschreibung eigentlich nur noch darum, welche Teile der Realität wie auf dem Computer abgebildet werden sollen. Je nach dem, wie das Pflichtenheft strukturiert ist, kann jetzt bereits eine ausführliche Beschreibung erfolgen, oder, falls dies später im Pflichtenheft vorgenommen wird, eine grobe, mehr prinzipielle Darstellung. Dies sei in das Ermessen bzw. die Gepflogenheiten der beteiligten Unternehmen gestellt. Für unsere Zwecke wählen wir den Weg, dass zunächst eine allgemeine Sollbeschreibung vorgenommen wird. Die nachfolgenden Abschnitte geben dann weitere, detailliertere Informationen. Eine Sollzustandsbeschreibung bezogen auf unser Beispiel könnte demgemäss z.B. als U-Case-Diagramm nebst verbaler Beschreibung so aussehen:

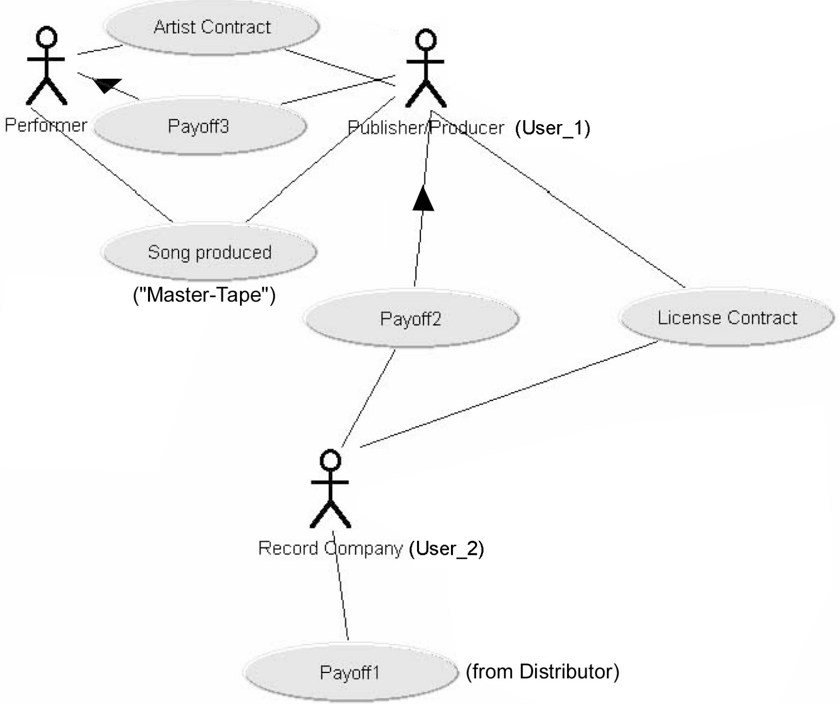


Abb. 4.2 U-Case-Diagramm Sollzustand (Abdeckungsbereich der Software)

```
Das zu entwickelnde Softwareprogramm soll den Teil des Ist-
zustands implementieren, welcher die administrativen Arbei-
ten bei der Vertragserstellung der Künstler- und Lizenzver-
träge abwickelt sowie die Lizenzabrechungen automatisch er-
stellt. Die Arbeiten der Musikverlagsverwaltung werden dabei
ausgekoppelt und evtl. zu einem späteren Zeitpunkt ebenfalls
automatisiert (eigenes Projekt). Es sollen sowohl (freie)
Produzenten als auch Schallplattenfirmen mit dem Programm
arbeiten können. In ersterem Fall werden Künstlerverträge
erstellt und abgerechnet und in zweiten Fall Schallplatten-
verträge. Grundlage der Abrechung ist in ersterem Fall die
Abrechnung des Produzenten mit dem Künstler und in zweitem
Fall die Lizenzabrechung der Plattenfirma an den Produzen-
ten. Die elektronisch abzuwickelnden Realitätsausschnitte
sind dem U-Case-Diagramm (Abb. 4.2) zu entnehmen. Die Benut-
zer des zu entwickelnden Programms werden dort mit User_1
(für den ersten Fall) und User_2 (für den zweiten Fall) be-
zeichnet.
```

Als Input für das zu schreibende Programm dienen also zum Zwecke der Vertragserstellung alle Angaben aus dem Künstler/Lizenzvertrag (insbesondere der für die spätere Abrechnung notwendigen Daten wie Vertragspartner, Vertragsdauer, prozentuale Beteiligungen etc.) und für die Abrechnung dann die Daten des Distributors (Payoff1) und/oder der Plattenfirma (Payoff2) (z.B. über Anzahl und Verkaufspreis der verkauften Tonträger). Der Output des Programms liefert dann die jeweiligen Verträge bzw. Abrechnungen. Damit verbunden ist die Möglichkeit gewisser statistischer Auswertungen (welcher Künstler hat wie viele Tonträger verkauft etc., siehe Problembeschreibung). Natürlich müssen alle bereits erfolgten Abrechungen gespeichert und doppelte Abrechnungen verhindert werden. Außerdem ist eine Adressverwaltung für alle beteiligten Firmen und Personen zu integrieren wie auch eine Tonträgerverwaltung, da sich in der Regel auf einem Tonträger mehrere Songs (Master-Tapes) befinden, die von verschiednen Interpreten sein können (d.h. sich auf verschiedene Künstler/Lizenzverträge beziehen).
Dieses Beispiel zeigt schon die grundsätzliche Problematik: Es muss einem DV-Fachmann der Ablauf eines Gebietes verständlich gemacht werden, von dem er i.d.R. keine Ahnung hat. Umgekehrt kennt der Auftraggeber normalerweise nicht die Methoden des Software-Engineering. Dies kann relativ hohe Anforderungen an beide Seiten stellen, denn es muss sich „in der Mitte“ getroffen und eine Spezifikation entwickelt werden, die beide Parteien vollständig verstehen können. Zur Präzisierung trägt es natürlich bei, wenn im Pflichtenheft schon Grundelemente der grafischen Datenbeschreibung vorhanden sind. Meine persönliche Erfahrung ist es, dass man durchaus einem Auftraggeber innerhalb

einer halben Stunden z.B. die Grundelemente eines ER-Diagramms erklären kann, so dass dann gemeinsam ein solches erarbeitet und dann in das Pflichtenheft aufgenommen werden kann. Es soll daher an dieser Stelle auf die wichtigsten Bestandteile eines ER-Diagramms eingegangen werden.
Chen entwickelte 1976 das sog. Entity Relationship Model (ERM), dessen grafische Darstellung dann Entity Relationship Diagram (ERD) heißt. Es findet praktische Anwendung hauptsächlich beim Entwurf von relationalen Datenbanken. Dieses Modell hat u.a. den Vorteil, dass es sowohl in der Analysephase für das Pflichtenheft sowie in verfeinerter Form im DV-Entwurf benutzt werden kann. Obwohl die Komponenten des ERM wie z.B. der Begriff einer Entität, einer Beziehung etc. erst später exakt definiert werden, soll hier soweit über den Aufbau schon gesprochen werden, dass wenigstens eine grobe Version des ERD, wie sie in Pflichtenheften zur semantischen Datenmodellierung häufig vorkommt, benutzt werden kann. Dazu betrachten wir folgende Komponenten:

Entity = Objekt der Realität oder Anschauung (z.B. Person, Projekt)

Relationship = Beziehung zwischen Enities, z.B. „...ist Mitarbeiter von...“

In relationalen Datenbanken werden Entitäten und Beziehungen in der Regel durch Tabellen dargestellt, wobei die Namen der Spaltenüberschriften auch *Attribute* genannt werden. Solche Attribute besitzen einen Wertebereich, engl. Domain, wie z.B. *integer* für ganze Zahlen oder *Character* für Buchstaben. Entities und Relationships fasst man bei Bedarf auch zu Mengen oder Klassen zusammen:

Entity set/class = Menge/Klasse von Objekten, z.B. PERSON, PROJEKT etc.

Relationship set/class = Menge/Klasse von Beziehungen

Bei Beziehungen spielt noch der sog. Assoziationstyp eine wichtige Rolle. Er gibt an, in welchem numerischen Verhältnis die Einträge der beteiligten Entitäten stehen. Die wichtigsten Assoziationstypen sind *1:n* (sprich „1 zu n“) und *n:m*. Beispielsweise hat *1* Unternehmen *n* Abteilungen, während in *n* Projekten *m* Mitarbeiter beschäftigt sein können (siehe Beispiel Abb. 4.3).

Für die grafische Darstellung sind die wichtigsten Symbole:

	Entity (Entität)
	Relationship (Relation, Beziehung)
	Attribut
	Verbindungsline

Nachfolgend sei ein Beispiel angegeben, wie es typischerweise in einem Pflichtenheft vorkommen könnte.

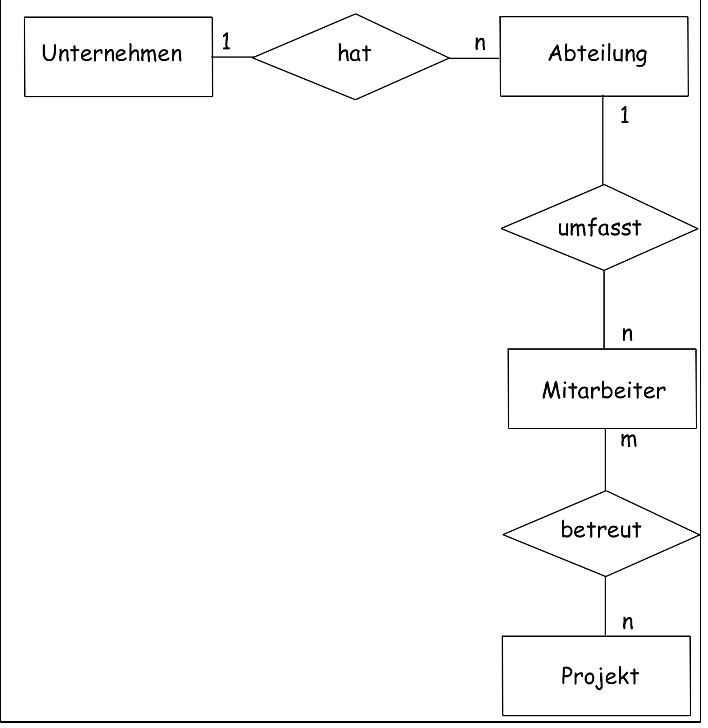


Abb. 4.3 Beispiel für ein Entity Relationship Diagram

Die *n:m*-Beziehung in Abb. 4.3 bedeutet konkret, dass an *einem* Projekt *m* Mitarbeiter beteiligt sein können und dass jeder Mitarbeiter an *n* Projekten beteiligt sein kann. Zu betonen wäre noch, dass für die Beschriftungen der Entitäten und Beziehungen der Singular verwendet wird, während in Datenflussdiagrammen üblicherweise der Plural benutzt wird.
Wir werden im nächsten Kapitel nochmals auf ERDs zurückkommen und ein wenig detailliertere Betrachtungen darüber anstellen.
Nun ist es so, dass ERDs nur statische Sachverhalte wiedergeben können. Sie sind damit z.B. für den Entwurf von Datenbanken sehr geeignet, denn die Datenstrukturen werden als stabil über die Zeit betrachtet; was sich mit der Zeit ändert, sind allenfalls die Inhalte der Tabellen. Nun gibt es aber oft Anwendungen, wo sich der Zustand der beteiligten Komponenten während eines Programmablaufs ändern kann. Z.B. die Steuerung eines Geldautomaten setzt gewisse zeitabhängige Abläufe voraus (erst Geldkarte eingeben, dann PIN, dann Geldbetrag etc.). In so einem Fall reicht die statische Beschreibung (wie mit ERD) nicht aus. Es werden dafür zusätzlich repräsentative Szenarien beschrieben, und zwar mit der Hilfe sog. Zustands- und Sequenzdiagramme. Dadurch werden die zeitlichen Abläufe und Reihenfolgen gewisser Aktionen deutlich. Dies ist insbesondere manchmal wichtig für gewisse Abhängigkeiten der Daten untereinander. So können z.B. manche Berechnungen erst dann ausgeführt werden, wenn zeitlich vorher bestimmte Daten eingegeben wurden.
Die Darstellung solcher Zeitabhängigkeiten kann durch gewöhnliche verbale Beschreibung der Abfolgen geschehen und/oder durch standardisierte Zustandsdiagramme. Letztere beschreiben die zeitliche Abfolge wichtiger Änderungen am Zustand des Systems. Jede Eingabe, Ausgabe oder sonstige Aktion der Software wie z.B. eine Berechnung, verändert den aktuellen Zustand der Software bzw. der Hardware (z.B. des Bildschirms oder Druckers oder der Festplatte etc.). Ein Beispiel für eine verbale Szenariobeschreibung in Verbindung mit einem entsprechenden U-Case-Diagramm haben wir bereits in der Ist- und Sollzustandsbeschreibung gesehen.
Zur Zeitdynamischen Modellierung wollen wir nur kurz die Syntax zweier „Sorten“ von dynamischen UML-Diagrammen angeben[10]:

Mittels *Sequenzdiagrammen* beschreibt man die Interaktionen zwischen den Modellelementen. Jedoch steht beim Sequenzdiagramm der zeitliche Verlauf des Nachrichtenaustausches im Vordergrund. Die Zeitlinie verläuft senkrecht von oben nach unten, die Objekte werden durch senkrechte Lebenslinien beschrieben und die gesendeten Nachrichten waagerechtentsprechend ihres zeitlichen Auftretens eingetragen.

---

[10] Für eine ausführlichere Behandlung siehe z.B. Zöller-Greer, P.: *Software-Architekturen: Grundlagen und Anwendungen*, Verlag composia, 2010

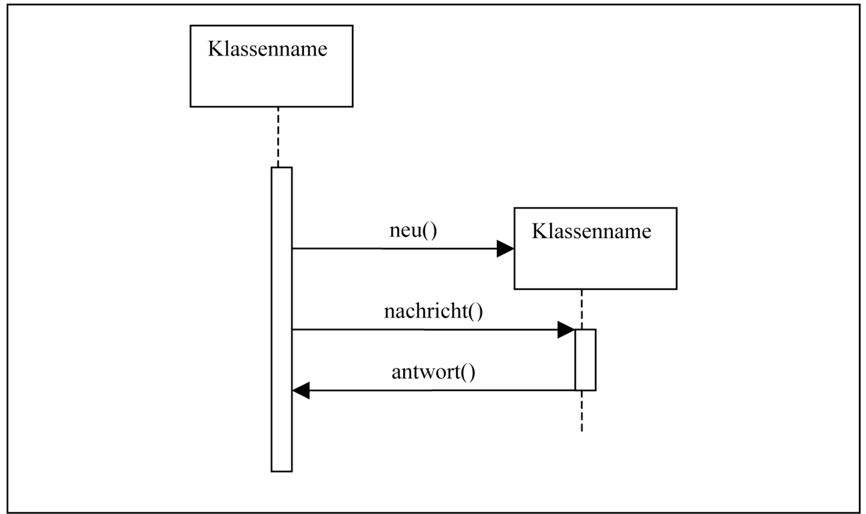


Abb. 4.4 Notation Sequenzdiagram

Die Objekte werden durch Rechtecke visualisiert. Von Ihnen aus gehen die senkrechten Lebenslinien, dargestellt durch gestrichelte Linien. Die Nachrichten werden durch waagerechte Pfeile zwischen den Objektlebenslinien beschrieben. Auf diesen Pfeilen werden die Nachrichtennamen in der Form:

nachricht(argumente)

notiert. Nachrichten, die als Antworten deklariert sind, erhalten die Form:

antwort: =nachricht(...).

Nachrichten können Bedingungen der Form:

[bedingung] nachricht(...)

zugewiesen werden. Iterationen von Nachrichten werden durch ein Sternchen "*" vor dem Nachrichtennamen beschrieben. Objekte, die gerade aktiv an Interaktionen beteiligt sind, werden durch einen Balken auf ihrer Lebenslinie gekennzeichnet. Objekte können während des zeitlichen Ablaufes des begrenzten Kontextes erzeugt und gelöscht werden.

Ein Objekt wird erzeugt, indem ein Pfeil mit der Aufschrift *neu()* auf ein neues Objektsymbol trifft.

Ein *Zustandsdiagramm* beschreibt eine hypothetische Maschine, die sich zu jedem Zeitpunkt in einer Menge endlicher Zustände befindet. Sie besteht aus:

- einem Anfangszustand
- einer endlichen Menge von Zuständen
- einer endlichen Menge von Ereignissen
- einer endlichen Anzahl von Transaktionen, die den Übergang des Objektes von einem zum nächsten Zustand beschreiben
- einem oder mehrerer Endzustände

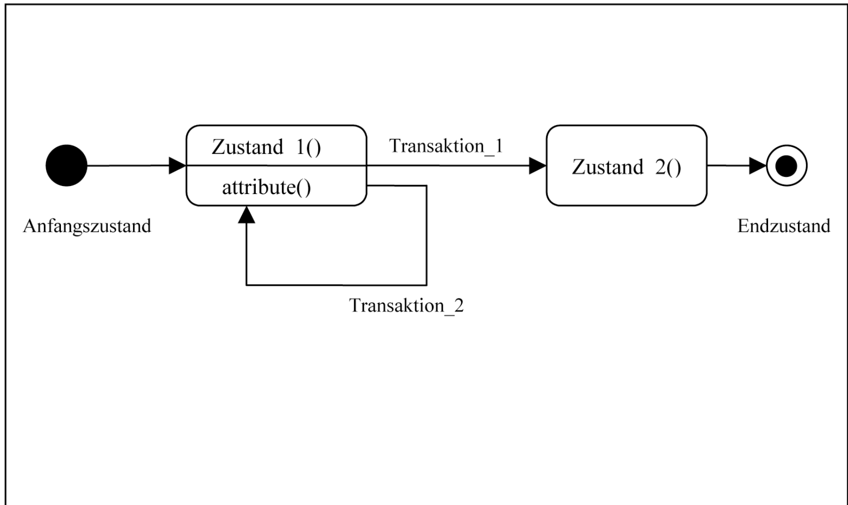


Abb. 4.5 Notation Zustandsdiagramme

In Zustandsdiagrammen werden Zustände als abgerundete Rechtecke, verbunden durch Pfeile, auf denen die Transaktionen stehen, dargestellt. Anfangszustand ist ein gefüllter Kreis, der Endzustand ist ein leerer Kreis mit einem kleinen gefüllten Kreis in der Mitte.

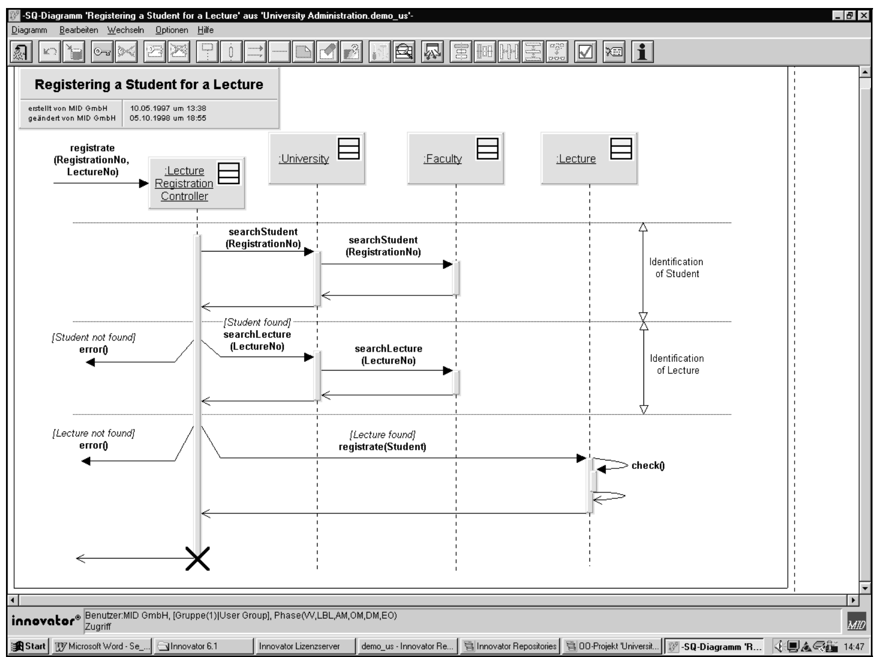


Abb. 4.6 Beispiel für ein Sequenzdiagramm im CASE-Tool Innovator®

Da hier Änderungen im Zeitverlauf dargestellt werden können, spricht man manchmal auch von *dynamischer Datenmodellierung*.

Es bleibt nun dem „Gefühl" des Systemanalytikers vorbehalten, ob neben den U-Case-Diagrammen bereits im Pflichtenheft schon ERDs bzw. Sequenz- und/oder Zustandsdiagramme aufgenommen werden sollen. Kann der Auftraggeber diese verstehen, sollte man es machen. Wenn nicht, muss es im Pflichtenheft bei der verbalen Beschreibung bleiben und erst im DV-Entwurf werden diese Diagrammtechniken eingesetzt.

Übungen zum Selbsttest

1. Was muss ein semantisches Datenmodell beinhalten? In welcher Phase des Wasserfallmodells kommt es vor?

2. Ein Konzern hat folgende Hierarchie:
   Eine Holding wacht über Geschäftsbereiche. Jeder Geschäftsbereich ist in Hauptabteilungen und diese sind jeweils in Abteilungen unterteilt. Jede Abteilung beschäftigt Mitarbeiter. Davon gibt es Außendienstmitarbeiter und welche im Innendienst. Sowohl im Außen- als auch im Innendienst gibt es freie und feste Mitarbeiter. Innerhalb der Mitarbeiter gibt es Vorgesetzte und Untergebene.

   a) Erstellen Sie als Softwarehersteller ein Angebot für die Entwicklung einer Verwaltungssoftware des Personals des Konzerns, welche die obigen Hierarchien berücksichtigt.
   b) Entwickeln Sie ein semantisches Datenmodell und ein Pflichtenheft („erfinden" Sie ggf. sinnvolle Annahmen)
   c) Wählen Sie ein Phasenmodell und geben Sie kurz die jeweiligen Inhalte jeder Phase bezogen auf das obige Projekt wider.

3. Ein Softwareunternehmen habe n Mitarbeiter, die 8 Stunden am Tag arbeiten. Aufgrund statistischer Untersuchungen geht die Geschäftsführung davon aus, dass im Durchschnitt jeder Mitarbeiter 6 Minuten am Tag mit einem anderen spricht.
   a) Ab wie viel Mitarbeitern sind alle nur noch mit reden beschäftigt, so dass keine produktive Arbeit mehr getätigt werden kann?
   b) Ab wie viel Mitarbeitern hat die Produktivität (also die Zeit des tatsächlichen Arbeitens) ihr Maximum?

5. Datenmodellierung

In Kapitel 4 lernten wir schon eine einfache Datenmodellierungen kennen: Das *semantische Datenmodell*, welches Datenabläufe und Strukturen verbal oder mit Hilfe einfacher Diagrammtechniken (U-Case, ERD, ggf. auch schon Sequenz- und/oder Zustandsdiagramme) beschreibt. In diesem Kapitel soll die Datenmodellierung weiter vertieft werden. Im Wasserfallmodell entspricht dies der Phase „DV-Entwurf". In der Praxis wird man allerdings nur die Phasen Initialisierung und DV-Konzept nach dem Wasserfallmodell durchführen und dann aus den in Kapitel 1 und 2 erläuterten Gründen zum Spiralmodell übergehen.
Ungeachtet dieser „Meta-Vorgehensweise" wollen wir uns einiger etablierter Techniken des DV-Entwurfs annehmen. Wir wollen dabei den relationalen und den objektorientierten Ansatz näher untersuchen. Dabei gehen wir so vor, dass wir zunächst aus der objektorientierten Welt den Klassenbegriff ableiten und danach in den „Spezialfall" des relationalen Ansatzes einführen. Von da aus verallgemeinern wir dann zum objektorientierten Ansatz. Diese auf den ersten Blick umständlich erscheinende Vorgehensweise ist meiner Erfahrung nach aber für das tiefere Verständnis für den Leser die einfachere.

Klassen und Objekte
In jüngster Zeit setzen sich objektorientierte Ansätze gegenüber relationalen immer mehr durch. Dies liegt u.a. daran, dass eine objektorientierte Systemanalyse (OOA) eine bessere Abbildung der Realität ermöglicht, denn dort hat man es ja gerade mit Objekten zu tun. Eine präzise Definition des Objektbegriffs ist dabei sehr schwer. Wir lehnen uns an die Definition des Duden (Bedeutungswörterbuch, BI 1985) an:

Definition 5.1 (**Objekt**)
Unter einem *Objekt* versteht man eine Person oder einen Gegenstand (der Realität oder Anschauung), auf die/den das Denken und Handeln bzw. jemandes Interesse gerichtet ist.

Diese Definition mag etwas verschwommen erscheinen, aber man versteht, dass es sich bei einem Objekt um eine „Sache" handelt, die wahrgenommen und hinlänglich beschrieben und über die „nachgedacht" werden kann. Im Softwareengineering interessiert in erster Linie, wie Objekte aus dem Leben auf den Computer abgebildet werden können. Natürlich gibt es dabei auch Objekte, die ausschließlich auf einem Computer existieren, man denke z.B. an eine Schaltfläche oder den Mauszeiger etc.; auch das sind Objekte, denn sie besitzen unbestreitbar zumindest eine virtuelle Realität und man kann sein Interesse auf sie richten.

Möchte man Objekte auf einem Computer abbilden, so müssen diese in Form von Daten abgelegt werden. Und Daten auf dem Computer erfordern ein Datenformat, also eine Datenstruktur. Die Beschreibung von Datenstrukturen und ihren Beziehungen nennt man auch Datenmodell. Das ist aber noch nicht alles. Auf Daten muss man zugreifen können, d.h. man benötigt Zugriffsoperationen wie Edit, Delete, Add usw.; außerdem fasst man ähnliche Daten häufig zusammen. Es muss also Kriterien geben, welche die Zugehörigkeit (auch *Integrität* genannt) festlegen. Solche Zugehörigkeitskriterien nennt man auch *Integritätsbedingungen*. Grob kann man also sagen, was für ein Objekt im datentechnischen Sinn gelten muss:

Objekt	=	Datenstruktur + Operationen + Integritätsbedingungen
Objektmodell	=	Datenmodell + Operationen + Integritätsbedingungen

Dadurch, dass also auch Operationen zu einem Objekt gehören, sind die Daten eines Objekt selbst nur über diese Operationen zugreifbar. Man redet in diesem Zusammenhang von *Datenkapselung*. Die Operationen werden dabei als Funktionen implementiert und dann *Methoden* genannt (vgl. Abb. 5.1). Ein erstelltes Datenmodell wird manchmal auch ein *Schema* genannt.

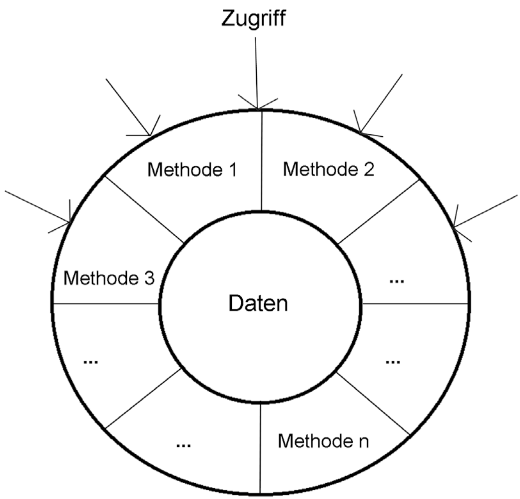


Abb. 5.1 Datenkapselung in einem Objekt

Objekte, die in irgend einem Sinne zusammengehörig sind, werden in sog. Klassen zusammengefasst. Es muss also ein Kriterium geben, nach welchem die Zusammengehörigkeit von Objekten bestimmbar ist. In der 4. Schulklasse einer Grundschule beispielsweise sitzen Kinder (Objekte), denen gemeinsam ist, dass sie alle die 3. Klasse hinter sich gebracht haben, einer bestimmten Altersgruppe angehören, einen bestimmten (Klassen-) Lehrer haben, in einem bestimmten (Klassen-) Zimmer sitzen, einen bestimmten Stundenplan haben usw.; in der Mathematik kennt man u.a. Restklassen. Darunter versteht man die Zusammenfassung von denjenigen ganzen Zahlen, die bei Division durch eine festgelegte Zahl den gleichen Rest besitzen. Betrachtet man z.B. die Menge {..., 7, 11, 15, 19...}, so handelt es sich hierbei um die Klasse der ganzen Zahlen, welche bei Division durch die Zahl 4 alle den gleichen Rest, nämlich 3, besitzen. Dieser Klassenbegriff führt zu nächsten Definition.

Definition 5.2 (***Klasse***)
Unter einer *Klasse* versteht man eine Menge von zusammengehörigen Objekten. Eine atomare Klasse ist eine Klasse mit nur einfachen (also nicht zusammengesetzten) Objekten.

Die Objekte von Klassen bekommen einen eigenen Namen:

Definition 5.3 (**Instanz**)
Die Objekte einer Klasse werden auch *Instanzen* oder *Exemplare* der Klasse genannt.

Instanzen oder Exemplare einer Klasse können prinzipiell selbst wieder Klassen sein. Es sei bemerkt, dass die sich Menge der Exemplare einer Klasse mit der Zeit verändern kann. Daher sind eine Klasse und ihre Exemplarmenge i. A. nicht dasselbe. Betrachtet man z.B. eine Adresstabelle, so kann natürlich diese Tabelle durch Löschen oder Hinzufügen von Einträgen verändert werden. Stellt die Tabelle eine Klasse dar und die Zeilen darin sind die Instanzen, so ändert man z.B. beim Löschen einer Zeile die Exemplarmenge. Daher sollte man zwischen den beiden Begriffen wohl unterscheiden.

Relationale Ansätze
Wir kommen hier noch mal auf Entitäten und Relationen zurück. In relationalen Datenbanken werden Entitäten und Relationen als Tabellen repräsentiert (wobei alle Tabellen mathematisch gesehen Relationen darstellen). Es werden hierbei gewisse Datenbankkenntnisse vorausgesetzt. So sollten die Begriffe DBMS, Schlüsselkandidaten, Primärschlüssel, Fremdschlüssel, referentielle Integrität und mind. die ersten 3 Normalformen bekannt sein.

Beim Entwurf von Softwaresystemen spielen bei relationalen Ansätzen die Entitäten eine wichtige Rolle. Entitäten sind im Prinzip Objekte, deren Struktur durch das Vorhandensein von Attributen gekennzeichnet ist. In der Literatur wird häufig eine zirkuläre Definition derart benutzt, dass man sagt, Entitäten sind Objekte, welche Attribute besitzen, und später dann Attribute als Bestandteile von Entitäten definiert. Es sei jetzt der Attributsbegriff genauer betrachtet:

*Definition 5.4 (***Attribut***)*
Ein *Attribut a* einer Exemplarmenge einer Klasse *K* ist eine Abbildung von der Exemplarmenge von K in die Menge der Werteklasse W ($a{:}K \rightarrow W$), wobei die Werteklasse *W* aus der Menge der möglichen Domains der Attribute besteht. Eine Werteklasse wird daher auch manchmal Domainklasse oder einfach nur Domain eines Attributs genannt.

Diese Präzisierung des Attributsbegriffs erlaubt eine allgemeinere Verwendung, insbesondere in Hinblick auf objektorientierte Datenmodellierung. Aber auch in der Welt der relationalen Modellierung ist diese Präzisierung von Nutzen. Die Werteklassen für Tabellenattribute ist z.B. im Datenbanksystem MS-Access, welches teilweise zur Erstellung der Tabellen in nachfolgenden Abbildungen diente, gegeben durch (WKM steht für „Werteklassenmenge“):

$W \in WKM$ = *{Text, Memo, Zahl, Datum/Uhrzeit, Währung, AutoWert (Zähler), Ja/Nein (Boolesch), OLE-Objekt, Hyperlink}*

Das heißt: Jeder Eintrag in diesen Tabellen ist von einem Datentyp aus WKM. Die Einträge einer festen Spalte sind immer vom gleichen Datentyp (*W*). Im Beispiel einer Adresstabelle könnte das konkret folgendes bedeuten:

Klasse *K*	=	Tabelle „Adressen“
Werteklasse	=	$W \in WKM$ (siehe oben)
Attribute	=	*{lfd, Name, Ort, Strasse, Telefon}*
Exemplarmenge	=	Datensätze (Zeilen) der Tabelle

Damit ist z.B. das Attribut *a=NAME* eine Abbildung von der Exemplarmenge (den Daten der Tabelle, genauer, denjenigen Daten, die in der Spalte *NAME* stehen) in die Werteklasse *W* (genauer, die Klasse *Text*). Attribute, deren Donains entweder numerisch, alphanumerisch (Charakters) oder Boolesch sind, nennt man auch elementare Attribute. Allgemein können Attribute auch zusammengesetzt sein. Wenn dies der Fall ist, so ist W ein Element der Potenzmenge der oben angegeben Werteklassenmenge.
Nachdem der Attributsbegriff formal eingeführt ist, sei jetzt auch der Begriff einer Entität genauer spezifiziert. Dieser Begriff ist in der objektorientierten

Welt allerdings kaum noch von Bedeutung, da der dort eingeführte Klassenbegriff eigentlich alles bereits gut abdeckt.

Definiton 5.5 (**Entität**)
Eine *Entität* ist eine abstrakte Repräsentation eines Objektes der Anschauung oder der Realität.

Entitäten kann man nun zu sog. Entitätsklassen zusammenfassen. Entitätsklassen sind aber immer atomare Klassen, d.h. Entitäten sind nicht zusammengesetzt. Identifiziert ein Attribut ein Element der Exemplarklasse eindeutig (das Äquivalent eines Schlüssels), so wird dieses Attribut auch Identifikator genannt. In der Adresstabelle ist das Attribut *lfd* so ein Identifikator, da diese Zahl jeden Datensatz eindeutig identifiziert.

Wir hatten Tabellen als mögliche Repräsentationen von Relationen aufgefasst. Nun kann es ja auch Beziehungen zwischen Tabellen geben (z.B. eine *1:n*-Beziehung zwischen den Entitäten *Abteilung* und *Mitarbeiter*). In gewissem Sinn handelt es sich dabei also um Relationen von Relationen. Um diesem Tatbestand gerecht zu werden, ist es sinnvoll, Relationen als Instanzen von sog. *Relationenklassen* oder *Beziehungsklassen* aufzufassen:

Definition 5.6 (**Relationenklasse, Beziehungsklasse**)
Eine *Relationenklasse* oder Beziehungsklasse ist eine Relation
$y \subseteq y_1 \times y_2 \times \ldots \times y_n$, wobei jede Klasse y_i eine Entitätsklasse oder selbst eine Relationenklasse ist.

Die Klassen y_i müssen dabei nicht notwendig verschieden sein.

Wir hatten nun schön des öfteren über *1:n*- und *n:m* - Beziehungen gesprochen. Im Softwareengineering nennt man so etwas auch *Kardinalitätsbeschränkungen* oder *Multiplizitäten*. Wir wollen diesen Begriff etwas genauer definieren, doch dazu benötigen wir erst noch etwas Vorarbeit.

Wir werden dazu den Funktionsbegriff aus der Mathematik etwas verallgemeinern.

Definition 5.7 (**Funktionen**)
Funktionen sind Abbildungen von einer Exemplarmenge $Y_{1,t}$ einer Klasse Y_1 in die Exemplarmenge $Y_{2,t}$ einer Klasse Y_2 zum Zeitpunkt *t*. In Zeichen: $f_t: Y_{1,t} \rightarrow Y_{2,t}$. Man unterscheidet dabei:

- ***total definierte Funktion***: $f_t^1: Y_{1,t} \rightarrow Y_{2,t}$,
  d.h. zum Zeitpunkt t gilt: $\forall x_1 \in Y_{1,t}$ gibt es genau ein $x_2 \in Y_{2,t}$
- ***partiell definierte Funktion***: $f_t^c: Y_{1,t} \rightarrow Y_{2,t}$,
  d.h. zum Zeitpunkt t gilt: $\forall\ x_1 \in\ Y_{1,t}$ gibt höchstens ein $x_2 \in\ Y_{2,t}$
- ***mehrfach definierte Funktion***: $f_t^m: Y_{1,t} \rightarrow P(Y_{2,t})$,
  wobei P = Potenzmenge, d.h. zum Zeitpunkt t gilt: $\forall\ x_1 \in\ Y_{1,t}$ gibt es ein oder mehrere $x_2 \in\ Y_{2,t}$.

Es sei angemerkt, dass der übliche Funktionsbegriff in der Mathematik eigentlich nur mit total definierten Funktionen aus dieser Definition übereinstimmt. Bei partiell definierten Funktionen kann es Lücken geben und mehrfach definierte Funktionen entsprechen in der Mathematik Kurven, d.h. ein x-Wert kann mehrere y-Werte haben. Der hochgestellte Index des Funktionssymbols bedeutet

1	=	eindeutig (bei total definierten Funktionen)
c	=	conditioned (bei partiell definierten Funktionen)
m	=	multiple (bei mehrfach definierten Funktionen).

Es haben sich auch folgende Schreibweisen eingebürgert:

Für partiell definierte Funktionen:	(0,1)
Für total definierte Funktionen:	(1,1)
Für mehrfach definierte Funktionen:	(1,*)

In dieser Version steht die Schreibweise für (min, max) der möglichen Kardinalitäten der Exemplarmengen, d.h. es werden die unteren und oberen Schranken der Kardinalitäten angegeben („*“ steht dabei synonym für „viele“). Daher spricht man hier auch von Kardinalitätsbeschränkungen.
Eine andere, weit verbreitete alternative Schreibweise ist folgende:

c	(conditioned) bedeutet:	(0,1)
1	bedeutet:	(1,1)
m	(multiple) bedeutet:	(1,*)
mc	(multiple conditioned) bedeutet:	(0,*)

Diese Abkürzungen nennt man auch *Assoziationstypen*.

Trotz des stetigen Vordringens objektorientierter Anätze werden für Datenbanken fast ausschließlich relationale Datenbanksysteme verwendet. Es gibt zwar auch objektorientierte Datenbanken, doch diese sind wenig verbreitet. Für relationalen Datenbanksysteme wurden entsprechende relationale Ansätze im Entwurfsbereich entwickelt. Es handelt sich dabei um vorwiegend grafische Methoden zur Darstellung der Datenstrukturen und ihrer Beziehungen. Diese auch als ER-Diagramme oder ERD bezeichnete Technik wurde von Chen in den 70er Jahren des letzten Jahrhunderts entwickelt und ist bis heute weit verbreitet. Wir haben bereits die Grundstruktur dieser Methode besprochen, da sie schon bei der Erstellung des Pflichtenhefts für die semantische Datenmodellierung von Nutzen sein kann. Der ursprüngliche Standard von Chen hat sich seither weiterentwickelt und wird bis heute immer wieder erweitert. In diesem Abschnitt sollen unsere Kenntnisse von ER-Diagrammen so erweitert werden, dass sie auch auf den detaillierteren DV-Entwurf angewendet werden können.
Man kann nun mittels einer Ellipse Attribute als grafisches Element in das Diagramm einzubringen. Doch in der Praxis wird dies sehr selten gemacht, da dadurch das ER-Diagramm schnell unübersichtlich wird. Wenn man bedenkt, dass es viele Entitäten geben kann, und jede Entität wiederum etliche Attribute besitzen kann, dann ist das ER-Diagramm schnell überladen mit Symbolen. Betrachten wir wieder das Beispiel der Adresstabelle. Das ER-Diagramm würde hierfür wie folgt aussehen:

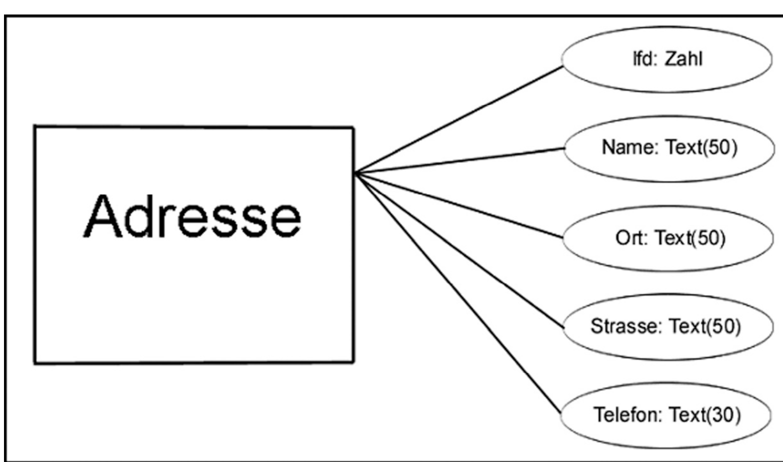


Abb. 5.2 Attribute im ER-Diagramm

Neben der Möglichkeit, die Attribute grafisch in das Diagramm aufzunehmen, gibt es natürlich auch eine textuelle Beschreibung der Attribute einer Entität im

Entity Relationship Modell. Der Aufbau sei am Beispiel der Adresstabelle gezeigt:

Adresse

	Primary Key				
Attribut	Lfd	Name	Ort	Strasse	Telefon
Domain	Zahl (Integer)	Text(50)	Text(50)	Text(50)	Text(30)

Links oben steht der Name der Entität (Adresse). Die nächste Zeile identifiziert den Schlüssel der Tabelle (Primary Key), gefolgt von der Zeile mit den Attributsnamen. In der letzten Zeile werden schließlich die Werteklassen für jedes der Attribute angegeben (Domain). Es ist aus Gründen der Übersichtlichkeit empfehlenswert, die Attribute im ER-Diagramm wegzulassen und eine getrennte, textuelle Attributsbeschreibung für jede Entität zu erstellen.
Nun lassen sich natürlich auch Relationen zwischen Tabellen beschreiben. Zu diesem Zweck betrachten wir nochmals ein Beispiel, das wir bereits bei der semantischen Datenmodellierung kennen gelernt hatten. Mit Hilfe unserer jetzigen Kenntnis über Kardinalitätsbeschränkungen können wir die Assoziationstypen jetzt genauer angeben. In Abb. 5.3 sehen wir statt der *1:n* und *n:m*-Beziehungen aus Abb. 4.3 jetzt die Eintragungen in der (min, max)-Schreibweise. Wir lesen z.B. bei den Unternehmen und Abteilungen von links nach rechts und sagen, dass ein Unternehmen ein oder mehrere Abteilung haben kann; es handelt sich also um eine (1,1):(1,*)-Beziehung. Der Eintrag „hat" in der Beziehungsraute wird manchmal auch die *Rolle* (oder, bei anderer Sichtweise auch *inverse Rolle*) der Beziehung genannt. Sie kann auch an die Verbindungslinien geschrieben werden.

Bevor wir nun zur objektorientierten Modellierung kommen, soll kurz erläutert werden, wie man die relationalen Beziehungen zwischen Tabellen praktisch umsetzt. Wir betrachten folgende Fälle:

(1,1):(1,1)-Beziehung:
Hier sollte gut überlegt werden, ob man überhaupt zwei Tabellen braucht! Normalerweise kann daraus eine einzige Tabelle gemacht werden, die alle Attribute aus beiden Tabellen enthält. Will man aber unbedingt zwei Tabellen haben, so wird dies i.d.R. so realisiert wie im nächsten Fall der (1,1):(0,1)-Beziehung.

(1,1):(0,1)-Beziehung:
Hier das (die) Primärschlüsselattribut(e) der (1,1)-Seitentabelle als Fremdschlüsselattribut(e) auf der (0,1)-Seitentabelle hinzufügen und dann diese(s) Fremdschlüsselattribut(e) selbst zum Primärschlüssel der (0,1)-Seitentabelle machen. Damit ist ereicht, dass der auf die (1,1)-Seite referenzierende Datensatz der (0,1)-Seite höchstens einmal in der (0,1)-Seite vorkommen kann.

(1,1):(0,*)-Beziehung oder (1,1):(1,*)-Beziehung:
Hier das (die) Primärschlüsselattribut(e) der (1,1)-Seitentabelle als Fremdschlüsselattribut(e) zu der (0,*)-Seitentabelle hinzufügen, aber zusätzlich auf der (0,*)-Seitentabelle einen eigenen Primärschlüssel erzeugen (also nicht den Fremdschlüssel dafür benutzen!).

(0,*):(0,*)-Beziehung oder (1,*):(0,*)-Beziehung oder
(1,*):(1,*)-Beziehung oder (0,*):(1,*)-Beziehung:
Hier wird eine eigene, neue „Beziehungstabelle" angelegt, welche die Primärschlüsselattribute aus beiden Seiten als Fremdschlüssel enthält. Die Kombination aller Fremdschlüssel kann dann zum Primärschlüssel der Beziehungstabelle gemacht werden.

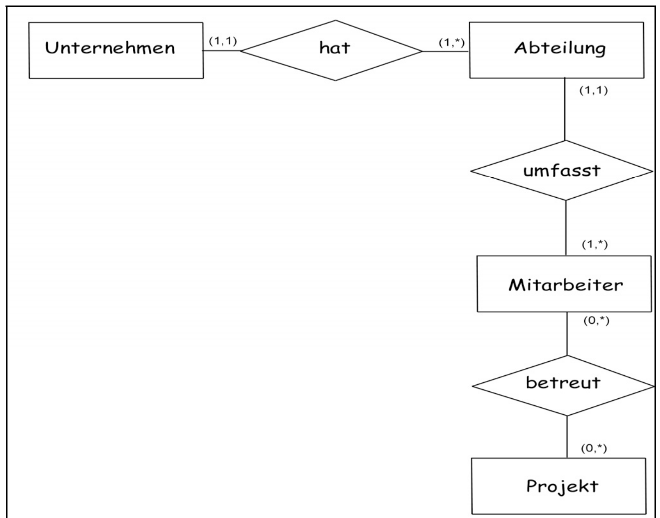


Abb. 5.3 ER-Diagramm mit präziseren Kardinalitätsbeschränkungen

Betrachten wir noch einen Spezialfall einer Beziehung, der manchmal in der Praxis nützlich ist und die Möglichkeit bietet, Hierarchien in einer Tabelle abzu-

bilden. Nehmen wir z.B. einmal an, dass ein Stammbaum einer Familie abgebildet werden soll. Es sei also eine Entität geben, welche die Namen aller Familienangehörigen enthält. Des Weiteren will man wissen, wer ist Vater oder Mutter von wem, wer hat welche Großeltern, Urgroßeltern, Kinder, Enkelkinder, Großenkelkinder usw.; es soll keine Beschränkung nach „oben“ oder „unten“ in der Eltern-Kind-Beziehung geben. Insbesondere kann eine Person gleichzeitig jemandes Enkel, Urenkel, Vater, Großvater, Urgroßvater etc. sein. Dieses Problem lässt sich durch eine *n:m*-Beziehung auf ein und dieselbe Entität lösen. Solche Beziehungen nennt man auch *reflexive Beziehungen* (nicht zu verwechseln mit dem mathematischen Begriff der reflexiven Relation!).

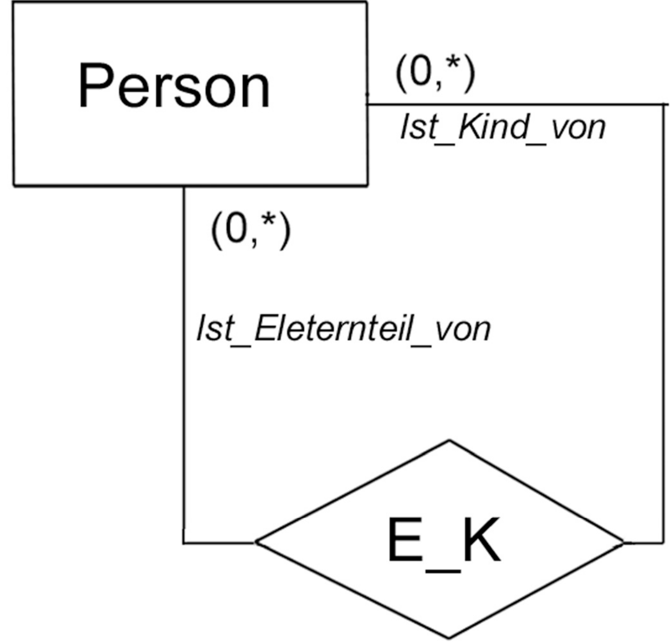


Abb. 5.4 Reflexive Beziehung

Die Kardinalitätsbeschränkungen der Rollen aus Abb. 5.4 lauten (*0,**), wobei die *0* eigentlich nur für die ersten oder letzten Personen einer Eltern-Kind-Beziehungskette zutreffen. Die Kardinalität „viele“ (*) wird in der Regel sich elternseitig wohl auf die Zahl *2* beschränken, wobei die Anzahl Kinder einer Person im Prinzip beliebig sein kann. Da es sich hier um eine *n:m*-Beziehung

handelt, wird für die Relation *E_K* wie üblich eine eigene Tabelle benötigt. Die Beschreibung der Attribute könnte so aussehen:

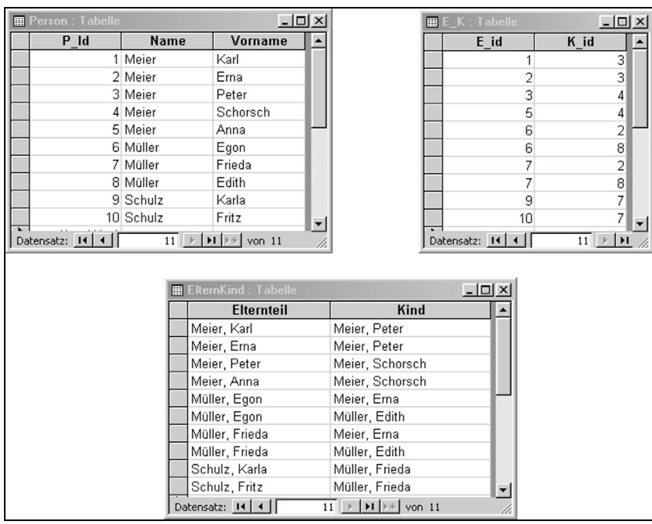
Person

	Primary Key		
Attribut	P_Id	Name	Vorname
Domain	Zahl (Integer)	Text(30)	Text(30)

Relation: *E_K*

	Primary Key	
	Foreign Key	Foreign Key
Fremd-Entität	Person	Person
Fremd-Attribut	P_Id	P_id
Attribut	E_Id	K_id
Domain	Zahl (Integer)	Zahl (Integer)

Hier macht es Sinn, in der Relation *E_K* für die Fremdschlüssel andere Namen zu vergeben als für den Schlüssel der beteiligten Entität (*Person*), denn in der ersten Spalte der Relationentabelle hat *P_Id* die Bedeutung des Elternteils (*E_Id*) und in der zweiten Spalte die Bedeutung des Kindes (*K_Id*). Konkret könnte das Ganze z.B. so aussehen (Abb. 5.5):



Person : Tabelle

P_Id	Name	Vorname
1	Meier	Karl
2	Meier	Erna
3	Meier	Peter
4	Meier	Schorsch
5	Meier	Anna
6	Müller	Egon
7	Müller	Frieda
8	Müller	Edith
9	Schulz	Karla
10	Schulz	Fritz

Datensatz: 11 von 11

E_K : Tabelle

E_id	K_id
1	3
2	3
3	4
5	4
6	2
6	8
7	2
7	8
9	7
10	7

Datensatz: 11

ElternKind : Tabelle

Elternteil	Kind
Meier, Karl	Meier, Peter
Meier, Erna	Meier, Peter
Meier, Peter	Meier, Schorsch
Meier, Anna	Meier, Schorsch
Müller, Egon	Meier, Erna
Müller, Egon	Müller, Edith
Müller, Frieda	Meier, Erna
Müller, Frieda	Müller, Edith
Schulz, Karla	Müller, Frieda
Schulz, Fritz	Müller, Frieda

Datensatz: 11 von 11

Abb. 5.5 Beispiel einer reflexiven Relation

Die beiden oberen Tabellen aus Abb. 5.5 stellen die Tabelle *Person* sowie die Beziehung *E_K* dar, während die untere Tabelle nur noch einmal die Namen und Vornamen der zugehörigen ID's aus der Beziehungstabelle visualisiert. Beispielsweise erkennt man, dass *Peter Meier* einerseits das Kind von *Karl* und *Erna Meier* ist, und selbst Vater von *Schorsch Meier* ist, wodurch angezeigt ist, dass *Karl* und *Erna Meier* die Großeltern väterlicherseits von *Schorsch Meier* sind. Wenn man unterstellt, dass alle Elternteile, welche ein gemeinsames Kind besitzen, auch verheiratet sind, so lassen sich sofort auch alle Ehepaare bestimmen (z.B. *Frieda* und *Egon Müller*). Offenbar ist dann auch *Frieda Müller* die Schwiegermutter von *Karl Meier*. Des Weiteren ist *Edith Müller* die Tante von *Peter Meier* etc.; es lassen sich fast alle Verwandtschaftsverhältnisse aus der Beziehungstabelle entnehmen. Es zeigt sich dadurch, dass reflexive Relationen dazu geeignet sind, Hierarchien über beliebig viele Ebenen hinweg widerzuspiegeln. So können solche Beziehungen z.B. auch eingesetzt werden zur Abbildung von Konzernhierarchien oder Gesamt-Teil-Relationen für Maschinenteile (eine Pumpe kann Teil eines Motors sein, dieser wiederum Teil eines Automobils etc.).

Objektorientierte Ansätze(UML)
Wir hatten bereits über die Definition von Objekten gesprochen. Objekte besitzen eine Datenkapselung, und auf die Daten wird über Methoden zugegriffen. Ähnliche Objekte haben wir als Klasse zusammengefasst und die Exemplare einer Klasse auch Instanzen genannt. Klassen können selbst Instanzen höherer Klassen sein. Für die objektorientierte Vorgehensweise spielt der Klassen- und Objektbegriff naturgemäß eine große Rolle. Im modernen Softwareengineering ist man dazu übergegangen, Datenmodellierungen fast ausschließlich mit objektorientierten Methoden durchzuführen. In der Vergangenheit gab es verschiedene Ansätze, doch mittlerweile hat sich die sog. Unified Modeling Language (UML) als eine der wichtigsten Modellierungsmethoden überwiegend durchgesetzt. Deswegen wird diese Methode nachfolgend auszugsweise behandelt[11].
Im Jahr 1996 veröffentlichten Jim Rumbaugh (maßgeblicher Autor der Object Modeling Technique, OMT), Grady Booch (Autor der Booch Modeling Method, BMM) und Ivar Jacobson (Autor des Object Oriented Software Engineering, OOSE) eine Datenmodellierungsmethode, die sie Unified Modeling Language (UML) nannten. Damit wurde dem Konkurrenzkampf der jeweiligen Methoden dieser Autoren ein Ende gesetzt und ein einheitlicher Standard geschaffen. Dennoch kursieren diverse Dialekte von UML, die sich aber meistens nur in kleineren Abweichungen von den grafischen Standarddarstellungen wiederfinden.

[11] Wir betrachten hier hauptsächlich Klassendiagramme; für eine Besprechung aller 13 UML-Diagrammtypen siehe z.B. Zöller-Greer, P.: *Software-Architekturen: Grundlagen und Anwendungen*, Verlag composia, 2010

Zunächst seien nachfolgend einige grafische Bausteine von UML beschrieben. Ein UML-Klassen-Diagramm besteht aus einer Anzahl Graphen verbundener Knoten, wie z.B. Rechtecke. Die Größe dieser Graphen untereinander und ihre relative Position spielen dabei keine Rolle. Es sind nur zweidimensionale Graphen erlaubt. Es gibt 4 grundlegende graphische Konstrukte:

- Ein *Icon* ist ein Diagramm mit fester Größe und Form. Es kann mit anderen Symbolen durch Pfade verbunden oder auch in einem Symbol enthalten sein.
- *Zweidimensionale Symbole* (z.B. Rechtecke) haben die Eigenschaft, dass sie beliebig vergrößert werden können um ihren Inhalt darzustellen. Diese Symbole enthalten in der Regel weitere Unterteilungen.
- Ein *Pfad* ist ein Liniensegment, an dessen beiden Enden eines der anderen Konstrukte steht. Ist dies ein Icon, wird der Icon auch Terminator genannt. Es ist möglich, dass mehrere Pfade mit demselben Symbol verbunden sind, oder dass sie zu einem Pfad kombiniert werden.
- Ein *String* enthält die verschiedensten Arten von Information in einer nicht formalisierten Weise. Strings sind gewöhnlich in Textform und sollen andere Konstrukte erweitern. Das können z.B. Namen sein, um ein Symbol oder einen Pfad zu identifizieren. Strings können in geschweifte Klammern ({}) geschrieben werden.

Neben diesen vier Grundkonstrukten unterscheidet man noch drei Arten der Darstellung von Beziehungen zwischen ihnen: *Verknüpfungen*, also die Verbindung zweier Konstrukte mit einem Pfad; *Inhalte*, das sind Konstrukte innerhalb eines anderen Konstruktes, und schließlich *Attachments*, das sind Konstrukte oder Symbole, die nah beieinander liegen (wie z.B. die Beschriftung eines Pfades).
Gelegentlich findet man in UML-Diagrammen auch Notizen und Kommentare. Diese sind jedoch nicht Bestandteil der zu entwickelnden Anwendung und dienen sozusagen nur den „Menschen“ als Zusatzinformation. So eine Notiz wird dargestellt durch ein Rechteck, dessen rechte obere Ecke „eingefaltet“ ist. Sie werden durch eine gestrichelte Linie mit den anderen UML-Konstrukten verbunden.

Abb. 5.6 Darstellung einer Notiz in UML

Klassendiagramme
Die Notation einer Klasse innerhalb des UML-Diagramms ist durch ein Rechteck gegeben, welches maximal drei Unterteilungen enthält: Im oberen Teil steht der Name der jeweiligen Klasse, im mittleren Teil steht eine Liste der Attribute der Klasse und im unteren Teil stehen schließlich die Methoden, welche diese Klasse zur Verfügung stellt (werden auch manchmal *Klassenoperationen* genannt). Der mittlere und untere Teil kann auch leer bleiben oder ganz wegfallen.

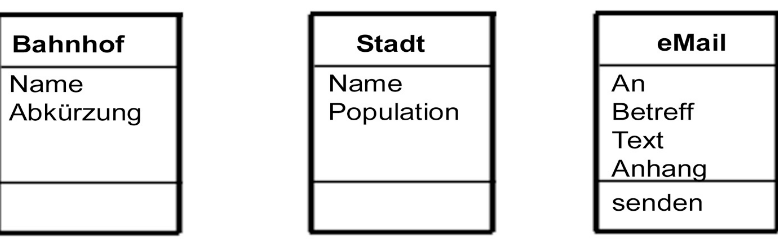


Abb. 5.7 Beispiele für Klassen

Instanzendiagramme
Es ist auch möglich, einzelne Instanzen der Klassen in UML darzustellen. So eine Darstellung heißt dann *Instanzendiagramm*. Die Werte der Instanzen werden unterhalb des jeweiligen Objektnamens aufgelistet.
Die Notation der Objekte in einem Instanzendiagramm ist ähnlich wie die bei Klassen:

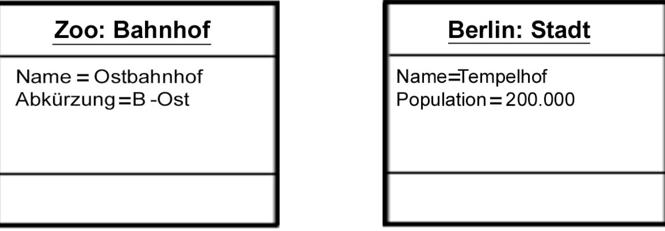


Abb. 5.8 Instanzendiagramm

In einem Instanzendiagramm werden wie beim Klassendiagramm die Objektattribute unterhalb des Klassennamens/Objektnamens aufgelistet. Die

Werte der Objektattribute stehen hinter dem Attributnamen, getrennt durch ein Gleichheitszeichen. Der Objektname steht links vor dem Klassennamen durch einen Doppelpunkt getrennt und beides ist unterstrichen. Als Analogie kann man sich merken: Objekte verhalten sich zu Klassen wir die Werte zu den Attributen. Für die Datenmodellierung spielen aber die Klassendiagramme die größere Rolle, Instanzdiagramme werden nur dann benutzt, wenn z.B. ein Szenario betrachtet werden soll. Objekte haben Identität, Werte nicht. Grundsätzlich kann man UML-Diagramme bereits in der Analysephase einsetzen, aber der Schwerpunkt liegt gewiss in der Entwurfsphase. In der Analysephase kennt man u.U. noch gar nicht alle Attribute und läßt daher üblichweise identifizierende Schlüsselattribute weg. In der Entwurfsphase jedoch werden oft alle Attribuite eines Objekts aufgelistet werden, auch die Schlüsselattribute:

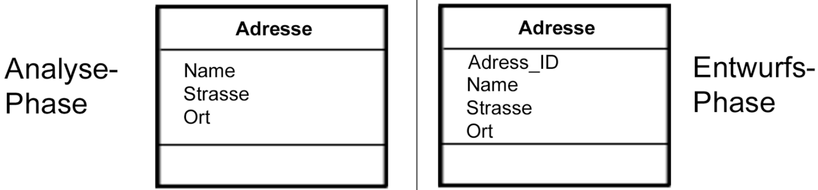


Abb. 5.9 Objektattribute in der Analysephase und in der Entwurfsphase

Operationen und Methoden

Eine Funktion oder Prozedur, die auf oder von Objekten einer Klasse angewendet wird, hatten wir *Methode* (bzw. *Operation*) genannt. Solche Methoden werden im unteren Teil des Klassensymbols aufgelistet. Der Name einer Methode kann dabei gefolgt sein von optionalen Details wie z.B. einer Argumentenliste oder der Angabe des Datentyps des Rückgabewerts. Operationen, die auf mehrere Klassen angewendet werden, heißen *polymorph*.

Häufig wird nicht zwischen den Begriffen *Operation* und *Methode* unterschieden und sie werden synonym verwendet. Genau genommen jedoch versteht man unter einer Methode die *Implementation* der jeweiligen Operation durch eine Funktion. Eine Methode ist also immer durch eine Software repräsentiert, was die Operation (noch) nicht sein muss. Außerdem kann es sein, dass *eine* polymorphe Operation *verschiedene* Methoden implementiert hat. Angenommen, eine Klasse „Equipment" besitzt eine Operation „kalkuliereKosten". Diese Operation sei polymorph und die zugehörigen Methoden können voneinander verschieden sein; die Berechnung der Kosten z.B. einer Pumpe kann eine andere Formel benutzen wie die Berechnung eines Tanks. Alle diese Methoden führen logisch gesehen dieselbe Task aus, können aber durch verschiedene Programme

realisiert sein. Objektorientierte Software wählt automatisch die passende Methode aus, um die jeweilige Operation gemäß der Klasse des Zielobjektes zu berechnen.

Links und Assoziationen
Neben den zweidimensionalen Klassensymbolen kommen im UML-Diagramm auch die Beziehungen zwischen den Klassen durch Pfade zum Ausdruck. Solche Beziehungen können nun verschiedenster Art sein. Zwei wichtige Beziehungstypen sind Links und Assoziationen.

Definition 5.8 (**Link**)
Ein Link ist eine physische oder konzeptionelle Verbindung zwischen Objekten. In der Regel werden mit einem Link zwei Objekte verbunden, es können aber auch drei oder mehrere sein.

Über den Begriff des Links kann man nun Assoziationen definieren.

Definition 5.9 (**Assoziation**)
Eine *Assoziation* ist eine Beschreibung einer Gruppe von Links, welche eine gemeinsame Struktur und Semantik besitzen.

Damit stellt ein Link also eine Instanz einer Assoziation dar. Links verbinden auf Instanzenebene i.d.R. genau zwei Obkekte miteineander. Im Beispiel der Mitarbeiter, die an Projekten arbeiten, haben wir auf Instanzenebe z.B. den Mitarbeiter Meier, der am Projekt A arbeitet, durch einen Link verbunden. Herr Meier kann auch an Projekt B arbeiten, dann haben wir wieder genau zwei Objekte verbunden. Die Menge der Links wird dann zu einer Assoziation zusammengefasst.
Den Relationen zwischen Entitäten aus der ER-Modellierung entsprechen hier die Assoziationen zwischen Klassen.

Die nachfolgenden Abbildungen illustrieren die Schreibweisen für Links und Assoziationen. Eine Assoziation wird dargestellt als Linie, welche die beiden Klassen verbindet. Wie üblich werden die Kardinalitätsbeschränkungen an die Klassen geschrieben (das Symbol * bedeutet „viele“, siehe unten). Es ist bei UML-Diagrammen häufig anzutreffen, dass sich Kardinalitäten anstatt auf die Rollen manchmal auf die sog. inversen Rollen beziehen. Manchmal ist das aber nicht sofort zu ersehen. Es sollte daher auf eine klar durchgehaltene Beschriftung geachtet werden, und es ist zudem sinnvoll, in der Dokumentation anzugeben, auf welche Art der Rollenbeschriftung man sich geeinigt hat; denn

bei den inveresen Rollen sind die Kardinalitätsbeschränkungen gegenüber den Rollen gerade vertauscht (obwohl das gleiche gemeint ist).

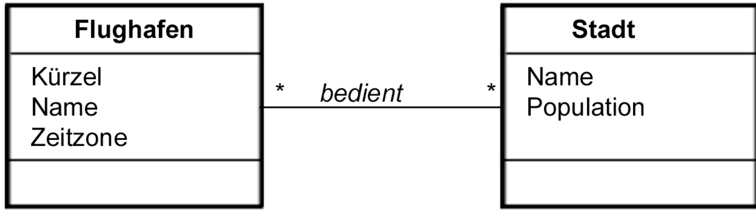


Abb. 5.10 Assoziation

Ein Link wird dargestellt durch eine Line, welche die zueinander in Beziehung stehenden Objekte verbindet. Eine "Linie" kann dabei aus mehreren Liniensegmenten bestehen. Zum Beispiel bedienen in nachfolgender Abb. 5.11 die Flughäfen SEA und BOE beide die Stadt Seattle.

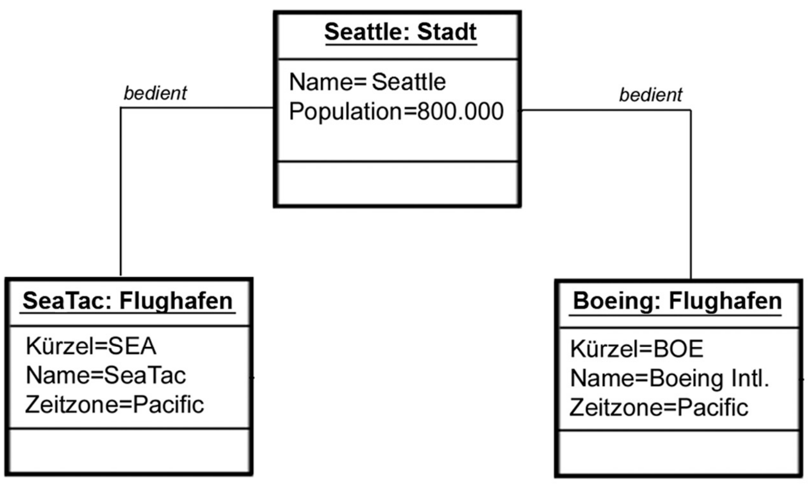


Abb. 5.11 Links

Für die Kardinalitätsbeschränkungen der Assoziationen gilt (Beschriftung der Pfade):

Assoziationstyp	**UML**
genau ein	keine Beschriftung oder *1*
viele (kein oder mehr)	*
kein oder ein	*0..1*
ein oder mehr	*1..**
geordnet viele	*ordered **
numerisch (z.B. 2,3 und 4)	*2..4*

Betrachten wir wieder unser Beispiel aus Abb. 5.3. Als UML-Diagramm sieht es folgendermaßen aus:

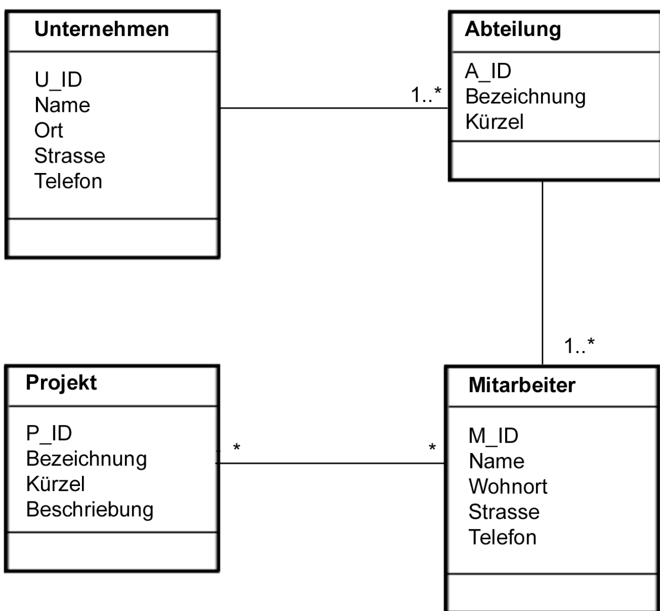


Abb. 5.12 Klassendiagramm

Die Assoziation zwischen den Klassen *Projekt* und *Mitarbeiter* stellt eine *viele-viele*-Assoziation dar. Dabei kann natürlich zwischen einem Objektpaar maximal ein einziger Link pro Assoziation bestehen. Es können aber mehrere Assoziationen zwischen Klassen bestehen. In diesem Fall sollten die Rollenbeschriftungen nicht fehlen:

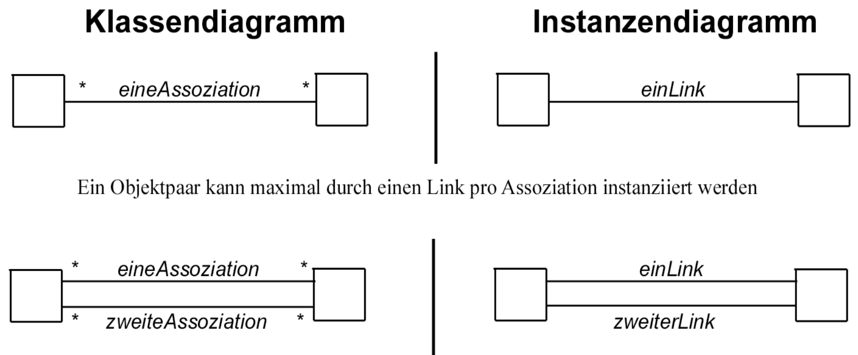


Abb. 5.13 Mehrfach-Assoziationen

Rollennamen werden manchmal auch Pseudoattribute genannt.
In Abb. 5.12 sind nur die Entitäten mit ihren Attributen eingezeichnet. Nun gibt es aber Attribute, welche als Eigenschaft einer Assoziation angesehen werden können.

Definition 5.10 (**Link-Attribut**)
Ein *Link-Attribut* ist eine Eigenschaft einer Assoziation, welche einen Namen hat und für jeden Link der Assoziation einen Wert besitzt.

Linkattribute werden in einem Klassensymbol ohne Klassennamen eingetragen, wenn es sich um eine *1:n*-Assoziation handelt, und das Klassensymbol wird durch eine gestrichelte Linie, die senkrecht zur Assoziation steht, eingetragen. Am Ende der gestrichelten Linie zeigt manchmal zusätzlich ein ausgefüllter Pfeil auf das Klassensymbol. Hat man nun eine *n:m*-Assoziation und Linkattribute, so erzeugt man eine eigene „Assoziationsklasse" mit eigenen Namen.

Definition 5.11 (**Assoziationsklasse**)
Eine *Assoziationsklasse* ist eine Assoziation, deren Link-Objekte in einer eigenen Klasse untergebracht sind und deren Links an weiteren Assoziationen teilnehmen können.

In Abb. 5.14 sieht man, dass z.B. das Attribut „Tätigkeit" ein Link-Attribut der Assoziationsklasse M_P darstellt, denn das Attribut Tätigkeit kann weder dem

Mitarbeiter selbst noch dem Projekt zugeordnet werden (es kann z.B. sein, dass der gleiche Mitarbeiter in verschiedenen Projekten verschiedene Tätigkeiten ausübt).

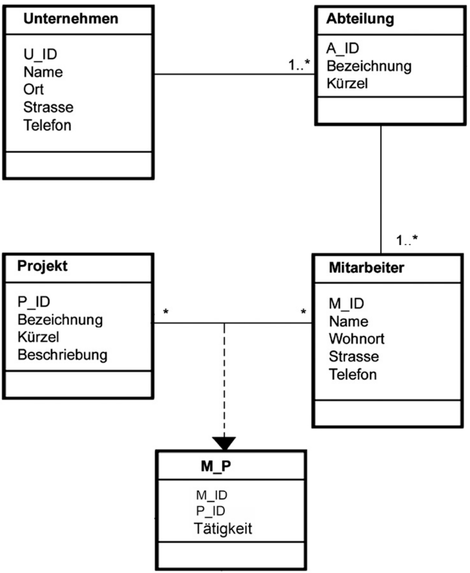


Abb. 5.14 Link-Attribute in Assoziationsklassen

Bei der relationalen Datenmodellierung sahen wir, dass *n:m*-Relationen durch eigene Tabellen realisiert wurden. In diese werden naturgemäß alle die Attribute gepackt, die auch ausschließlich für die Beziehung zuständig sind. Beziehungstabellen, die *nur* Fremdschlüsselattribute enthalten, würde man in ein UML-Diagramm normalerweise nicht eintragen. Erst wenn es Link-Attribute gibt, sollte man eine eigene Assoziationsklasse dafür einzeichnen.
Natürlich sind auch in UML reflexive Beziehungen darstellbar (Abb. 5.15):

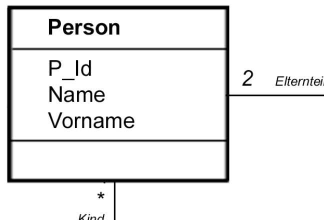


Abb. 5.15 Klassendiagramm mit reflexiver Beziehung

Auch hier kann bei Bedarf die Assoziation als Link-Attribut oder Assoziationsklasse eingezeichnet werden.

Generalisierung, Spezialisierung und Vererbung

Die objektorientierte Methode ist besonders gut dafür geeignet, einen Sachverhalt zu realisieren, der Generalisierung genannt wird. Dabei handelt es sich um eine Modellierung, bei der redundante Attribute und/oder Methoden in Klassen herausgenommen und einer einzigen, „generellen“ Klasse zu eigen gemacht werden.

> *Definition 5.12* (**Generalisierung**)
> Unter *Generalisierung* versteht man die Beziehung zwischen einer Klasse (der Super- oder Basis- oder Oberklasse) und einer oder mehreren Variationen (auch abgeleitete Klassen oder Subklassen oder Unterklassen genannt) dieser Klasse. Die Sichtweise ist dabei von unten nach oben: von den abgeleiteten Klassen wird nach oben zur Superklasse verallgemeinert.

Der Begriff „Variation“ bedeutet hier, wie bereits angedeutet, dass redundante Attribute und Methoden in den Subklassen aus deren Darstellung herausgenommen und in die Basisklasse verlegt werden. Generalisation organisiert Klassen durch ihre Ähnlichkeiten bzw. Differenzen, wobei eine Strukturierung der Objekte erfolgt. Die Basisklasse enthält die gemeinsamen Attribute und Methoden, aber auch evtl. Zustandsdiagramme und Assoziationen. Die abgeleiteten Klassen enthalten nur noch die spezifischen Attribute, Methoden, Zustandsdiagramme und Assoziationen. Es können dabei mehrere Ebenen von generalisierten Beziehungen auftreten. Eine Instanz einer abgeleiteten Klasse ist gleichzeitig eine (transitive) Instanz all ihrer Superklassen.

> *Definition 5.13* (**Spezialisierung**)
> Unter dem Begriff Spezialisierung versteht man den gleichen Sachverhalt wie bei der Generalisierung, jedoch ist die Perspektive von der Basisklasse auf die Subklassen gerichtet.

Während die Generalisierung also eine Bottom-Up-Perspektive darstellt, beginnend mit den Subklassen nach oben zur generelleren Superklasse aufschauend, wo Gemeinsamkeiten abstrahiert werden, so stellt die Spezialisierung eine Top-Down-Perspektive dar, beginnend mit der Superklasse, welche sich in ihre Variationen aufsplittet. Modellierungstechnisch und auch im UML-Diagramm besteht allerdings kein sichtbarer Unterschied, es handelt sich genau um den gleichen Sachverhalt. Einfache Generalisierung erzeugt damit eine Hierarchie innerhalb von Klassen. Jede Subklasse hat eine unmittelbare Superklasse. Im

UML-Diagramm werden Generalisierungen durch einen hohlen Pfeil dargestellt, welcher mit der Spitze direkt die Superklasse berührt und an deren Ende die Verbindungslinie zu den Subklassen startet. Man kann jede Subklasse mit einer eigenen Verbindungslinie und mit Pfeil ausstatten, jedoch ist es übersichtlicher, die abgeleiteten Klassen zu gruppieren und einen Baum zu erzeugen.
Abb. 5.16 zeigt ein typisches Beispiel für eine Generalisierung. Alle Geldanlagen besitzen einen Namen und einen momentanen Wert in einer bestimmten Währung. Aktien haben eine vierteljährliche Dividende, welche vom Vorstand festgesetzt wird. Pfandbriefe haben ein Fälligkeitsdatum, einen Fälligkeitswert und entweder eine feste oder variable Verzinsungsrate. Dann gibt es unterschiedliche Arten von Versicherungen wie Lebensversicherungen, Krankenversicherungen etc.; natürlich ist dafür ein Jahresbeitrag zu entrichten.

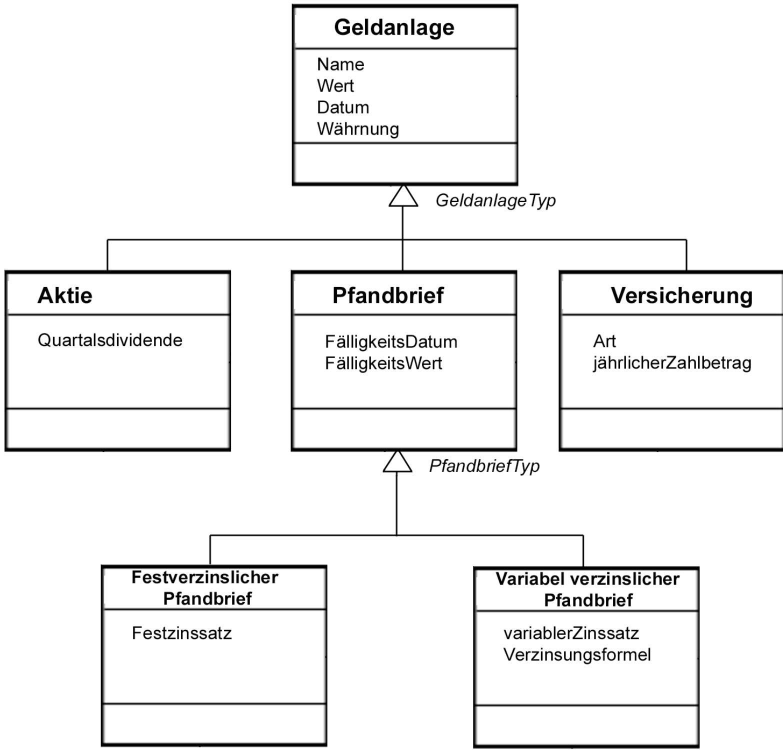


Abb. 5.16 Generalisierung am Beispiel Geldanlagen

Neben den hohlen Pfeilen in Abb. 5.16 steht noch eine optionale Bezeichnung. In objektorientierten Datenbanksystemen spielt diese keine besondere Rolle, doch in relationalen Datenbanksystemen lassen sich Generalisierungen nicht direkt implementieren. Dann macht es Sinn, daraus Relationen zu erstellen. So könnte eine Tabelle mit den Attributen der Klasse *Geldanlage* erzeugt werden und drei Tabellen mit jeweils den Attributen der Klassen *Aktie*, *Pfandbrief* und *Versicherung*, welche dann zu der Tabelle der Klasse *Geldanlage* jeweils in einer (1,1):(0,1)-Beziehung stehen. Damit aber klar ist, welche Objekte der jeweiligen drei Subklassen-Tabellen zu welchem Objekt der Geldanlage-Tabelle gehören, muss eine eindeutige Identifizierung vorhanden sein. Diese Aufgabe können –wie immer- Schlüsselattribute übernehmen. In der Tabelle der Basisklasse empfiehlt sich unter Umständen, ein oder mehrere Attribut(e) hinzuzunehmen, welche den Typ der Geldanlage angeben (z.B. bei nur einem Attribut den Wert *1* für Aktien, dem Wert *2* für Pfandbriefe und dem Wert *3* für Versicherungen; bei mehreren Attributen könnte jedes davon für eine Subklasse stehen und jeweils den Boole'schen Wert 0 bekommen, wenn kein Objekt in der entsprechenden Unterklasse vorhanden ist und eine 1 wenn ein zugehöriges Objekt in der Unterklasse vorhanden ist). Hierfür eignet sich dann das/die Attribut(e), welche(s) neben dem Generalisierungspfeil steht(stehen). Da diese Attribute die „Unterscheidung" zwischen den abgeleiteten Klassen darstellen, heißen sie Diskriminatoren oder Diskriminator-Attribute.
Eine weitere Spezifizierung für eine Generalisierung können zwei Angaben sein, welche in geschweiften Klammern ebenfalls in die Nähe des Generalisierungspfeils geschrieben werden. Dabei handelt es sich umfolgendes:

complete/incomplete:
Wenn die Exemplarmenge einer Basisklasse nur aus Exemplaren der Unterklassen aufgebaut ist, so heißt die Spezialisierung komplett oder vollständig (engl. complete), andernfalls unvollständig (engl. incomplete).
Eine vollständige Spezialisierung hat demnach nur Instanzen in ihren Subklassen. Der nächste Begriff unterscheidet, ob gleiche Objekte in mehreren abgeleiteten Klassen einer Basisklasse vorkommen können oder nicht.

disjoint/overlapping:
Wenn die Exemplarmengen der Unterklassen paarweise verschieden sind, so nennt man die Spezialisierung bzw. die Subklassen disjunkt (engl. disjoint), andernfalls überlappend (engl. overlapping).

Betrachtet man z.B. eine Basisklasse *Person* und die Subklassen *Kunde* und *Lieferant* (vgl. Abb. 5.17, die Attribute wurden weggelassen). Dies wäre dann ein Beispiel für eine überlappende Spezialisierung, da eine Person sowohl Kun-

de als auch Lieferant sein kann. Außerdem wäre die Spezialisierung dazu noch unvollständig, denn es kann Personen geben, die weder Kunde noch Lieferant sind (z.B. Privatpersonen).

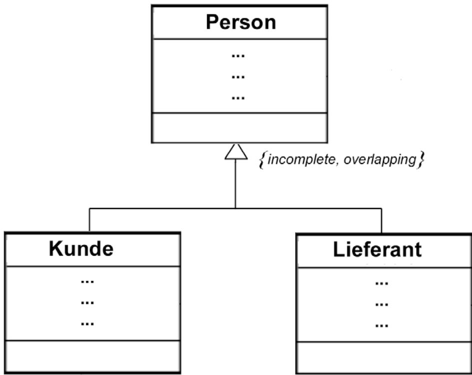


Abb. 5.17 Unvollständige, überlappende Spezialisierung

Man kann die Vollständigkeit jedoch erzwingen, in dem man weitere Subklassen hinzunimmt (in Abb. 5.17 z.B. eine Subklasse *Privatperson*). Im Fall einer vollständigen Spezialisierung besitzt eine Basisklasse selbst gar keine direkten Instanzen mehr. Solche Klassen nennt man abstrakte Klassen.

Definition 5.14 (**Abstrakte Klasse**)
Eine Klasse ohne direkte Instanzen heißt abstrakt, andernfalls konkret.

Abstrakte (Basis-)Klassen sind also immer das Ergebnis einer Spezialisierung vom Typ *complete*. Die Basisklassen *Geldanlage* und *Pfandbrief* aus Abb. 5.16 sind Beispiele für abstrakte Klassen. Hier ist also die Spezialisierung vollständig. Die Unterklassen *Aktie*, *Pfandbrief* und *Versicherung* sind hier disjunkt, da keine ihrer Instanzen gleichzeitig Exemplar einer der anderen Unterklassen sein kann. Es sei noch ein weiterer wichtiger Begriff im Zusammenhang mit der Generalisierung/Spezialisierung erwähnt.

Definition 5.15 (**Vererbung**)
Unter Vererbung wird der Mechanismus verstanden, welcher die Attribute, Methoden, Zustandsdiagramme und Assoziationen einer Basisklasse seinen abgeleiteten Klassen zur Verfügung stellt.

Grundsätzlich ist es noch möglich, dass eine Subklasse Eigenschaften von mehreren Basisklassen ererbt. Dieser Sachverhalt wird *Mehrfachvererbung* genannt. Der Vorteil dieser komplizierten Form der Generalisierung besteht darin, dass man eine höhere Flexibilität bei der Spezifikation der Klassen hat und dass bessere Wiederverwendungsmöglichkeiten bestehen. Ein Beispiel einer Mehrfachvererbung zeigt Abb. 5.18.

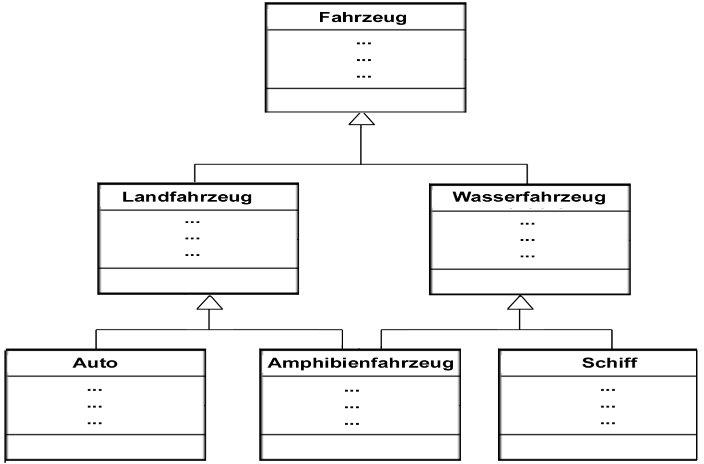


Abb. 5.18 Mehrfachvererbung

Vererbungen lassen sich nicht direkt in relationalen Schemata darstellen. Grundsätzlich sollte man sie bei einer relationalen Umsetzung als (1,1):(0,1) – Beziehungen modellieren. Der Diskriminator kann dabei als Attribut eingeführt werden, so dass eine genaue Identifikation der jeweiligen Subklassen in den Tabellen möglich ist.
Man kann Vererbungen auch mittels Assoziationen darstellen. Dann wird die Basisklasse zur 1-Seite der Assoziationen und die Subklassen jeweils zu einer (0,1)-Seite, es handelt sich dann also immer um (1,1):(0,1)-Assoziationen zwischen Basisklasse und jeder Subklasse. Damit kann man dann sofort relationale Tabellen nebst deren Beziehungen daraus machen.
Eine wichtige Rolle bei der UML-Modellierung spielt ein weiterer Beziehungstyp, die sog. Aggregation. Sie ermöglicht es, eine Gesamtheits-Teile-Beziehung grafisch darzustellen. Die Aggregation darf auf keinen Fall mit der Vererbung bzw. Generalisierung/Spezialisierung verwechselt werden; es handelt sich dabei

um einen völlig anderen Sachverhalt, nämlich um einen Spezialfall der Assoziation.

Definition 5.16 (**Aggregation**)
Unter einer *Aggregation* versteht man eine Assoziation zwischen einer Klasse von Objekten, die eine Komponentengruppe bilden und einer oder mehrerer anderer Klassen, deren Objekte die Komponenten bilden.

Aggregationen sind also Ganzheits-Teile-Assoziationen, welche in mehreren Ebenen vorkommen können. Aggregationen besitzen die Transitivitätseigenschaft: Wenn A Teil von B ist und B Teil von C, dann ist A auch Teil von C. Zudem sind Aggregationen antisymmetrisch: Wenn A Teil von B ist, dann ist B nicht Teil von A. In UML-Diagrammen werden Aggregationen wie Assoziationen dargestellt, wobei wie bei der Generalisierung eine Baumstruktur verwendet wird und die Verbindungslinie von einem kleinen Diamant ausgeht, welcher die Gesamtheitsklasse berührt. Eine Aggregatsbeziehung ist im Wesentlichen eine binäre Assoziation, d.h. eine Paarbildung zwischen der Komponentengruppenklasse und einer Komponentenklasse. Daher korrespondiert eine Komponentengruppenklasse, die mehrere Komponentenklassen besitzt, mit mehreren Aggregationen. Jede Paarbildung definiert also eine Aggregation, so dass die Multiplizität jeder Komponente innerhalb der Komponentengruppe spezifiziert werden kann.
In Abb. 5.19 stellt die Klasse *PC* eine Komponentengruppe (Gesamtheit) dar. Ein *PC* besteht physikalisch aus einem oder mehreren *Monitoren*, einer *Systemeinheit*, keiner oder einer *Maus* sowie einer *Tastatur*. Die Komponente *Systemeinheit* wiederum besteht aus einem *Gehäuse*, einer *CPU*, viel *RAM* und keinem oder einem *Lüfter* zur Kühlung. Anders als bei der Spezialisierung, wo die Eigenschaften einer Basisklasse durch ihre abgeleiteten Klassen verfeinert werden, wird hier die Klasse einer Komponentengruppe in ihre Bestandteile zerlegt. Es gibt daher natürlich auch kein Vererbungsmechanismus bei Aggregationen. Zur Bestimmung, ob eine Assoziation eine Aggregation ist oder nicht, mag es hilfreich sein, für die potenziellen Komponentenklassen die „ist-ein-Bestandteil-von“-Frage zu stellen. Im Zweifelsfall sollte die in Frage kommende Beziehung zunächst einfach als normale Assoziation modelliert werden.
Während die abgeleiteten Klassen einer Basisklasse in einer „oder“-Beziehung zueinander stehen (eine Geldanlage ist eine Aktie *oder* eine Pfandbrief *oder* eine Versicherung), stehen die Klassen einer Assoziation, also auch die Komponentenklassen einer Komponentengruppenklasse, in einer „und“-Beziehung (ein PC besteht aus Monitor *und* Systemeinheit *und* Maus *und* Tastatur).

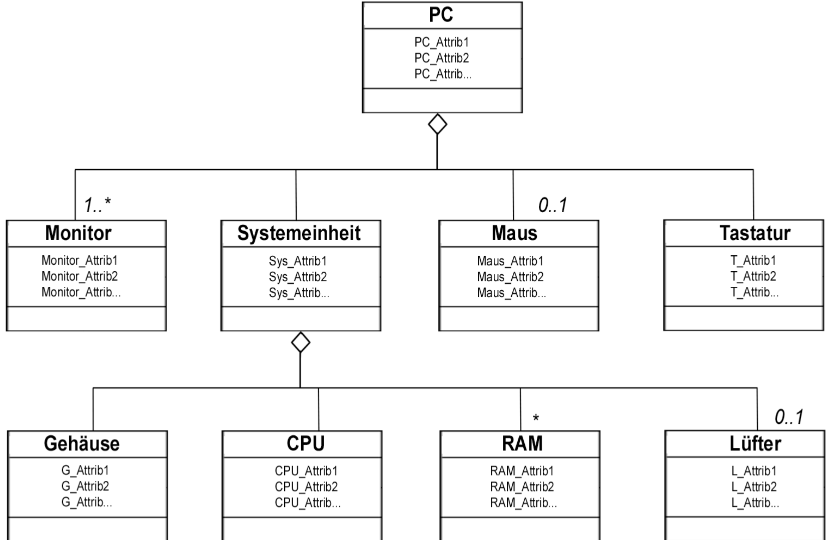


Abb. 5.19 Aggregationen

Manchmal wird noch zwischen *physikalischer Aggregation* und *Katalog-Aggregation* unterschieden. Die physikalische Aggregation bezeichnet eine Aggregation, bei der jedes Objekt einer Komponentenklasse maximal einem einzigen Objekt der Komponentengruppenklasse zugeordnet ist. Demgegenüber ist bei der Katalog-Aggregation ein Objekt einer Komponentenklasse für mehrere Komponentengruppenklassen nutzbar (z.B. eine Schraube als Instanz einer Komponentenklasse kann in mehreren Komponentengruppenklassen wie z.B. Motoren, Möbel etc. benötigt werden).
Ein Spezialfall der Aggregation ist die *Komposition*. Diese wird benutzt, wenn die Teilklassen „lebensnotwendiger“ Bestandteil der Gesamtheit sind. Man zeichnet dann anstatt der hohlen Raute bei der Gesamtheitsklasse eine ausgefüllte Raute.

Realzeitsysteme
Die bereits bekannten Methoden wie die Erstellung von Entity-Relationship-Diagrammen (ERD), Datenflusskontrolldiagrammen oder Zustandsdiagrammen werden natürlich auch für die Planung von Echtzeitsystemen eingesetzt. Man unterscheidet zunächst analog zu den Datenbankanwendungen zwischen einer Phase der strukturierten Analyse und einer Entwurfsphase.

Echtzeit-Strukturierte-Analyse (RTSA)
Diese auch als Real-Time Structured Analysis bezeichnete Methode, soll die Anforderungen des Echtzeitsystems ermitteln. Dabei wird folgendermaßen vorgegangen:

1. *Entwicklung eines System-Kontext-Diagramms*. Das System-Kontext-Diagramm definiert die Abgrenzungen des Systems zu seiner Umgebung. Es sollten dabei auch alle wichtigen Eingaben und Ausgaben des Systems zu sehen sein.
2. *Zerlegung des Datenflusskontrolldiagrams*. Das Datenflusskontrolldiagramm zeigt in hierarchischer Form die beteiligten Funktionen, auch Transformationen oder Prozesse genannt, und ihre Schnittstellen sowie den Fluss der Daten. Die Zerlegung soll die Funktionen und die Datenflüsse trennen, so dass z.B. die Daten durch entsprechende Datenstrukturbeschreibungen (Data Repository) spezifiziert sind. Die reinen Datenstrukturen und Datenflüsse werden im Data Dictionary abgelegt. Das Ganze wird auch Boeing-Hatleys-Methode genannt. Ein anderer Ansatz, die sog. Ward-Mellor-Methode, sieht vor, dass mit einer Ereignis-Liste begonnen wird (eine Liste, die alle möglichen Inputs enthält) und zu jedem Ereignis das Systemverhalten (d.h. der jeweilige Output) bestimmt wird. Da das Ergebnis vom Zustand des Systems abhängen kann, sind ein oder mehrere Zustandsdiagramme zu entwerfen.
3. *Erarbeitung von Steuerungstransformationen bzw. – Spezifikationen*. Der wichtigste Punkt der Erweiterung der strukturierten Analyse auf Echtzeitsysteme besteht in der Betrachtung von Steuerungen (Controls). Damit soll das Verhalten des Systems beschrieben werden. Dies kann durch die Einführung sog. endlicher Zustandsmaschinen erreicht werden, welche letztlich Zustandsdiagramme oder - tabellen darstellen. Jedes Zustandsdiagramm zeigt verschiedene Zustände des Systems oder Subsystems. Weiter wird das Eingabeereignis bzw. die Eingabebedingung gezeigt, welche Zustandsänderungen verursacht, sowie die Ausgaben, welche durch die Zustandsänderungen hervorgebracht werden.
4. *Definition von Mini-Spezifikationen (Prozess-Spezifikationen)*. Jede Transformation in einem Datenflusskontrolldiagramm ist durch eine Mini-Spezifikation beschreibbar. Man bedient sich dabei in der Regel eines Pseudocodes.
5. *Data Dictionary*. Eine Zusammenfassung aller vorkommenden Datenstrukturen und Beziehungen muss ebenso vorhanden sein.

Echtzeit-Design

6. *Zuweisung von Transformationen zu Prozessoren*. Die RTSA-Transformationen werden jetzt entsprechenden Prozessoren auf dem Zielsystem zugewiesen. Falls erforderlich, muss für jeden Prozessor das jeweilige Datenflussdiagramm neu gezeichnet werden.
7. *Zuweisung von Transformationen zu Tasks*. Die Transformationen jedes Prozessors sind mit entsprechenden Tasks verknüpft. Jede Task repräsentiert dabei ein Teilprogramm.
8. *Strukturiertes Design*. Im Rahmen der Echtzeitanwendungsentwicklung unterscheidet man zwei Strategien: Die Transformationsanalyse sowie die Transaktionsanalyse. Die Transformationsanalyse bildet die Datenflussdiagramme in Struktogramme ab, wobei hauptsächlich der Eingabe-Prozess-Ausgabe-Fluss dargestellt wird. Das Ganze wird aus der funktionalen Struktur der Spezifikation abgeleitet. Die Eingabebereiche, die zentralen Transformationen sowie die Ausgabebereiche werden aus dem Datenflussdiagramm identifiziert und als separate Bereiche im Struktogramm dargestellt. Die Transaktionsanalyse bildet ebenfalls das Datenflussdiagramm in ein Struktogramm ab, hier werden jedoch die verschiedenen Transaktionstypen betrachtet. Die Funktionen jedes Transaktionstyps werden identifiziert und in einzelne Bereiche im Struktogramm übergeführt.

Einige der genannten Phasen bzw. Diagramme sollen an einem Beispiel erläutert werden. Hierbei handelt es sich um die Planung eines Fahrstuhlkontrollsystems. Dieses System kontrolliert einen oder mehrere Fahrstühle, wobei die Fahrstühle optimal auf Passagieranforderungen reagieren sollen („Elevator-Scheduling"). Außerdem sollen die Bewegungen der Fahrstühle gesteuert werden. Es sei angemerkt, dass es verschiedene Ausprägungen der jeweiligen Design-Strategien gibt, auf die wir jedoch hier nicht näher eingehen.

In den Diagrammen für Echtzeitanwendungen werden spezielle grafische Symbole benutzt. Abb. 5.20 zeigt die für die nachfolgenden Diagrammbeispiele benutzte Nomenklatur.

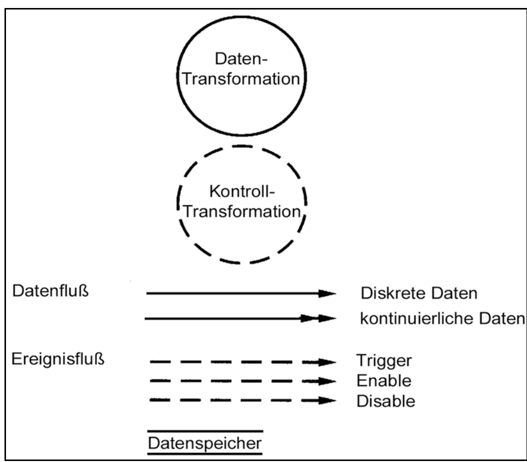


Abb. 5.20 Notation in RTSA-Diagrammen

Zunächst sei das System-Kontext-Diagramm für das Beispiel des Fahrstuhlkontrollsystems betrachtet.

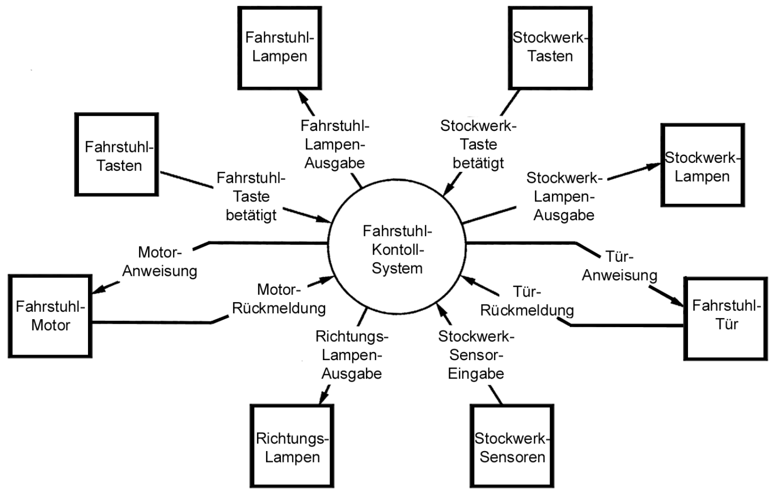


Abb. 5.21 System-Kontext-Diagramm

Diese Abbildung zeigt das System-Kontext-Diagramm, speziell die externen Entitäten bzw. deren Schnittstellen. Jede externe Schnittstelle ist hier durch einen Terminator in Form eines Rechtecks repräsentiert. Zu jedem Fahrstuhl existiert eine Menge Fahrstuhl-Schalter („Knöpfe") sowie eine Menge Fahrstuhllampen, die anzeigen, auf welchem Stockwerk sich der Fahrstuhl gerade befindet. Dann gibt es einen Fahrstuhlmotor, der von bestimmten Befehlen gesteuert wird, z.B. hochfahren, runterfahren, starten und anhalten. Die Fahrstuhltür wird ebenfalls durch bestimmte Steuerbefehle betrieben, und zwar zum Öffnen und Schließen. Auf jedem Stockwerk befinden sich Schalter-Tasten zum Hoch- oder Runterfahrwunsch sowie ein dazugehöriges Paar Anzeigelampen, welche die gewünschte Fahrtrichtung anzeigen. Dann gibt es auf jedem Stockwerk noch ein Paar Lämpchen die anzeigen, in welche Richtung der Fahrstuhl gerade fährt. Natürlich existiert am obersten und untersten Stockwerk jeweils nur eine Lampe. Zusätzlich gibt es noch einen Stockwerkankunftssensor, welcher die Ankunft eines Fahrstuhls auf einem Stockwerk feststellt.
Zu den Hardware-Eigenschaften der I/O-Geräte gehören u.a., dass die Fahrstuhlschalter, Stockwerkslampen und Stockwerkankunftssensoren entsprechend synchronisiert sind. Das heißt, dass ein Interrupt erzeugt werden muss, falls eine Eingabe von einem dieser Geräte erfolgt. Die anderen I/O-Geräte sind alle passiv. Die Fahrstuhl- und Stockwerklampen werden von der Hardware eingeschaltet, müssen aber von der Software ausgeschaltet werden. Die Richtungslampen werden nur von der Software ein- und ausgeschaltet.

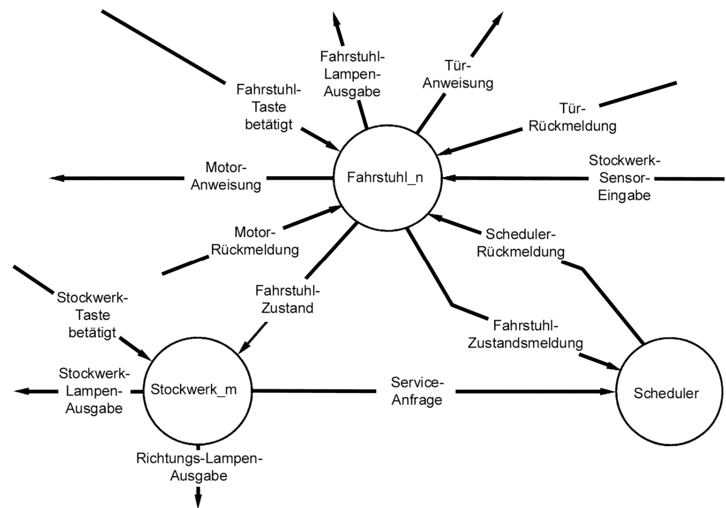


Abb. 5.22 Zerlegung des Gesamtsystems in Subsysteme

In Abb. 5.22 sehen wir die Zerlegung des Gesamtsystems in die drei Teil- oder Subsysteme Fahrstuhl, Stockwerk und Scheduler. Im Allgemeinen ist die Zerlegung eines Systems in Teilsysteme von der Art und Weise der Dienste abhängig, die vom jeweiligen Teilsystem zur Verfügung gestellt werden. In unserem Beispiel ist das Fahrstuhl-Teilsystem ein echtzeitgesteuertes System, während das Stockwerk-Teilsystem Daten sammelt. Beide Systeme sind Aggregationsobjekte des Gesamtsystems im Sinne der objektorientierten Datenverarbeitung. Außerdem stellen sie nicht-abstrakte Klassen dar, denn in der realen Welt gibt es jeweils entsprechende Instanzen dieser Teilsysteme.

Gibt es mehrere Fahrstühle, so müssen diese untereinander koordiniert werden. Wird beispielsweise von einem Stockwerk ein Fahrstuhl angefordert, muss es von einem der Fahrstühle bedient werden; ein Echtzeitkoordinationssystem ist dann erforderlich. In unserem Fall übernimmt der Scheduler diese Funktion. Dabei handelt es sich um eine Software, die entscheidet, welcher Fahrstuhl welches Stockwerk bedienen soll. Hier muss auch koordiniert werden, was passiert, wenn mehrere Passagiere auf verschiedenen Stockwerken Fahrstühle anfordern bzw. wenn innerhalb eines Fahrstuhls mehrere Zielstockwerke einen Haltewunsch haben.

Die beiden Subsysteme Fahrstuhl und Stockwerk müssen ggf. noch weiter zerlegt werden.

Abschließend sei noch das Zustandübergangs-Diagramm (State-Transition-Diagram) der Fahrstuhlkontrolle angegeben.

Hier werden die jeweiligen Zustandsänderungen grafisch dargestellt. Es sollte dabei darauf geachtet werden, dass solche Darstellungen möglichst vollständig sind, d.h. es darf kein „undefinierter“ Zustand eintreten.

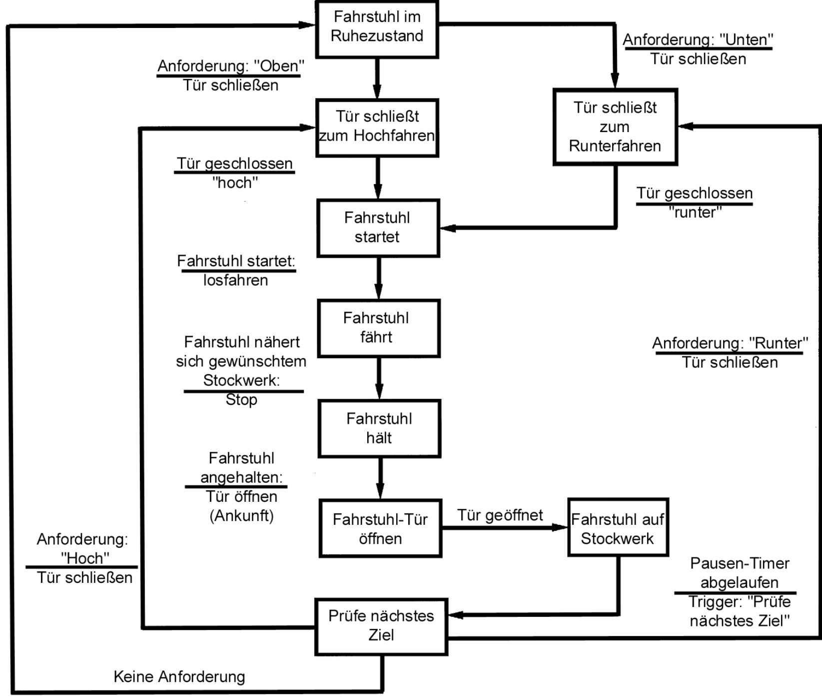
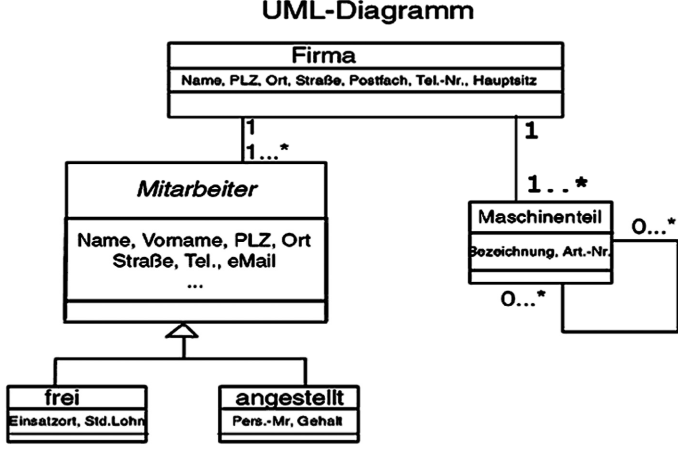


Abb. 5.23 Zustandübergangs-Diagramm des Fahrstuhlkontrollsystems

Übungen zum Selbsttest

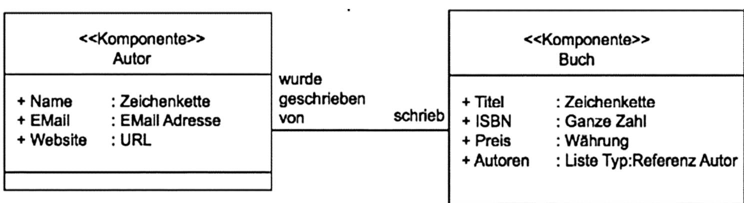
1. Es sei folgender Sachverhalt eines Bankgeschäftes gegeben:
   Eine **Transaktion**, für welche das *Datum* und die *Zeit* festgehalten wird, besteht aus mindestes einer **Aktualisierung**, deren *Betrag* und *Art* zu speichern ist, welche ein **Konto** betrifft. Das Konto besitzt einen *Saldo*, einen *Kreditrahmen* sowie einen *Kontotyp* und eine *Kontonummer*. Ein Konto kann jedoch von mehreren Aktualisierungen betroffen sein. Das Konto selbst wird von einer **Bank** (die einen *Namen* und eine *Bankleitzahl* besitzt) geführt (diese führt natürlich mehrere Konten). Ein **Kunde** kann mehrere Konten besitzen (*Name* und *Adresse* des Kunden sind zu speichern). Die genannte Transaktion gibt es in zwei Varianten: Eine **Kassierertransaktion** und eine **Außentransaktion**. Die Außentransaktion bezieht sich auf **Kreditkarten** (welche die *Bankleitzahl*, die *Kartennummer* und eine *Kreditkartenkontonummer* beinhalten) und wird von einer **Kreditkartenberechtigung** veranlasst. Diese besitzt ein *Passwort*, eine *Kreditkartennummer* und einen *Höchstbetrag*. Eine Kreditkartenberechtigung kann mehrere Außentransaktionen auslösen. Die Kreditkartenberechtigung identifiziert eine oder mehrere Kreditkarten. Ein Kunde kann mehrere Kreditkarten besitzen. Eine Bank vergibt natürliche mehrere Kreditkartenberechtigungen. Die Kassierertransaktion wird von einem **Kassierer**, der sich mit *Name* identifiziert und eine *Personalnummer* besitzt, ausgelöst. Ein Kassierer kann mehrere solche Transaktionen auslösen. Die Bank beschäftigt mehrere Kassierer. Die Transaktion selbst wird ausgelöst an einem **Eingabeterminal**, welches in zwei Variationen existiert: als **Geldautomat**, der einen *Terminalcode* besitzt und Bargeld ausgibt, und der einem **Konsortium** zugeordnet ist (ein Konsortium besitzt natürlich mehrere Geldautomaten), welches eine oder mehrere Banken besitzt, sowie einem **Kassenterminal** (welches ebenfalls einen *Terminalcode* besitzt) und der Bank direkt zugeordnet ist (eine Bank besitzt natürlich mehrere solche Kassenterminals).
   Erstellen Sie ein UML-Diagramm, welches die beschreibenden **Klassen**, deren Assoziationen nebst Kardinalitätsbeschränkungen und *Attribute* und ggf. Methoden wiederspiegelt.

2. Entwickeln Sie aus nachfolgendem UML-Diagramm ein ER-Diagramm, welches keine Vererbung (also nur Entitäten und Beziehungen) enthält!



3. Wie könnte man die 4 möglichen Fälle für eine UML-Vererbung {complete, overlapping}, {complete, disjoint}, {incomplete, overlapping} und {incomplete, disjoint} relational noch schärfer umsetzen (durch Einführung von Diskriminatorattributen)?

4. Im Kapitel über Softwarearchitekturen hatten wird die Transformationen vom PIM nach PSM besprochen. Es sei auf PIM-Ebene folgendes gegeben:



Diese Abbildung stelle ein (gekürztes) Autor/Buch-Modell dar, wobei gegenüber den Plattformen der eingesetzten Programmiersprache und Middleware abstrahiert wurde. Erstellen Sie ein entsprechendes UML-Diagramm auf PSM-Ebene für eine Sprache Ihrer Wahl (z.B. Java Beans o.ä.).

5. Wie könnte eine weitere Zerlegung des Realzeit-Fahrstuhlsystems des System-Kontext-Diagramms aus Abb. 5.21 aussehen, insbesondere mit Hinblick auf die Fahrstuhlkontrolle?

6. Entwurfsmuster

Ein Entwurfsmuster[12] beschreibt eine „generische" Lösung für ein mehr oder weniger häufig auftretendes Entwurfsproblem. Es ist damit im Prinzip eine wieder verwendbare Vorlage zur Problemlösung im DV-Entwurf. Entwurfsmuster eignen sich z.B. dafür, Wissen erfahrener Entwickler über die Struktur und das Zusammenspiel von Bausteinen und Klassen innerhalb von Softwaresystemen zu formalisieren, zu dokumentieren und weiterzugeben.
Der Einfluss von Entwurfsmustern ist häufig eher lokal begrenzt, d.h. nur innerhalb von einzelnen Bausteinen vorhanden. Entwurfsmuster können angewendet werden, ohne dass die Gesamtarchitektur davon betroffen ist. Sie eignen sich deshalb besonders für den Einsatz durch einzelne Entwickler einzelner Bausteine. Entwurfsmuster werden nach einem einheitlichen Schema dokumentiert. Hierdurch sind sie leichter anwendbar, und auch der Vergleich zwischen scheinbar ähnlichen Entwurfsmustern fällt leichter. Eine Möglichkeit sieht folgende Komponenten vor[13]:

Mustername und Klassifizierung
Der Mustername vermittelt den wesentlichen Gehalt des Musters. Ein guter Name ist wichtig, da er Teil des Entwurfsvokabulars ist. Muster werden hinsichtlich zweier Kriterien klassifiziert: Aufgabe und Gültigkeitsbereich. Die Aufgabe gibt wieder, was das Muster macht. Muster können entweder eine *erzeugende*, eine *strukturorientierte* oder eine *verhaltensorientierte* Aufgabe haben. Der Gültigkeitsbereich legt fest, ob sich ein Muster primär auf Klassen oder auf Objekte bezieht.

Zweck
Der Zweckabschnitt beantwortet die folgenden Fragen: Was macht das Entwurfsmuster? Welche spezifischen Fragestellungen oder Probleme im Entwurf behandelt es?

Motivation
Der Motivationsabschnitt besteht aus einem Szenario, welches ein Entwurfsproblem schildert und beschreibt, wie die Klassen- und Objektstrukturen des Musters das Problem lösen. Das Szenario hilft, die folgenden abstrakteren Beschreibungen des Musters leichter zu verstehen.

---

[12] Für eine ausführlichere Behandlung siehe z.B. Zöller-Greer, P.: *Software-Architekturen: Grundlagen und Anwendungen*, Verlag composia, 2010
[13] vgl. E. Gamma et. all: *Design Pattern,* Addison-Wesley, 1996

Anwendbarkeit
Der Anwendbarkeitsabschnitt beschreibt, in welchen Situationen das Entwurfsmuster angewendet werden kann. Es benennt die Problemsituationen, in denen das Muster helfen kann und woran diese Situationen erkannt werden können.

Struktur
Das Strukturdiagramm besteht aus einer grafischen Repräsentation der Klassen im Entwurfsmuster. Hierzu werden in erster Linie UML-Interaktionsdiagramme verwendet, um Abfolgen von Operationsaufrufen zwischen Objekten zu veranschaulichen.

Teilnehmer
Der Teilnehmerabschnitt beschreibt die am Entwurfsmuster beteiligten Klassen und Objekte sowie ihre Zuständigkeiten.

Interaktionen
Der Interaktionsabschnitt beschreibt, wie die Teilnehmer zur Erfüllung der gemeinsamen Aufgabe zusammenarbeiten.

Konsequenzen
Der Konsequenzabschnitt diskutiert, wie das Muster seine Ziele zu erreichen versucht, welche Vor- und Nachteile sich durch seine Anwendung ergeben, was für Ergebnisse zu erwarten sind und welche Aspekte der Systemstruktur voneinander unabhängig variiert werden können.

Implementierung
Der Implementierungsabschnitt präsentiert Fallen, Tipps und Techniken, deren Kenntnis für die Implementierung des Musters hilfreich sind. Darüber hinaus werden ggf. sprachspezifische Aspekte und Implementierungsmöglichkeiten benannt.

Beispielcode
Der Beispielcode diskutiert Codefragmente, die veranschaulichen, wie das Entwurfsmuster beispielsweise in C++ implementiert werden kann.

Bekannte Verwendungen
Dieser Abschnitt benennt Beispiele für das Muster, die in realen Systemen zu finden sind. Dabei werden mindestens zwei Beispiele aus unterschiedlichen Anwendungsbereichen aufgeführt.

Verwandte Muster
Der letzte Abschnitt der Musterbeschreibung setzt das Muster in Bezug zu anderen Entwurfsmustern, diskutiert die relevanten Unterschiede und erläutert, mit welchen Mustern das jeweilige Entwurfsmuster zusammen verwendet werden kann.

Generell sollte die Dokumentation des Entwurfsmusters ausreichende Informationen über das Problem, welches das Muster behandelt, über den Kontext der Anwendung und über die vorgeschlagene Lösung enthalten. Entwurfsmuster lassen sich nach verschiednen Aufgaben und Gültigkeitsbereichen klassifizieren. Eine Möglichkeit wäre[14]:

Entwurfsmuster		***Aufgabe***		
		Erzeugungsmuster	***Strukturmuster***	***Verhaltensmuster***
Gültigkeits-bereich	***Klassen-basiert***	Fabrikmethode	Adapter	Interpreter Schablonenmethode
	Objekt-basiert	Abstrakte Fabrik Erbauer Prototyp Singleton	Adapter Brücke Dekorierer Fassade Fliegengewicht Kompositum Proxy	Befehl Beobachter Besucher Iterator Momento Strategie Vermittler Zustand Zuständigkeitskette

Eine wichtige Aufgabe des Analytikers ist es jetzt, herauszufinden, welches der obigen Entwurfsmuster in einem konkreten Fall benutzt werden soll. Dabei ist es nicht so, dass man einfach nur eine „Checkliste“ durcharbeiten braucht und am Schluss kommt der gewünschte Mustertyp heraus. Es ist vielmehr so, dass man oft das gleiche Problem mit verschiedenen Mustertypen lösen könnte, nur eignen sich manchmal eben bestimmte Mustertypen für bestimmte Problemfelder besser als andere. Hinzu kommt auch, dass in der Industrie häufig aus „traditionellen Gründen“ mit bestimmten Mustertypen gearbeitet wird, relativ unabhängig vom Problemtyp, da mit diesen Mustern schon viel Erfahrung vorhanden ist.
Um den Einsatzbereich der jeweiligen Muster etwas näher zu spezifizieren, seien nachfolgend die gebräuchlichsten Typen ganz kurz charakterisiert. Dabei wird nach den Aufgabentypen unterschieden.

[14] ibid.

Erzeugungsmuster

Klassenbasiert:

Fabrikmethode (Factory Method, Virtual Constructor)

Methode zum Erzeugen von Objekten, von denen nur die abstrakte Basisklasse bekannt ist. Eine (bis dahin abstrakte) Anwendung benötigt ein (bis dahin abstraktes) Objekt, also etwas, das sie nicht direkt erzeugen kann (da abstrakt), obwohl sie muss. Sie benutzt dafür dann eine Methode, die ein solches Objekt erzeugt. Die spätere nicht-abstrakte Anwendung überschreibt die Methode und erzeugt das konkrete Objekt.

Objektbasiert:

Abstrakte Fabrik (Abstract Factory, ToolKit)

Objekte und deren Erzeugung sind vor dem Klient verbergen. Fabrik und Objekte werden dadurch austauschbar. Der Klient soll gewisse Objekte benutzen, aber ihre Klassen nicht kennen. Um sie zu erzeugen, benutzt er eine Fabrik, welche eine Schnittstelle bietet, um die (dahinter verborgenen) Objekte zu erzeugen. Durch Austausch der Fabrik können andere Objekte erzeugt werden.

Erbauer (Builder)

Man trennt hier die Berechnungen (=Konstruktion, Director) von der Präsentation (=Builder), wodurch bei gleicher Berechnung leicht unterschiedliche Präsentationen gewählt werden können.

Prototyp (Prototype)

Ein Erzeuger kreiert Objekte (z.B. mit der Clone-Funktion) eines übergebenen Objekts (Prototypen). Der Erzeuger soll also Objekte erzeugen, von denen er nur eine Basisklasse kennt (d.h. die konkrete Klasse ist unbekannt). Eine Clone-Funktion in dieser Basisklasse (Prototyp) erzeugt damit eine Kopie des Objekts. Jeder Erzeuger erhält bei Initialisierung ein fertiges Objekt von der Klasse, von der er Objekte erzeugen soll (die Benutzung der Clone-Funktion ermöglicht dies auch ohne die Klasse zu kennen).

Singleton

Her wird garantiert, dass nur *ein* Objekt von diesem Typ erzeugt wird. Dazu muss der Konstruktor privat sein, ebenso muss es eine private Klassenvariable des eigenen Typs geben und eine öffentliche Klassenmethode für den Zugriff (und gegebenenfalls für die Erzeugung).

Strukturmuster

Klassenbasiert:

Adapter (Wrapper)

Hier werden Schnittstellen vorhandener Objekte für Klienten angepasst. Es existiert z.B. bereits eine Klasse für eine spezielle Aufgabe, aber ihre Schnittstelle passt nicht zum Klienten. Ein Adapter bietet dem Klienten die passende Schnittstelle, benutzt intern jedoch für die Funktionalität ein Objekt der vorhandenen Klasse. Alternativ könnte der Adapter auch über Mehrfachvererbung von beiden Schnittstellen abgeleitet sein.

Objektbasiert:

Adapter (Wrapper)

Wie bei „Klassenbasiert"

Brücke (Bridge, Handle, Body)

Dieses Muster trennt Abstraktion und Implementierung durch „Abstraktionshierarchie" und „Implementierungshierarchie". Wenn eine Hierarchie aufgrund unterschiedlicher Implementierung auf jeder Ebene aufgespalten werden muss, macht es Sinn, Abstraktion und Implementierung zu trennen, um so eine Abstraktionshierarchie und eine Implementierungshierarchie zu erzeugen.

Dekorierer (Decorator, Wrapper)

Dies ist eine Ummantelung von Objekten, wobei Funktionalität hinzugefügt wird. Klassen sollen mehr Funktionen erhalten, jedoch will man das Ableiten vermeiden, weil man wegen einer zusätzlichen Funktion alle Klassen ableiten müsste. Um ein Objekt mit neuen Funktionen auszustatten, wird ein Dekorierer eingesetzt, welcher das Objekt aggregiert. Der Dekorierer ist vom gleichen Basistyp, bietet also dieselbe Schnittstelle. Allerdings reicht er alles an das Objekt weiter, kann aber dabei zusätzliche Funktionen ausführen. Durch Schachtelung von Dekorierern können einem Objekt unterschiedliche Funktionalitäten hinzugefügt werden.

Fassade (Facade)

Funktionalität und Vielfalt eines Subsystems wird auf eine Klasse konzentriert und beschränkt. Ein Subsystem bietet eine Vielfalt an Funktionalität und Klassenhierarchie. Für viele Anwender ist dies zu komplex, da mitunter nur wenig Funktionalität benötigt wird. Eine Fassade kapselt das Subsystem und fasst die wichtigste Funktionalität in einer Klasse zusammen (indem sie intern die benötigten Subklassen aggregiert und unnötige Funktionalität ausblendet).

Fliegengewicht (Flyweight)

Kontextabhängige Information wird aus einem Objekt entfernt und extern verwaltet, um so die Anzahl benötigter Klassen zu minimieren. Dies wird benutzt, um zu viele Objekten einer Klasse zu reduzieren. Wenn es möglich ist, aus der

Klasse kontextabhängige Informationen (extrinsisch) herauszunehmen und von einem Kontextobjekt verwalten zu lassen, so enthält die Klasse nur noch kontextunabhängige (intrinsische) Informationen. Dann können Objekte, die sich bisher nur im Kontext unterschieden haben, durch ein einzelnes Objekt repräsentiert werden. Statt mit vielen verschiedenen Objekten wird nun mit Referenzen auf gemeinsam genutzte Objekte gearbeitet. Einer Methode wird beim Aufruf die kontextabhängige Information mitgegeben.

Kompositum (Composite)

Gruppierung von Objekten, wobei die Gruppe selbst wieder Objekt gleicher Art ist. Einzelne Objekte sollen also so gruppiert werden, dass die Gruppe sich verhält wie ein einzelnes Objekt und damit wieder mit anderen Objekten/Gruppen gruppiert werden kann. Für die Objekte existiert eine Basisklasse, und von dieser wird das Kompositum abgeleitet, wodurch es die gleiche Schnittstelle wie die Objekte hat und sich entsprechend verhält. Hinzu kommt aber, dass das Kompositum Objekte dieser Basisklasse aufnehmen kann. Ein Methodenaufruf wird von dem Kompositum an alle aggregierten Objekte weitergereicht (z.B. für die Gruppierung von Grafikelementen o.ä.).

Proxy (Surrogat)

Steuerung schwerfälliger bzw. zugriffsbeschränkter Objekte. Ein Klient benutzt Objekte (Subjekt), welche z.B. lange Ladezeiten haben oder deren Zugriff kontrolliert werden muss. Ein Proxy wird dem Subjekt vorgeschaltet, um eine schnellere Arbeit zu ermöglichen und erst dann einen Ladevorgang auszulösen, wenn dies wirklich notwendig ist. Der Ladevorgang wird hinausgezögert, indem kleinere Informationen über das Subjekt bereits im Proxy enthalten sind. Ist das Subjekt geladen, leitet der Proxy alle Anfragen weiter. Der Klient merkt nicht, ob er mit dem Proxy oder Subjekt direkt arbeitet, da das Proxy von der Basisklasse des Subjekts abstammt.

Verhaltensmuster

Klassenbasiert:

Interpreter

Überführung einer Grammatik in Objekte zur Konstruktion eines Interpreters. Wird eine Sprache benutzt, die einer Grammatik folgt, so kann diese durch Objekte dargestellt werden (Terminal/Nichtterminal-Objekte), welche wiederum zum Bau eines Interpreters verhelfen, der Sätze der Sprache interpretieren kann.

Schablonenmethode (Template Method)

Aus Primitivmethoden zusammengesetzte Skelett-Algorithmen, wobei die Primitiven durch Nachfolgeklassen implementiert werden. Eine Schablonenmethode stellt einen Algorithmus dar, von dem nur das Skelett existiert. Dieses wird aus primitiven (und vor allem abstrakten) Methoden zusammengesetzt. Damit ist

der grobe Fahrplan des Algorithmus festgelegt, die abstrakten Methoden werden durch die Unterklassen implementiert. Anders gesagt: Wenn die Algorithmen der Unterklassen strukturell gleich sind und sich nur in spezifischen Implementierungen unterscheiden, so kann man diese Struktur mittels abstrakter Methoden auch schon in der Basisklasse festlegen.

Objektbasiert:

Befehl (Command, Action, Transaction)

Befehle werden als Objekte gekapselt, die dann „herumgereicht" werden können. Der Aufrufer kennt mitunter den Empfänger nicht, aber er kennt die Basisklasse Befehl mit der Methode Execute. Der konkrete (abgeleitete) Befehl kennt dann den Empfänger und ruft dessen Methode (Action) auf. Parameter, Stapelung, Makrobefehle, Logbuch und Rückgängig wird dadurch möglich.

Beobachter (Observer, Publish-Subscribe-Methode)

Die Änderung eines Objektes zieht die Änderung anderer nach sich, gesteuert vom Observer. Um die Objekte nicht eng aneinander zu koppeln bietet der Observer die Möglichkeit, Änderungen eines Objektes an alle (angemeldeten) Beobachter weiterzumelden. Das geänderte Objekt braucht seine Beobachter also nicht zu kennen.

Besucher (Visitor)

Algorithmen werden aus den Klassen gelöst und in Algorithmenklassen konzentriert. In einer Hierarchie mit vielen Klassen verteilt sich ein Algorithmus (in Form von Methoden) auf jede einzelne Klasse. Bei vielen Algorithmen wäre es schöner, wenn die Klassenhierarchie zunächst unabhängig von den Algorithmen ist und ein Visitor, was einem Algorithmus-Objekt entspricht, die einzelnen Objekte traversiert und dann den Algorithmus entsprechend dem Objekt ausführt. Dadurch sind (objektspezifische) Algorithmen nicht mehr auf die Objekte verteilt, sondern beim Algorithmus-Objekt zentralisiert. Die Objekte müssen den Visitor akzeptieren und sich selbst an den Algorithmus übergeben.

Iterator (Cursor)

Trennung von Listen-Datenstruktur und Element-Traversion/Iteration mittels Iteratorobjekten. Man trennt die Listenstruktur von der Iteration derselben, indem man ein eigenes Iterator-Objekt entwirft, welches für die Traversierung zuständig ist. Dadurch können auf einer Liste verschiedene Iterationen stattfinden, Iterationen mit Filter etc. sind möglich, unabhängig von der inneren Struktur. Die Liste erzeugt einen Iterator mittels einer Fabrikmethode. Dadurch erzeugt die konkrete Liste ihren konkreten Iterator.

Memento (Token)

„Innerer-Zustand"-Speicherung für Undo-Operationen ohne Aufhebung der Kapselung. Ein Objekt (Urheber) soll seinen Zustand so speichern, dass ein eventuelles Undo möglich ist. Dies geschieht, indem der Klient ein Memento

von dem Objekt anfordert, indem das Objekt seinen Zustand kodiert. Durch Rückgabe des Mementos an das Objekt kann dieses sich in den alten Zustand zurückversetzen. Andere Objekte können das Memento nicht interpretieren, die Kapselung bleibt bewahrt.

Strategie (Strategy, Policy)

Trennung von Algorithmen und Objekten durch austauschbare „Algorithmen-Objekte“. Man entkoppelt Algorithmen vom Kontext-Objekt und erzeugt eigenständige Algorithmen-Objekte. Die Algorithmen-Objekte benutzen die gleiche Schnittstelle, wenn sie die gleiche Aufgabe bearbeiten. So kann dem Kontext-Objekt ein beliebiger Algorithmus (Auswahl durch den Klienten) zugeordnet werden zur Lösung der Aufgabe. Algorithmenverbesserung/-hinzunahme ist so ohne die Änderung des Kontext-Objektes möglich (siehe Beispiel unten).

Vermittler (Mediator)

Objektabhängigkeiten werden nicht durch die Objekte, sondern durch Vermittler festgelegt. Wenn sich ein Verhalten auf mehrere Objekte verteilt, so können Abhängigkeiten zwischen den Objekten entstehen. Dadurch erfordern Änderungen die Reaktionen anderer Objekte. Um nicht die Objekte/Klassen alle verknüpfen zu müssen, wird ein Vermittler zwischengeschalten, der die Reaktion auf Veränderungen steuert. Alle Objekte sind nur mit dem Vermittler verknüpft, der Vermittler kennt alle beteiligten Objekte und steuert ihr abhängiges Verhalten.

Zustand (State)

Objekte delegieren hier bestimmtes Verhalten an aggregierte, austauschbare Zustandsobjekte. Reagiert ein Objekt abhängig von seinem Zustand völlig anders, so macht es Sinn, eine allgemeine „ObjektZustandKlasse“ mit gleicher Schnittstelle wie das Objekt zu entwerfen. Für jeden konkreten Zustand wird davon eine Unterklasse abgeleitet, die das jeweilige Verhalten implementiert. Das Objekt selbst aggregiert solch eine Klasse (wechselnd entsprechend ihrem Zustand) und leitet Anfragen direkt an ihr Zustandsobjekt weiter.

Zuständigkeitskette (Chain of Responsibility)

Anfragen werden entlang einer Kette an ihre potentiellen Empfänger weitergereicht. Sendet ein Objekt eine Anfrage, wo der Empfänger noch nicht bekannt bzw. die Anfrage an mehrere Objekte gerichtet ist, so kann man die potentiellen Empfänger entlang einer Kette testen, bis ein Objekt auf die Anfrage reagiert. Dabei kennt jedes Objekt einen Nachfolger bzgl. der Zuständigkeit (Baumstrukturen etc. sind möglich).

<u>Beispiel für ein Entwurfsmuster:</u>

Angenommen, in einem Programm zur Mitarbeiterverwaltung gibt es drei Arten von Mitarbeitern, die in drei Klassen implementiert sind. Die Struktur der Ob-

jektvariablen ist gleich, nur der Algorithmus zur Gehaltsberechnung ist verschieden. Ein „Klient“ stellt dabei die Anfrage zur Gehaltsermittlung.
Abb. 6.1 stellt ein mögliches UML-Diagramm hierfür dar.

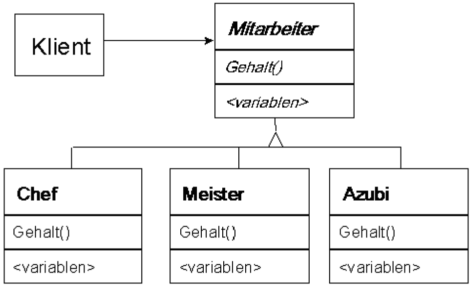


Abb. 6.1 UML-Diagramm Gehaltsabfrage

In Abb. 6.1 stellt der Pfeil zwischen Klient und Mitarbeiter eine sog. „Referenz“ (im Sinne einer Nachrichtenabfrage) dar. Die Attribute sind weggelassen, nur die Methoden „Gehalt“ sind eingezeichnet.
Eine genauere Analyse des Problems legt nahe, hierfür das objektbasierte Verhaltensmuster „Strategie“ (vgl. obige Klassifikationen) zu benutzen. Dieses Muster wird nämlich vorzugsweise immer dann benutzt, wenn sich die meisten Klassen nur in ihrem *Verhalten* unterscheiden. Der Strukturabschnitt dieses Musters zeigt uns, wie wir Algorithmen in Objekte kapseln:
Ein "Mitarbeiter"-Objekt erhält zusätzlich eine Referenz auf ein Strategieobjekt, welches hier durch eine abstrakte Oberklasse "Gehaltsermittler" vertreten wird. Eine Anfrage eines Klienten nach der Höhe des Gehaltes leitet das "Mitarbeiter"-Objekt an seinen "Gehaltsermittler" weiter. Hierbei übergibt es diesem eine Referenz auf sich selbst, damit der Algorithmus Zugriff auf die Objektvariablen des "Mitarbeiters" hat (vgl. Abb. 6.2).

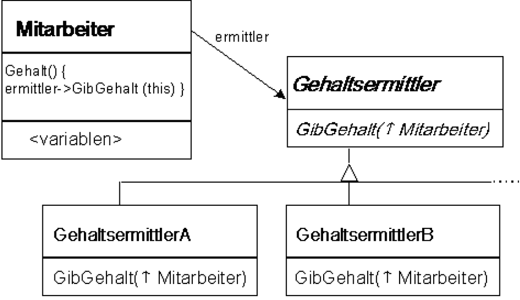


Abb. 6.2 Entwurfsmusterfreundliche Darstellung

Somit ist eine Struktur entstanden, die nur noch eine "Mitarbeiter"-Klasse kennt und in der die Menge der Algorithmen zur Berechnung des Gehaltes beliebig veränderbar ist.

Dieser kleine Ausflug in die Technik der Entwurfsmuster sollte zumindest eine Idee geben, wie man beim Anwendungsentwurf unter Umständen wiederverwendbare Komponenten entwickelt.

Übungen zum Selbsttest

1. Was sind Entwurfsmuster?

2. Auf was ist beim UML-Entwurf zu achten, wenn Entwurfsmuster wie bei dem Strategiemusterbeispiel aus Abb. 6.2 berücksichtigt werden sollen?

7. Anwendungsentwurf

Es ist ein schwieriges Unterfangen, möglichst allgemein darzustellen, wie der Entwurf von Benutzeroberflächen realisiert werden soll. In einem Kompaktkurs wie diesem kann das eigentlich gar nicht durchgeführt werden, es sei denn, man würde auf höchstem Abstraktionslevel nur ganz allgemeine Strategien besprechen. Da dieses Buch aber eher praxisorientiert sein soll, werde ich mich bewusst auf einfache Beispiele beschränken, die das wesentliche in repräsentativer Form zum Ausdruck bringen. Insbesondere werde ich nur Beispiele aus der Welt der relationalen Datenbanken benutzten (da diese in absehbarer Zeit weiterhin die wichtigste und am weitesten verbreitete Form der Datenbanken sein wird) und dafür das bekannte System MS-Access® aus der Klasse der Office-Anwendungen benutzten. Das hat den Vorteil, dass die allen Office-Anwendungen gemeinsame objektorientierte Programmiersprache Visual Basic for Applications (VBA) eingesetzt wird, welche sehr einfach zu erlernen und zudem fast „selbstsprechend" ist. So können auch Leser, die VBA bisher nicht kennen, doch die hier gezeigten Beispiele direkt verstehen, zumal die verwendeten Befehle zumindest prinzipiell erläutert werden. Es wird hier natürlich kein VBA-Kurs geboten, sondern es werden lediglich diejenigen Befehle benutzt und erklärt, welche für das Verständnis der dargestellten Anwendungsentwurfsprinzipien nötig sind. Ich bin der Meinung, dass sich diese Prinzipien leicht abstrahieren und auf beliebige andere Programmiersprachen übertragen lassen können.
Wenn man nun eine Anwendung (mit den oben genannten Einschränkungen) aus dem Entwurf heraus implementieren will, so sollten folgende Schritte in dieser Reihenfolge durchgeführt werden:

1. Anlegen aller Tabellenstrukturen
2. Anlegen aller relationalen Beziehungen
3. Evtl. hier bereits Eingabe einiger Testdaten direkt in die Tabellen, um zu prüfen, ob die Relationen die erwarteten Resultate liefern
4. Erzeugen aller benötigten Abfragen (z.B. in SQL)
5. Erzeugen von Bildschirmmasken (Screen-Objekte)
6. Erzeugung von Print-Outs oder ggf. Export-Schnittstellen (Ausgabe-Objekte)
7. Codierung der selbst zu schreibenden Funktionen
8. Einbau der Funktionen in die jeweiligen Masken oder Ausgabe-Objekte

Natürlich müssen die Punkte 5-8 nicht alle völlig komplett jeweils für sich abgeschlossen werden. Es ist ggf. sinnvoller (vgl. Spiralmodell), diese Punkte zyklisch zu durchlaufen und dabei stufenweise zu vervollständigen.

Betrachten wir als erstes Beispiel das Anlegen einer Tabelle in MS-Access®. Man kann einfach in die vorgegebene Klasse *Tabellen* gehen und dort im Menü die Methode *Neu* aufrufen, welche eine neue Tabelle anlegt. Entsprechend wer-

den die Methoden *Entwurf* (zum Bearbeiten) und *Löschen* etc. zur Verfügung gestellt. Die jeweiligen Attribute können angelegt und jeweils aus einer Menge vorgefertigter Eigenschaften die passenden herausgegriffen werden.

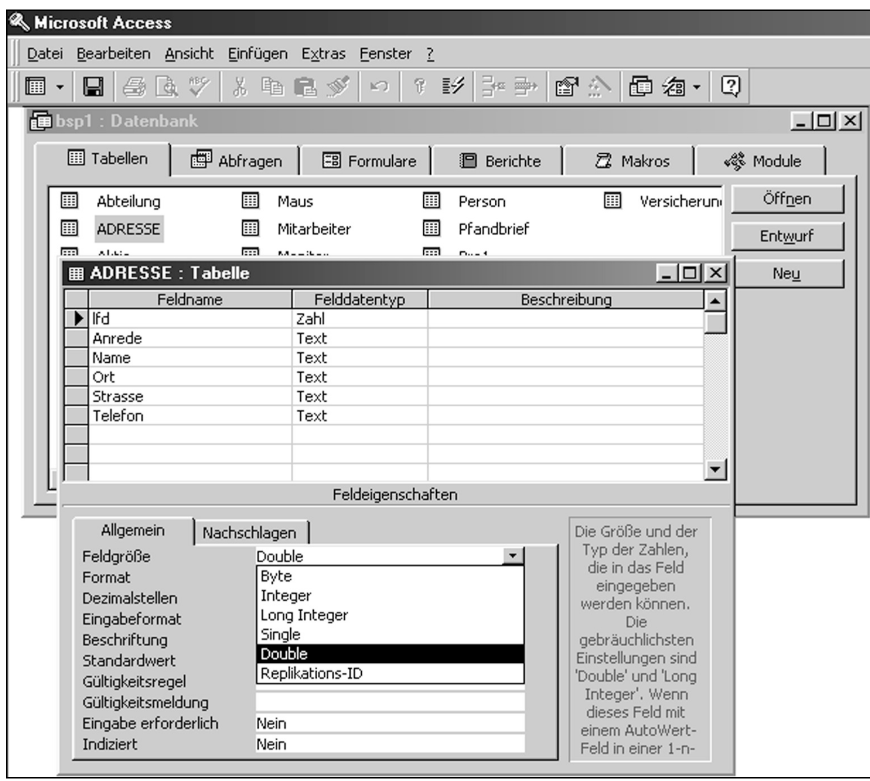


Abb. 7.1 Erzeugen und Bearbeiten von Tabellenstrukturen

An Abb. 7.1 sehen wir das Anlegen der Tabelle *Adresse* durch Betätigen der Schaltfläche *Neu* innerhalb der Klasse *Tabellen*. Die Namen der Attribute (hier *Feldname* genannt) sowie die zugehörigen Domains (hier *Felddatentyp* genannt) können eingetragen und spezifische Eigenschaften zugeordnet werden. Man sieht auch, dass unter den Eigenschaften (*Allgemein*) sich Begriffe wie *Standardwert* oder *Gültigkeitsregel* befinden. Zu Beachten ist, dass auch die Primärschlüssel entsprechend des Planungsschemas anzulegen sind.

Sind auf diese Weise alle Tabellen einer Anwendung definiert, so müssen als nächstes die relationalen Beziehungen realisiert werden. Als Beispiel diene wiederum MS-Access®. Hier lassen sich Beziehungen besonders einfach anlegen. In einem eigenen Beziehungsfenster kann man die beteiligten Tabellen importieren und dann per Maus das oder die Schlüsselattribut(e) von der *1*-Seite auf die *n*-Seite „ziehen“. Danach öffnet sich ein weiteres Fenster, wo die Angaben zur referentiellen Integrität gemacht werden.

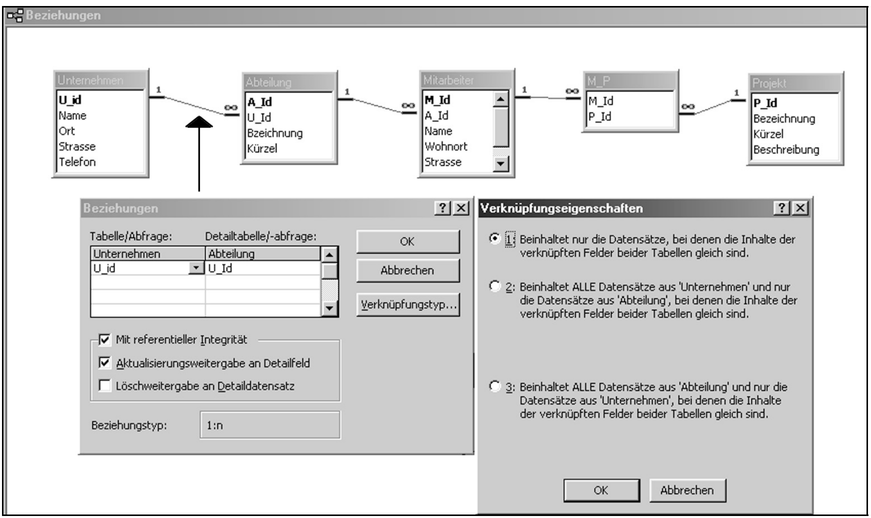


Abb. 7.2 Implementierung von Relationen

In Abb. 7.2 sind die relationalen Verknüpfungen des Beispiels aus Abb. 5.12 implementiert. Die Verknüpfungseigenschaft „mit referentieller Integrität“ sollte unbedingt ausgewählt werden, ebenso die Aktualisierungsweitergabe von Änderungen am Primärschlüssel der *1*-Seite an die *n*-Seite. Die Löschweitergabe (Kaskadenlöschen) sollte dagegen wirklich nur ausgewählt werden, wenn der Kunde dies ausdrücklich wünscht, denn beim Löschen eines Datensatzes der *1*-Seite kann sonst eine ganze Flut von Löschungen auf allen davon abhängigen *n*-Seiten einsetzen. Bei den Verknüpfungseigenschaften kann noch ausgewählt werden, ob die relationale Beziehung vollständig oder nur teilweise realisiert werden soll. Bei atomaren Primärschlüsseln spielt diese Einstellung keine Rolle, doch bei zusammengesetzten Primärschlüsseln kann hier angegeben werden, ob die Verknüpfung der Instanzen der Tabellen nur erfolgen soll, wenn alle Inhalte der Attribute des Primärschlüssels eines Datensatzes auf der *1*-Seite die gleichen

Werte haben wie die zugehörigen Fremdschlüsselattributsinhalte der *n*-Seite, oder ob bereits Teilmengen davon ausreichen. Gewöhnlich wird man hier den vorgeschlagenen Verknüpfungstyp *Gleichheit* beibehalten.
Die Beziehungen sollte man anlegen, bevor man irgend welche Daten in die Tabellen eingibt, denn im Falle inkonsistenter Eingaben kann unter Umständen die referentielle Integrität verletzt sein, und die gewünschte Beziehung lässt sich dann gar nicht erstellen. Sind schließlich alle Beziehungen angelegt, so ist es sinnvoll, einige Testdaten anzulegen, und zwar aus verschiedenen Gründen. Zunächst sieht man bereits beim Anlegen, ob die Integritätseigenschaften sinnvoll gewählt wurden. Inkonsistente Eingaben werden nämlich bereits bei der Eingabe abgelehnt. Außerdem sind angelegte Testdaten später bei der Entwicklung von Bildschirmmasken dann in denselben auch schon sichtbar, und man muss nicht nur mit leeren Screens arbeiten.
Es sollte noch über das Anlegen von sog. „Indizes" gesprochen werden. Ein Index ist eine eigene interne Tabelle, die nach gewünschten Attributen sortiert ist und einen Verweis auf den Datensatz der eigentlichen Tabelle besitzt. In Indextabellen kann sehr effektiv gesucht werden, sie benötigen aber auch zusätzlichen Speicherplatz. Die Indextabellen selbst sind in der Regel versteckt und das Datenbankmanagementsystem kümmert sich um deren Verwaltung. Als Faustregel gilt: Für Attribute, die häufig Gegenstand eines Suchvorgangs oder einer Sortierung sind, sollte ein Index angelegt werden. Seltene Suchvorgänge können auch „seriell" durchgeführt werden, das dauert zwar dann etwas länger, benötigt aber keinen zusätzlichen Speicherplatz.
Obwohl wir relationale Datenbanksysteme zugrunde legen, werden für die Programmiersprachen der Implementation dennoch objektorientierte Methoden benutzt. Ein wesentliches Prinzip so einer objektorientierter Programmiersprache ist es nun, dass es spezielle Datentypen für *Objekte* gibt, die dann folgerichtig *Objektdatentypen* genannt werden. Diese können dann zu Klassen zusammengefasst werden. Wir werden im Laufe dieses Kapitels einige solcher Objektdatentypen näher kennen lernen. Betrachten wir eine typische Vereinbarung aus einem VBA-Programm in MS-Access®:

```
Dim dbank As Database
Dim dtab As Recordset
Set dbank = CurrentDb()
Set dtab = dbank.OpenRecordset("a90umw")
```

Der `DIM`-Befehl wird in VBA zur Deklarierung des Datentyps benutzt. Hinter dem Wörtchen *As* steht dann der jeweilige Datentyp. Zulässige Typen sind: *Byte, Boolean, Integer, Long, Currency, Single, Double, Decimal*, *Date, String*

(für Zeichenfolgen variabler Länge), *String * Länge* (für Zeichenfolgen fester Länge), *Object, Variant*, ein benutzerdefinierter Typ oder ein *Objekttyp*.
Database ist z.B. ein Objekttyp, der eine ganze Datenbank kennzeichnet. Der Objektvariablen *dbank* kann damit später ein Inhalt zugeordnet werden, der auf eine Datenbank verweist. Durch den Befehl

```
Set dbank = CurrentDb()
```

wird die aktuelle ACCESS-Datenbank zugewiesen. Es wäre auch denkbar, hier eine externe Datenbank unter Angabe des Laufwerks und Pfads und Dateinamens zuzuweisen. Der Datentyp Recordset ermöglicht es, der Objektvariablen *dtab* eine Tabelle mit Namen *a90umw* zuzuweisen. Dabei wird auch ein allgemeines Prinzip deutlich: Die Befehlszeile

```
Set dtab = dbank.OpenRecordset("a90umw")
```

liefert uns nicht nur die „Wertzuweisung“ der Tabelle mit Namen *a90umw* zu der Objektvariable *dtab*, sondern diese Tabelle wird gleichzeitig geöffnet. Allgemein gilt für den Zugriff auf *Objekte* einer Klasse mit einer *Methode* der Syntax:

```
Objekt.Methode
```

Die Methode *OpenRecordset*, welche auf die Objektvariable *dbank*, die den aktuellen Wert *CurrentDb()* hat, zugreift, öffnet damit die Tabelle *a90umw* und weist die geöffnete Tabelle der Objektvariablen *dtab* zu. Liefert der Zugriff einer Methode auf ein Objekt wieder ein Objekt, so kann darauf mit weiteren Methoden zugegriffen werden:

```
Objekt.Methode1.Methode2.Methode3 ...
```

Hier liefert *Objekt.Methode1* ein weiteres Objekt, auf das mit *Methode2* zugegriffen wird und wieder ein Objekt liefert, das mit *Methode3* angesprochen wird und so weiter.
Sind Tabellen und Beziehungen implementiert und auch schon Testdaten eingegeben, dann folgt die Implementierung der Bildschirmmasken. Man unterscheidet dabei zwischen Navigationsmasken und Eingabemasken. Reine Bildschirmausgaben fallen nicht unter den Begriff eines Dialogobjekts, da hier kein „Dialog“, sondern nur eine Anzeige stattfindet. Solche „Masken“ zählen zu den Ausgabeobjekten, die im Prinzip wie Formulare behandelt werden können.

Man kann verschiedener Meinung sein darüber, ob die Entwicklung von Bildschirmmasken schon im DV-Entwurf vorgenommen werden sollte oder erst während der Implementierung. In der Tat ist es so, dass früher, bevor sich Prototyping etablierte, dies auf jeden Fall zuerst im Entwurf gemacht wurde. Da aber damals die Programmierung solcher Masken, wie bereits erwähnt, viel zu aufwändig gewesen wäre, hat man die Maskenentwürfe einfach gezeichnet. Das heißt, man hat die Bildschirme mit ihren Elementen teilweise sogar schon im Pflichtenheft zeichnerisch dargestellt, spätestens jedoch im DV-Entwurf. Da man heute aber über geeignete Tools verfügt und außerdem von der strikten linearen Abfolge der Phasen im Wasserfallmodell abgekommen ist und eher die Spiralmodellphasen anwendet, ist der Maskenentwurf in die Implementierungsphase „gerutscht". Die Grenzen der Entwicklungsphasen sind ohnehin nicht starr und sollten sogar projektspezifisch angepasst werden. Je nach Projektanforderungen kann es daher durchaus Sinn machen, die Maskenentwürfe bereits in der DV-Entwurfsphase zu planen, zumindest deren Hierarchien. Im Prototypingfall liegt aber die Maskenhierarchie oft noch gar nicht genau fest, so dass eine zu starre Planung eher hinderlich als nützlich sein kann. Wir gehen jedenfalls davon aus, dass durch das Datenmodell implizit auch zumindest die abstrakten Inhalte der Masken festliegen, denn die Attribute in den Objektklassen müssen mit Daten gefüllt werden, und die Klassen selbst sind bereits ein guter Ansatzpunkt für die zu entwerfenden Masken und Menüs. Mittlerweile hat sich auch ein Verfahren für die Entwicklung von Bildschirmmasken etabliert, das in der Programmierung schon längst bekannt ist: Das Wiederverwenden bereits vorhandener Module. So etwas wird auch im Bereich der Maskenentwicklung mit dem Begriff *Entwurfsmuster* umschrieben (vgl. Kapitel 6). Man benutzt vorgefertigte Bildschirmschemata, die immer wieder verwendet werden können und in Klassen zu sog. Screenobjekten zusammengefasst sind. Beispiele für so etwas sind z.B. die Bildschirmmasken in Microsoft®-Anwendungen wie z.B. MS-Word® oder MS-Excel®. Hier finden sich immer wieder die gleichen Bildschirmaufbauten. Dies bietet zweierlei Vorteile: Für den Entwickler, dass er nämlich auf bereits bestehende Muster zurückgreifen kann und diese nur anzupassen braucht, und für den Anwender, dass er sich, wenn er beispielsweise MS-Word® schon kennt, beim Einarbeiten z.B. in MS-Excel® an seine Kenntnisse über die Funktionen und Menüpunkte anlehnen kann. Es ist gerade zu ein Markenzeichen für große Softwarehersteller geworden, dass sie bei all ihren Produkten ähnliche Bildschirmmasken mit vergleichbaren Funktionen an immer wieder den selben Stellen erzeugen. Kennt man eines der Produkte bereits, dann liegt es nahe, bei dem selben Herstellen ein anderes benötigtes Produkt mit ähnlichem Aufbau wieder zu kaufen, auch wenn andere Hersteller ein vergleichbares Produkt anbieten. Die Psychologie und die schnelleren Einarbeitungszeiten spielen hier eine wichtige Rolle. Dabei spielt auch noch der Begriff *Softwareergonomie*

eine große Rolle. Programme sollen für einen Anwender möglichst freundlich wirken, leicht zu bedienen und übersichtlich sein. Es sollte keine Maske existieren, auf der nicht zu erkennen ist, wie man z.B. wieder aus ihr heraus kommt. Es sollen immer Hilfe-Fenster aufrufbar sein, die dem Benutzer weiterhelfen können. Die Zeit dicker Benutzerhandbücher, wo ein Anwender erst stundenlang nachlesen musste, wie er die Software bedienen muss, ist schon lange vorbei. Deswegen beinhalten heutzutage Benutzerhandbücher, wenn sie denn überhaupt noch existieren, allenfalls Erklärungen zu bestimmten Funktionen oder einen grundsätzlichen Überblick über die Philosophie der Maskengestaltungen und ihrer Elemente und eine kurze Erklärung der Schaltflächen oder ähnliches.
Es ist in jedem Fall aber sinnvoll, dass die jeweiligen Bildschirmmasken zumindest grob in ihrer Hierarchie geplant werden. Für die praktische Ausprägung so eines Maskenhierarchiediagramms gibt es viele Möglichkeiten. So kann z.B. eine verbale Formulierung, die durch textuelle Einrückungen die Hierarchien wiederspiegelt, erfolgen:

```
Hauptmenü
      Untermenü 1
            Untermenü 11
                  Erfassen Daten 111
                  Anzeigen Daten 112
            Untermenü 12
      Untermenü 2
            Drucken Daten 21
            Suchen Daten 22
...
```

Eine andere Alternative ist die grafische Darstellung. Hierbei ist es üblich, in Form eines Organigramms die jeweiligen Masken und Menüs darzustellen. Abb. 7.3 zeigt dies am Beispiel des schon erwähnten Lizenzabrechnungsprogramms. Je nach Komplexität kann so ein Diagramm auch in mehrere Teildiagramme zerlegt werden, wobei ein Hauptdiagramm die wichtigsten Verzweigungen zeigt, und dort gewisse Black-Boxen auf weitere Unterdiagramme verweisen, welche dann an anderer Stelle genauer dargestellt sind.

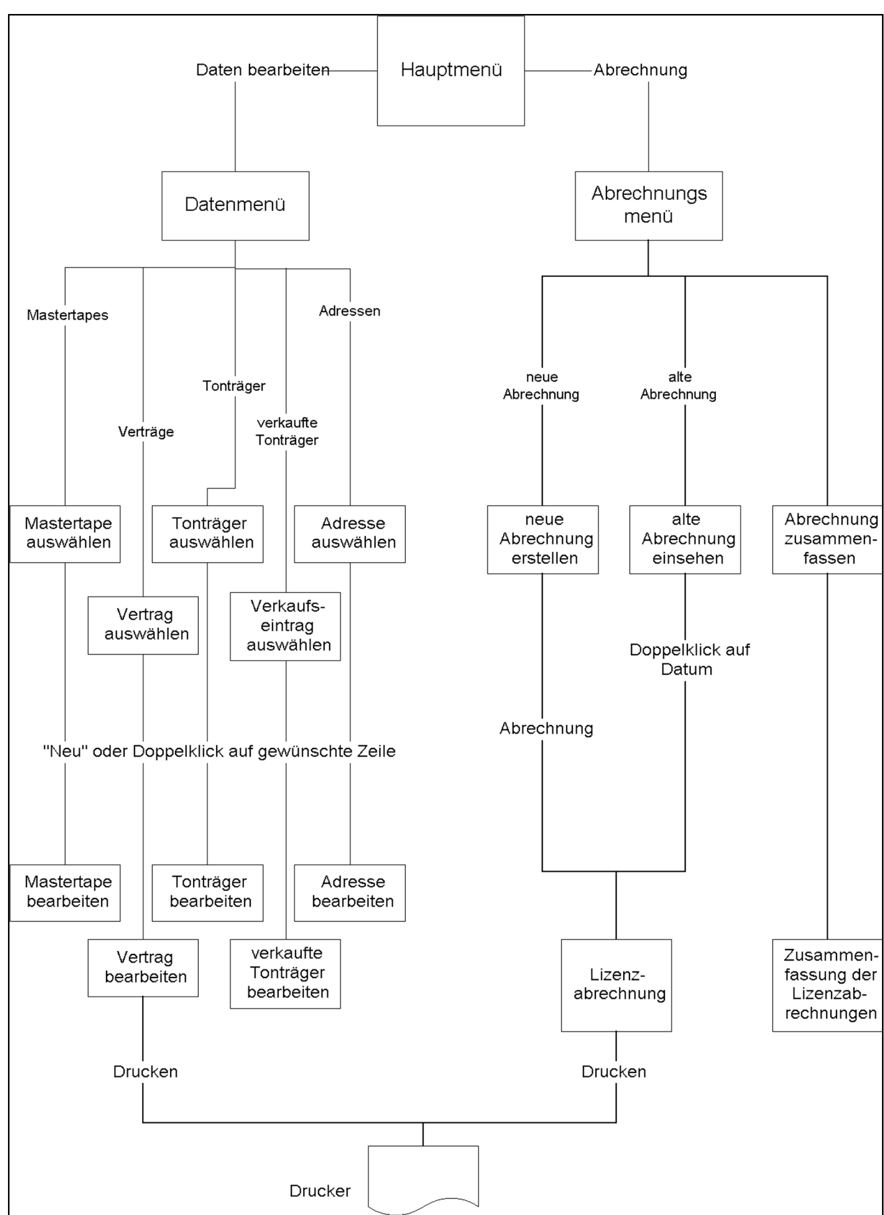


Abb. 7.3 Maskenhierarchiediagramm

Navigationsmasken
Nachfolgend seien einige typische Elemente einer Navigationsmaske gezeigt.

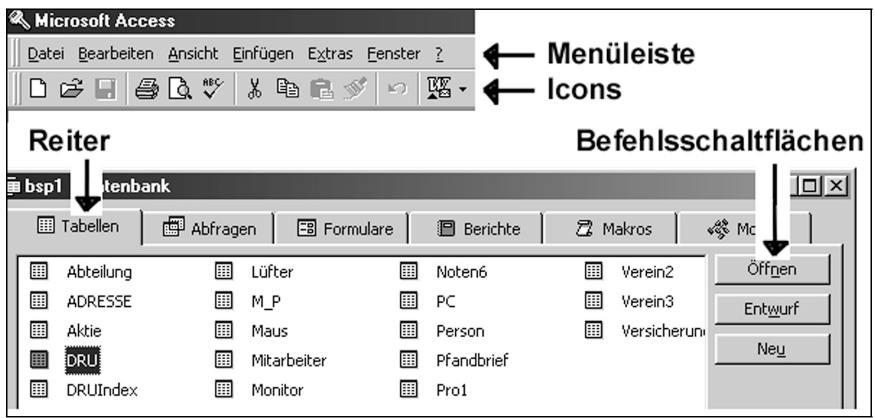


Abb. 7.4 Screen-Elemente zur Navigation

Welche Elemente letztendlich bevorzugt eingesetzt werden, ist Geschmackssache. Es sollte ggf. mit dem Auftraggeber auch darüber Rücksprache gehalten werden. Wir werden uns daher auch nicht weiter mit solchen eher kosmetischen Masken-Design-Fragen beschäftigen, sondern uns auf die Inhalte konzentrieren.
Betrachten wir zunächst einmal eine Navigationsmaske mit Schaltflächen. Hinter einer Schaltfläche können sich prinzipiell vier verschiedene Dinge verbergen:

1. Ein Untermenu mit weiteren Schaltflächen
2. Ein reine Bildschirmanzeige ohne Editiermöglichkeiten (aber ggf. mit Auswahlmöglichkeiten. z.B. Liste von Datensätzen)
3. Eine Maske zur Datenerfassung
4. Kombinationen aus 1., 2. und 3.

Zum Beispiel das Abspielen eines Videoclips oder die Anzeige einer Autoreninformation des Anwendungsprogramms fällt unter Punkt 2. In der Entwurfsansicht einer Bildschirmmaske lassen sich die Aktionen, die z.B. durch das Betätigen einer Schaltfläche ausgelöst werden sollen, einstellen.
Ich möchte in diesem Kapitel nun die „abstrakteren" Ebenen des Entwurfs verlassen und „hinabsteigen" in die bodenständige Anwendungsentwicklung. Es wird also davon ausgegangen, dass ein Pflichtenheft und ein vollständiger DV-Entwurf (incl. UML-Diagramme, Funktionsbeschreibungen, I/O- Beschreibungen etc.) vorhanden ist. Wir haben auch schon über die Umsetzung der UML-Klassendiagramme in relationale Entitäten und Beziehungen gesprochen. Klas-

sen werden zu Entitäten, Assoziationen und Aggregationen zu Beziehungen zwischen Entitäten und Vererbungen werden zu (1,1):(0,1)-Beziehungen zwischen der Basisklasse und den Subklassen. Damit ist es also möglich, alle hier vorgestellten UML-Beispiele in relationale Tabellen nebst deren Beziehungen umzusetzen. Jetzt geht es nur noch darum, um die „Tabellen herum“ Bildschirmmasken zu basteln, so dass schließlich eine benutzerfreundliche Anwendungsoberfläche für die Nutzer des Programmsystems entsteht.
In diesem Kompaktkurs soll das ohne viel Theorie an einem Beispiel praktisch demonstriert werden. Es geht am Schluss einfach immer nur darum, wie bestimmte Assoziationstypen in relationalen Datenbanken abgebildet und welche Arten von Bildschirmmasken diesen verschiedenen Typen zugewiesen werden. Das Prinzip ist immer gleich, nur die Inhalte ändern sich von Anwendung zu Anwendung. Weiß man einst, wie man eine *1:n*-Beziehung zwischen Tabellen auf einer Bildschirmmaske darstellt, so spielt es keine Rolle, ob es sich dabei um eine Beziehung zwischen einer Abteilung und ihren Mitarbeitern oder um eine CD und deren darauf enthaltenen Songs handelt. Das Bildschirm-Layout ist das gleiche, nur die Beschriftungen sehen anders aus.
Wir geben also folgendes einfache Beispiel vor und demonstrieren daran den Anwendungsentwurf.
Es sei eine Behörde gegeben, die mehrere Abteilungen besitzt. In der Behörde gibt es Mitarbeiter, welche hierarchisch strukturiert sind (d.h. es gibt Vorgesetzte und Untergebene). Die Mitarbeiter der Behörde betreuen „Kunden“, welche nicht zu der Behörde gehören (z.B. die Abteilung „Finanzamt“ hat bestimmte Mitarbeiter, die steuerpflichtige Bürger „betreuen“). Dabei ist es so, dass ein Kunde von mehreren Mitarbeitern betreut werden kann und ein Mitarbeiter kann mehrere Kunden betreuen. Des weiteren soll die Anwendung so gestaltet sein, dass ein Zugang über die Abteilungen besteht, wo auch neue Abteilungen angelegt werden können und die Mitarbeiter der jeweiligen Abteilung angezeigt werden. Aber Mitarbeiter können hier nicht angelegt werden, dafür gibt es eine eigene Erfassungsmaske. Beim Anlegen eines Mitarbeiters wird dieser jeweils einer Abteilung (die dann schon angelegt sein muss) zugewiesen. In dieser Erfassungsmaske sollen auch gleich Untergebene und Vorgesetzte (die schon angelegt sein müssen) zuweisbar sein. Natürlich soll hier nicht nur neu angelegt, sondern auch nachträglich wieder etwas verändert werden können. Dann soll es im Programm einen Zugang zu den Kunden geben, d.h. es sollen Kunden angezeigt und angelegt werden können. Beim Anlegen eines Kunden werden auch gleich die zuständigen Mitarbeiter der Behörde (die schon angelegt sein müssen) zugeordnet. Auch hier soll nachträglich wieder etwas verändert werden können.
Ein mögliches Klassenschema für diese Anforderungen könnte folgendermaßen aussehen (Abb. 7.5):

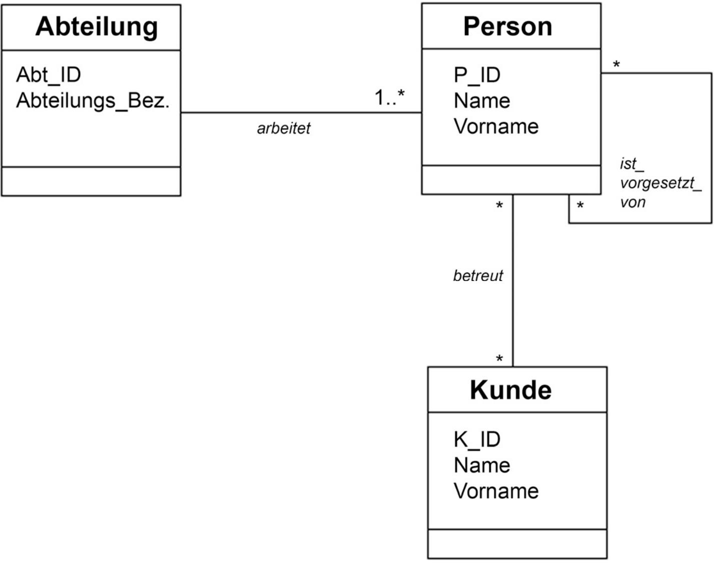


Abb. 7.5 UML-Klassendiagramm

Die Mitarbeiter sind in der Klasse *Person* untergebracht und es sind keine Assoziationsklassen eingezeichnet, da diese lediglich Fremdschlüssel beinhalten würden und keine eigenen Linkattribute. Da hier keine Vererbungen vorliegen, kann dieses Klassendiagramm praktisch direkt in ein relationales Tabellendesign umgewandelt werde. Gemäß Kapitel 6 heißt dies, dass wir aus jeder der abgebildeten Klassen eine Tabelle machen. Die Assoziation zwischen Abteilung und Person entspricht einer *1:n*-Beziehung und wird daher so umgesetzt, dass in der Tabelle *Person* der Primärschlüssel der Tabelle *Abteilung* als Fremdschlüsselattribut hinzugefügt wird. Die reflexive Beziehung Vorgesetzte/Untergeben der Tabelle Person sowie die Beziehung zwischen Person und Kunde sind jeweils *n:m*-Beziehungen und es werden deswegen dafür jeweils eine eigene Beziehungstabelle, welche nur die Fremdschlüssel der beteiligten Tabellen enthält, angelegt.

In MS-Access wird nun so vorgegangen, dass zunächst alle Tabellen angelegt werden, auch die beiden Beziehungstabellen (wie in Abb. 7.1 gezeigt). Danach werden die Beziehungen im MS-Access-Beziehungsfenster angelegt (vgl. Abb.7.2), in dem man durch Drag-und-Drop die Beziehungen einzeichnet. Letzteres führt dann zu in Abb. 7.6 dargestelltem Ergebnis. Die Primärschlüssel der jeweiligen Tabellen erscheinen fettgedruckt.

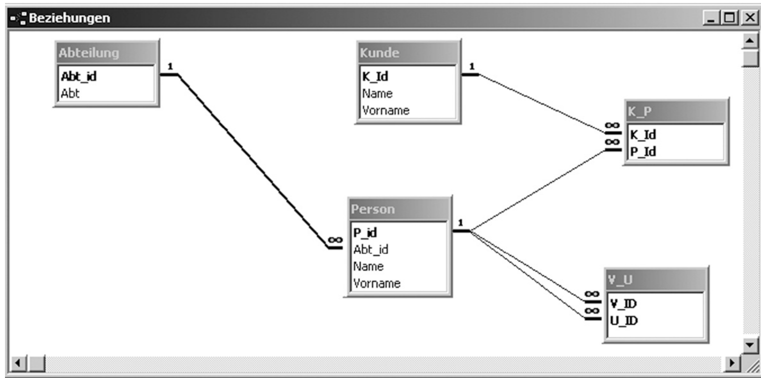


Abb. 7.6 Beziehungen in MS-Access

Nachdem die Beziehungen angelegt sind, empfiehlt es sich, bereits ein paar Testdaten in die Tabellen einzutippen, um festzustellen, ob alles so funktioniert wie es soll. Außerdem hat man dann für die nachfolgend zu erstellenden Bildschirmmasken schon ein paar Daten zur Ansicht.
Der nächste Schritt besteht im Planen der Bildschirmmasken. Zunächst bedarf es eines „Startmenüs". Dazu wird in MS-Access ein sog. „Formular" erstellt. Dabei handelt es sich um ein Element der Access-Klasse „Formulare", das sind also Bildschirmmasken, die sowohl zum Erfassen als auch zum reinen Anzeigen oder Navigieren benutzt werden können. Die Objekte dieser Klasse sind die Formulare, und diese enthalten i.d.R. weitere Objekte wie Schaltflächen, Textanzeigen, Kopf- und/oder Fußzeilen, Eingabefelder etc.; also hier haben wir die klassentypische Möglichkeit, dass Objekte selbst wieder Klassen darstellen können.
In Abb. 7.7 sehen wir so eine mögliche Einstiegsmaske gemäß den eingangs beschriebenen Kundenwünschen unseres Beispiels. In Abb. 7.8 sehen wir die Entwickleransicht davon. Grundsätzlich ist es dabei so, dass man mit Hilfe eines Macros MS-Access dazu veranlassen kann, beim Start dieses Macro ausführen zu lassen. In MS-Access werden zunächst alle Elemente des Programm in einer Datei mit der Endung .mdb gespeichert (Masken, Tabellen, Macros, VBA-Module etc.). Der fest definierte Name *Autoexec* eines Marcos führt dazu, dass dieses Macro beim Aufruf der .mdb-Datei direkt ausgeführt wird. Beim Anlegen des Macros kann man dann also z.B. den Eintrag „Öffne Formular(*name_des_Formulars*)" auswählen, so dass beim Start der Datei dann dieses Formular als erstes erscheint. In Abb. 7.8 kann man auch sehen, dass die Schaltflächen eine Eigenschaft „Beim Klicken" besitzen. Man kann dann entweder ein Macro oder ein VBA-Modul damit als Ereignis verknüpfen.

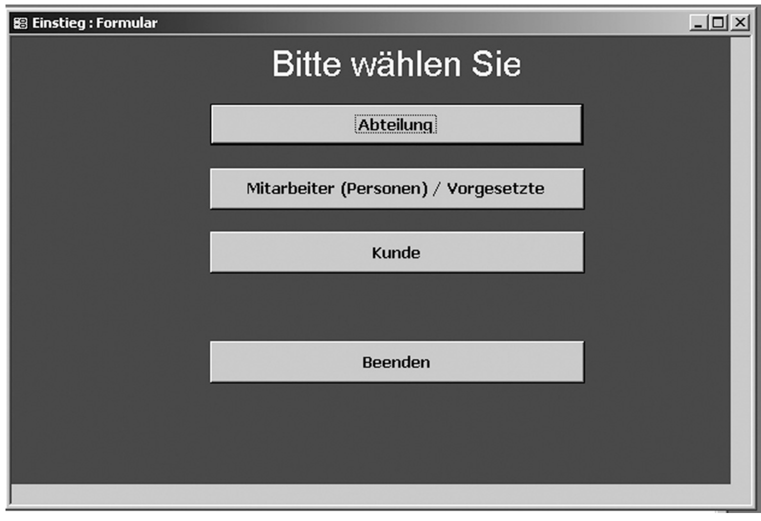


Abb. 7.7 Einstiegsmaske (User-Ansicht)

In nachfolgender Abbildung ist die Schaltfläche „Abteilung" markiert und die Eigenschaftsanzeige bezieht sich darauf.

Abb. 7.8 Einstiegsmaske (Entwickler-Ansicht)

Wenn man nun rechts neben dem Eintrag „Beim Klicken (Ereignisprozedur)“ auf die kleine Schaltfläche mit den drei Punkten klickt, öffnet sich der VBA-Editor und man kann einen VBA-Code eintippen, welcher dann beim Klicken auf die Schaltfläche „Abteilung“ abläuft (vgl. Abb. 7.9).

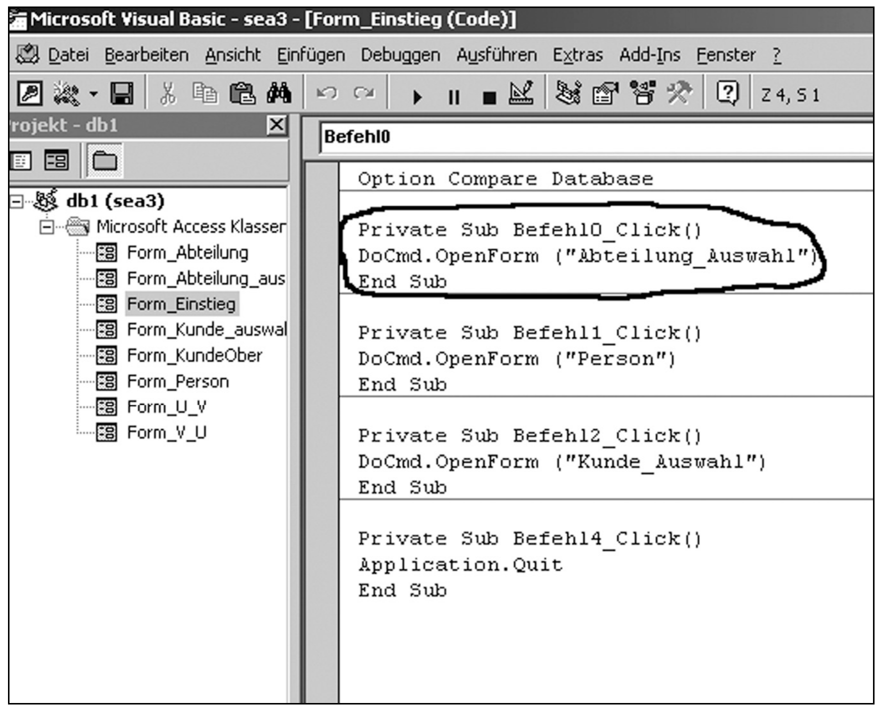


Abb. 7.9 VBA-Editor

In unserem Fall wird die Methode `OpenForm` mit dem Parameter „Abteilung_Auswahl“ (=Name eines anderen Formulars) auf das Objekt `DoCmd` ausgeführt; das Objekt `DoCmd` ist eine Art Dummy-Objekt, welches immer benutzt wird, wenn einfach eine Aktion ausgeführt werden soll, die sich (vorher) auf kein konkretes Objekt bezieht (das Formular-Objekt wird hier ja erst durch das Ausführen der Aktion „Öffnen“ angezogen).
In Abb. 7.10 sehen wird die User-Ansicht des durch Anklicken der Schaltfläche „Abteilung“ gemäß des VBA-Programms geöffneten Formulars mit dem Namen „Abteilung_Auswahl“.

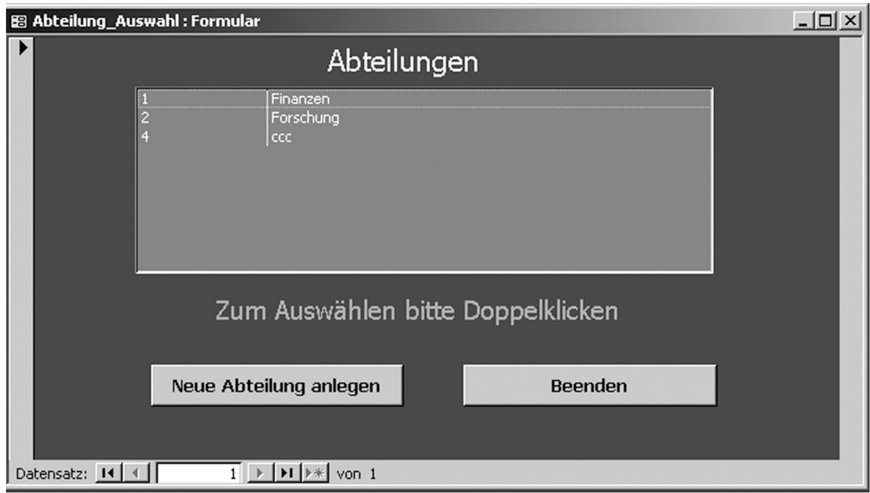


Abb. 7.10 Formular „Abteilung_Auswahl“ (User-Ansicht)

Den eigentlichen Abteilungen ist hier also eine Art Inhaltsverzeichnis vorgeschaltet, so dass der User per Doppelklick aus der (sortierten) Liste eine Abteilung auswählen kann (bei mehr Einträgen als die Fenstergröße hergibt entstehen am Rande automatisch Rollbalken zum Scrollen).

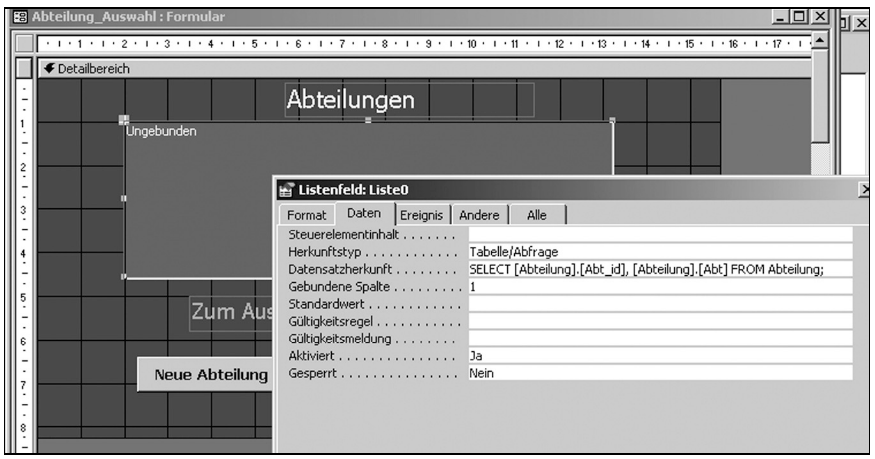


Abb. 7.11 Formular „Abteilung_Auswahl“ (Entwickler-Ansicht „Datensatzherkunft“)

In Abb. 7.11 sieht man, dass das Listenfeld für das „Inhaltsverzeichnis" (der User-Ansicht) in der Entwickleransicht als ungebundenes Feld erscheint. Man kann in MS-Access Objekte an andere Objekte nämlich binden, z.B. ein ganzes Formular oder eine Liste in einem Formular an eine Tabelle. Man kann es aber auch ungebunden lassen und z.B. wie in Abb. 7.11 ersichtlich, die Liste über den Eigenschaftsreiter „Daten" mit Werten aus einer SQL-Abfrage füllen (unter „Datensatzherkunft"). So geschehen hier, und man sieht auch, dass die 2 Attribute *Abt_ID* und *Abt* angezeigt werden sollen.
In Abb. 7.12 kann man dann sehen, dass durch die Eigenschaft „Beim Doppelklicken" auf einen Datensatz eine Ereignisprozedur aufgerufen wird.

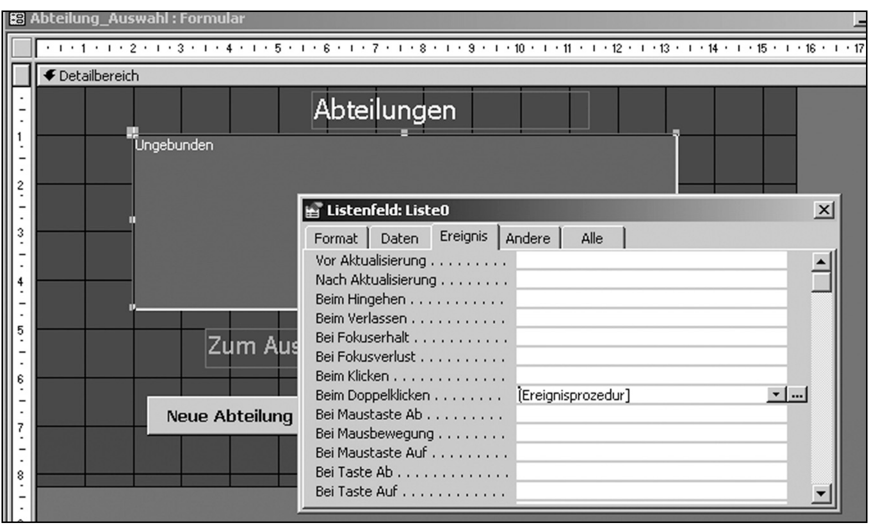


Abb. 7.12 Formular „Abteilung_Auswahl" (Entwickler-Ansicht „Ereignis")

Die Idee hierbei ist, dass durch Doppelklicken auf einen Datensatz (also eine Abteilung) ein neues Fenster aufgeht, wo diese Abteilung (und nur die) angezeigt wird. Dazu muss ein entsprechendes Formular existieren, welches dann sinnvollerweise an die Tabelle „Abteilung" gebunden ist. Durch den Doppelklick auf eine Abteilung im „Inhaltsverzeichnis" wird die dazugehörige Abt_ID ausgelesen und als „Filterbedingung" and das Formular der Abteilungen übergeben. So wird beim Öffnen des „hineingezoomten" Abteilungsformulars nur der doppelgeklickte Datensatz angezeigt. Der Aufruf und die Übergabe der Abt_ID erfolgt in der Ereignisprozedur, welche in Abb. 7.13 zu sehen ist.

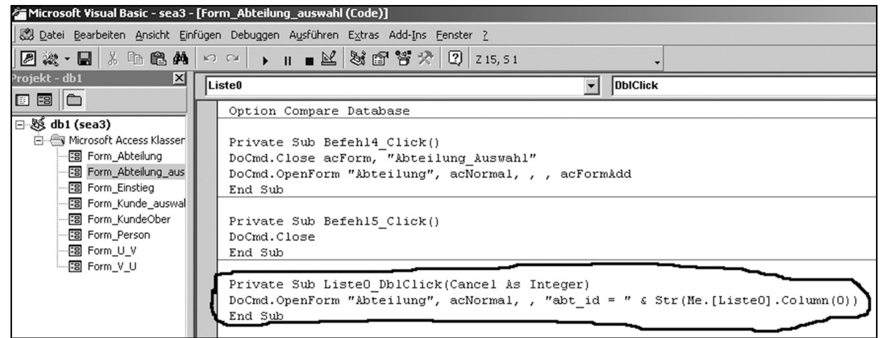


Abb. 7.13 Ereignisprozedur zum Aufruf des Abteilungs-Formulars

Wie man in Abb. 7.13 erkennt, wird das Formular „Abteilung" geöffnet, wobei der erste Parameter nach dem Namen angibt, dass eine „normale" Anzeige erfolgen soll, und danach kommt die Bedingung

```
"Abt_ID = " & Str(Me.[Liste0].Column(0))
```

Dies möchte ich kurz zum besseren Verständnis erläutern. Es handelt sich dabei um besagte „Filterbedingung" zum Öffnen der doppelgeklickten Abteilung. Ganz links steht durch Hochkomma eingeschlossen das Feld im Ziel-Formular (also im zu öffnenden Formular *Abteilung*). Dieses heißt offenbar „Abt_ID". Der Grund für das Hochkomma ist, dass die Methode `OpenForm` einen Übergabestring hier erwartet. Deswegen steht auch hinter dem *&* (was das Symbol für eine String-Addition darstellt) zuerst die Typkonvertierungsfunktion `Str(...)`. Danach kommt das aktuelle Objekt, welches den Namen des Formulars enthalten kann, oder einfach nur „Me", wenn das aufrufende Formular selbst das Objekt ist, auf welches sich die Methode dahinter bezieht. Die Methode nun heißt `[Liste0].Column(0)`. Es ist also eine verschachtelte Methode, denn `[Liste0]` ist zunächst selbst wieder ein Objekt. Es ist der Name des Listenfelds, auf das dopperlgeklickt wurde. Und dieses Feld hat 2 Spalten (die aus der Select-Anweisung aus Abb. 7.11). In VBA ist die Nummerierung der Spalten aber so, dass die 1. Spalte die Nummer 0 hat, die 2. Spalte die Nummer 1 und so weiter. Deswegen wird auf `Column(0)` verweisen, das ist die erste Spalte, da steht ja die *Abt_ID* drin. War diese doppelgeklickte *Abt_ID* z.B. die Nummer 3, so wird in dem Übergabestring schlicht stehen:

```
"Abt_ID = 3"
```

und damit hat das zu öffende Formular die ID des Datensatzes, der angezeigt werden soll. Doppelklickt man also auf einen Eintrag in der Auswahlliste, so öffnet sich z.B. die in Abb. 7.14 angezeigte Abteilung:

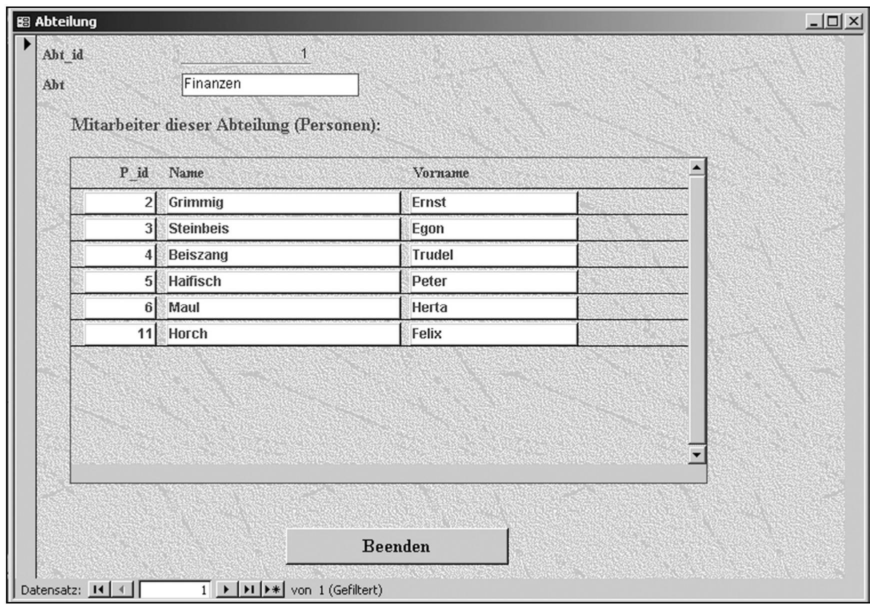


Abb. 7.14 Öffnen der doppelgeklickten Abteilung

Hier wurde auf die erste Abteilung im Listenfeld doppelgeklickt, so dass in die Abteilung Finanzen „hineingezoomt" wird. Man sieht jetzt die Abteilungs-ID nebst dem Namen der Abteilung (oben) und darunter eine Liste mit den Mitarbeitern dieser Abteilung. In dieser Maske kann aber nur der Name der Abteilung (hier: Finanzen) geändert werden. Es können weder Mitarbeiter hinzugefügt noch gelöscht oder sonst wie bearbeitet werden. Das wäre zwar technisch möglich gewesen, aber in der Anforderung an das Anwendungsprogramm stand ja, dass dies nicht gehen soll. Die Mitarbeiter der Abteilung kommen aus der Tabelle *Personen*. Bei dieser Bildschirmdarstellung handelt es sich also um die Realisierung einer 1:n-Beziehung. In MS-Access lässt sich so was relativ leicht realisieren. Man legt nämlich dafür einfach zwei Formulare an: Das „Hauptformular" für die „1-Seite" der Beziehung (hier ein Formular für die Tabelle „Abteilung") und ein zweites Formular für die Tabelle „Person". Man kann nun beim Anlegen von Formularen, die sich auf eine Tabelle beziehen, festlegen, welche Tabelle das sein soll (wieder unter „Datensatzherkunft" in der Entwurfsansicht des For-

mulars) und man kann weiter festlegen, wie die Darstellung ist: als „Einzelformular“ (d.h. pro Bildschirmseite ein Datensatz aus der Tabelle) oder als „Endlosformular“, dann werden mehrere Datensätze tabellenförmig angezeigt. Hier wurde für das Hauptformular (Abteilung) die Ansicht „Einzelformular“ und für das „Unterformular“ (Personen) die Endlosformular-Ansicht gewählt. Sind beide Formulare erstellt, kann man sie durch einfaches Drag-und-Drop zusammenführen. Dazu öffnet man das Hauptformular in der Entwurfsansicht und zieht aus der Liste der Formulare im MS-Access-Formularfenster das gewünschte Unterformular hinein. Da in der Beziehungsansicht schon eine 1:n-Beziehung der den Formularen jeweils zugrunde liegenden Tabellen definiert wurde, „weiß“ Access automatisch, welche Felder hier verknüpft sind und sorgt so dafür, dass immer nur die zu der aktuellen Abteilung zugehörige Mitarbeiter angezeigt werden.

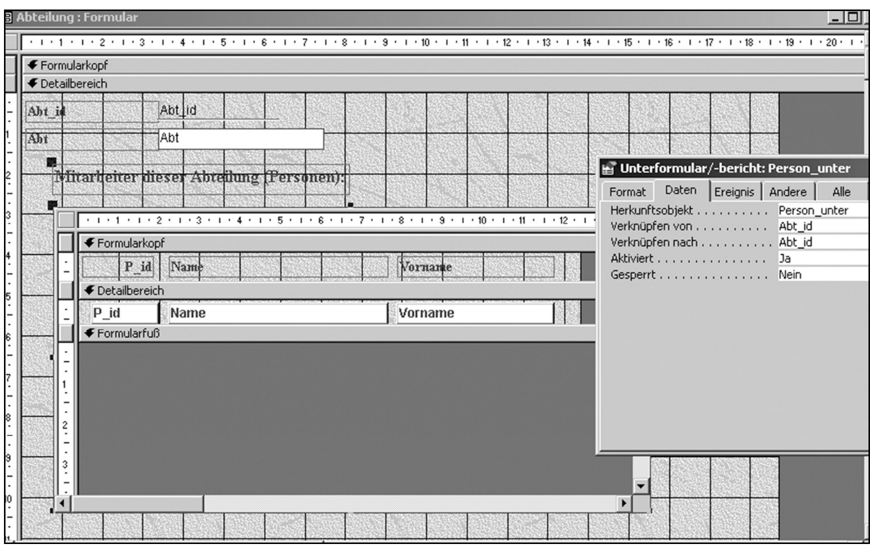


Abb. 7.15 Haupt- und Unterformular in der Entwurfsansicht

Wie man in Abb. 7.15 erkennt, heißt das eingebettete Formular „Person_unter“ und man kann auch sehen, dass die beiden Formulare über das Attribut *Abt_ID* verknüpft sind.

Diese Art der Darstellung sollte man also immer wählen, wenn man eine 1:n-Beziehung in einer Anwendung repräsentieren will. Es sei noch erwähnt, dass es möglich ist, auch mehr als nur ein Unterformular in ein Hauptformular einzubet-

ten. Hat man z.B. mehrere n-Seiten für ein und dieselbe 1-Seite, so kann für jede n-Seite ein eigenes Formular erstellt werden und durch Drag-und-Drop in das Hauptformular gezogen werden.

Nun wenden wir uns dem Formular zu, das geöffnet wird, wenn man im Einstiegsformular (Abb. 7.7) auf die Schaltfläche „Mitarbeiter (Personen) / Vorgesetzte" klickt. Dann soll sich ein Formular öffnen, das alle Mitarbeiter anzeigt.

Person

P_id	Name	Vorname	Abt_id	Abt		
1	Nimmersatt	Karl1	2	Forschung	Vorgesetze	Untergebene
2	Grimmig	Ernst	1	Finanzen	Vorgesetze	Untergebene
3	Steinbeis	Egon	1	Finanzen	Vorgesetze	Untergebene
4	Beiszang	Trudel	1	Finanzen	Vorgesetze	Untergebene
5	Haifisch	Peter	1	Finanzen Forschung ccc	Vorgesetze	Untergebene
6	Maul	Herta	1		Vorgesetze	Untergebene
7	Schleuder	Kurt	2	Forschung	Vorgesetze	Untergebene
8	allwissend	Fritz	4	ccc	Vorgesetze	Untergebene
9	Prasser	Hugo	2	Forschung	Vorgesetze	Untergebene
10	Besserwiss	Sonja	2	Forschung	Vorgesetze	Untergebene
11	Horch	Felix	1	Finanzen	Vorgesetze	Untergebene
12	Meier-Gerske	Hildtrud	4	ccc	Vorgesetze	Untergebene
15	Neumann	Gustav	4	ccc	Vorgesetze	Untergebene
(AutoWert)			0		Vorgesetze	Untergebene

Beenden

Datensatz: 4 von 13

Abb. 7.16 Formular Mitarbeiter

In Abb. 7.16 sieht man, wie dieses Formular in User-Ansicht aussieht. Es wurde hierfür eine Endlosansicht erstellt. Zunächst ist eine Personen-ID vorhanden (P_ID) die nicht veränderbar ist (da vom System vergeben). Daneben stehen die zwei Felder mit Namen und Vornamen des Mitarbeiters, gefolgt von der (wieder nicht veränderbaren) *Abt_ID* der daneben auswählbaren Abteilung. Es ist also so, dass die Zuordnung eines Mitarbeiters zu einer Abteilung hier durch Auswahl erfolgt, d.h. die Abteilungen müssen schon angelegt sein. Ganz rechts schließlich stehen zwei Schaltflächen, durch deren Betätigung jeweils eine Tabelle mit den Vorgesetzen bzw. Untergebenen des aktuellen Datensatzes angezeigt wird.
In Abb. 7.17 ist die Entwurfsansicht dieses Formulars zu sehen, wobei die Eigenschaften des „Kombinationsfelds" (also dem Auswahlfeld für die zugeordnete Abteilung) sichtbar sind. Ohne auf technische Einzelheiten weiter einzugehen

wurden hier die Abteilungsdaten über eine Select-Anweisung (siehe „Datensatzherkunft“ im Ausschnitt des Eigenschaftsfensters) in das Auswahlfeld gebracht.

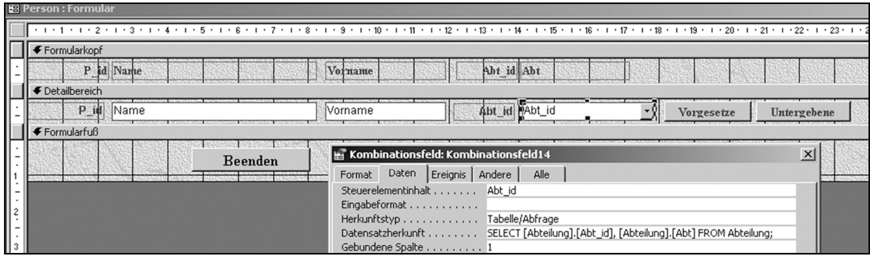


Abb. 7.17 Formular Mitarbeiter (Entwurfsansicht)

Interessanter ist, wie die reflexive *n:m*-Beziehung auf die Tabelle *Person* realisiert ist. Hierzu schauen wir zunächst was passiert, wenn man auf die Schaltflächen „Vorgesetzte“ oder „Untergebene“ drückt (Abb.7.18).

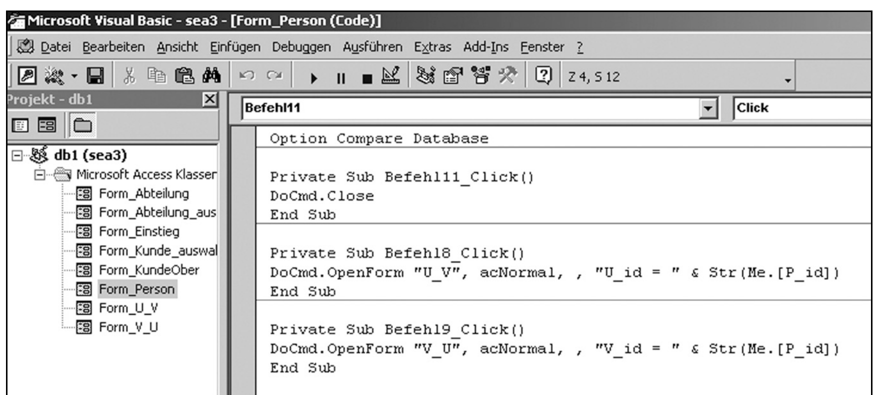


Abb. 7.18 VBA-Programme der Schaltflächen Untergebene/Vorgesetzte

Bei Drücken der Schaltfläche „Vorgesetzte“ wird die Ereignisprozedur `Befehl18_Click()` und bei der Schaltfläche „Untergebene“ die Ereignisprozedur `Befehl19_Click()` ausgeführt. Betrachten wir den ersten Fall. Da wird also ein Formular mit Namen *U_V* geöffnet, wobei die aktuelle Mitarbeiter-ID (P_ID) als Filterbedingung an das Formular übergeben wird.
Drückt man z.B. beim ersten Mitarbeiter auf die Schaltfläche „Vorgesetzte“, so öffnet sich folgendes Formular (Abb. 7.19):

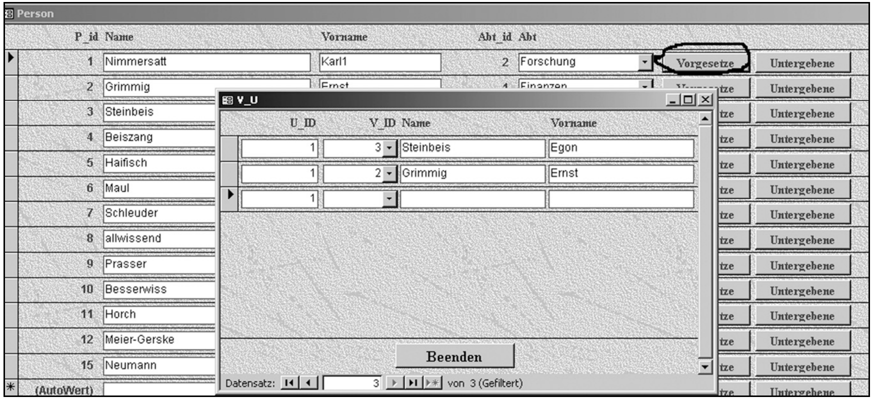


Abb. 7.19 Vorgesetzte

Es öffnet sich daraufhin also das Formular V_U mit den zugehörigen Vorgesetzten des Herrn Nimmersatt aus der gleichen Mitarbeiterliste. Wie man hier sieht, kann man diese Vorgesetzte auch bearbeiten (hinzufügen, ändern, löschen). Zur Kontrolle wird in der Spalte U_ID die aktuelle P_ID (hier von Herrn Nimmersatt) angesezeigt, der hier ja jetzt Untergebener (daher U_ID) ist. Daneben unter V_ID steht die P_ID aus der Tabelle Person, welche als Vorgesetzte von Herrn Nimmersatt zugeordnet sind. Wie kommen diese Einträge zustande? Das Formular V_U basiert auf einer SQL-Abfrage, die das erledigt:

```
SELECT V_U.U_ID, V_U.V_ID, Person.Name, Person.Vorname
FROM Person INNER JOIN V_U ON Person.P_id = V_U.V_ID
ORDER BY V_U.U_ID;
```

Es werden damit also in der Beziehungstabelle *V_U* alle zu P_ID (Nimmersatt) passenden Einträge gesucht und von denselben dann in der Tabelle *Person* Name und Vorname herausgeholt und das Ganze in dem Formular V_U angezeigt.

Damit lernen wir auch hier, wie grundsätzlich *n:m*-Beziehungen behandelt werden: Es wird auf der n-Seite ein Datensatz angezogen (hier Herr Nimmersatt). Dann wird wie bei einer *1:n*-Beziehung vorgegangen, wobei die 1-Seite jetzt ein fixierter Datensatz der n-Seite der *n:m*-Beziehung ist. Für die m-Seite wird jetzt die Beziehungstabelle (hier V_U) genommen. Und weil hier nur Fremdschlüssel drinstehen, muss die passende weitere Information transitiv aus der „eigentlichen" m-Seitentabelle geholt werden (da hier reflexiv, ist das wieder die Tabelle *Person*). Ob es sich dabei dann um eine reflexive Beziehung (wie hier) oder um zwei verschiedene Tabellen, wie Kunden und deren Bearbeiter handelt (s.u.), die

in einer *n:m*-Beziehung zueinander stehen, ist völlig gleichgültig. Das Vorgehen ist das selbe.
Für die Untergebenen ist das Vorgehen übrigens analog den Vorgesetzten und braucht daher nicht noch mal erläutert zu werden.

Der Vollständigkeit halber seien noch die Formulare für den Fall angegeben, dass ein User im Einstiegsmenu (Abb. 7.7) die Schaltfläche „Kunden“ betätigt. Es erscheint wieder eine Auswahlliste mit allen Kunden, und durch Doppelklicken wird ein bestimmter Kunde dann ausgewählt:

Abb. 7.20 Auswahlliste Kunden

Wie bei den Abteilungen wird durch Doppelklicken auch hier die jeweilige ID des angeklickten Datensatzes als Filterbedingung an ein anderes Formular übergeben. Dieses Formular heißt „Kunden“ und öffnet sich nach dem Doppelklick (Abb. 7.20).
Doppelklickt man z.B. auf Herrn Bettelstab, dann wird dessen Kundeneintrag im Kundenformular angezogen und steht zur Bearbeitung zur Verfügung. Es werden auch gleich alle zugehörigen Mitarbeiter, die diesen Kunden betreuen, mit angezeigt. Im Unterschied zu den Einträgen in der Abteilungsmaske kann man hier aber die zugeordneten Einträge des Unterformulars (also die Mitarbeiter) jetzt bearbeiten: Es können welche gelöscht, geändert oder neu hinzugefügt werden. Im Normalfall ist die Editierbarkeit von Unterformularen als „Default“ eingestellt. Man musste in dem Abteilungsformular diese Bearbeitungsmöglichkeit extra abschalten, was leicht über den Eintrag „Bearbeiten zulassen (j/n)“ des

Eigenschaftsfensters des Unterformulars möglich ist. Standardmäßig ist aber dieser Eintrag –wie gesagt- auf „ja“ gesetzt.

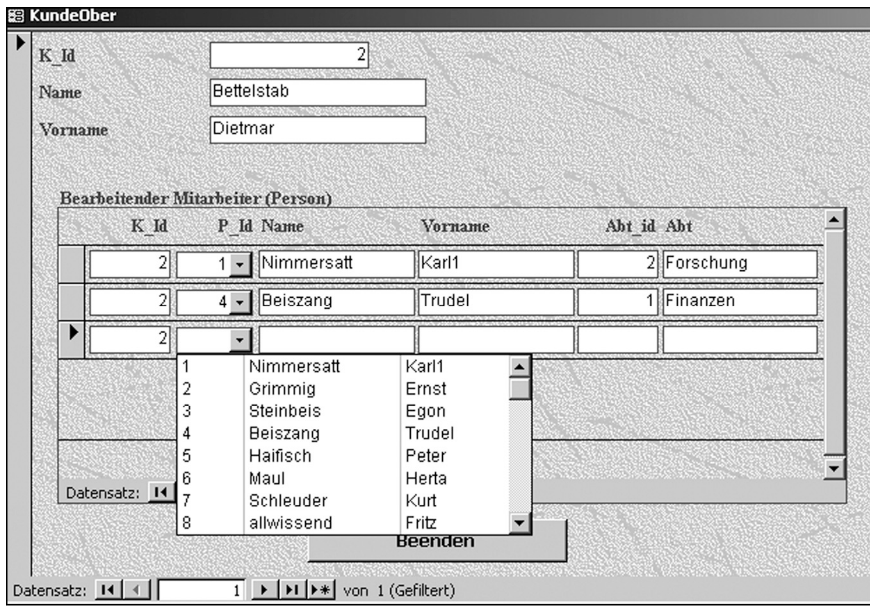


Abb. 7.21 Formular Kunden

Im Unterformular in Abb. 7.21 wurde zu Demonstrationszwecken im letzten „leeren“ Datensatz auf das Kombifeld geklickt. Es öffnen sich dann alle Mitarbeiter und durch Klicken auf einen davon würde dieser dann auch dem Herrn Bettelstab als Bearbeiter zugewiesen.
Da es sich ursprünglich um eine *n:m*-Beziehung zwischen Kunden und bearbeitenden Mitarbeitern handelt, sei hier, wie bei der reflexiven Vorgesetzten / Untergebenen-Beziehung, noch der SQL-Befehl angegeben, auf den sich das eingebettete Unterformular der Bearbeiter bezieht (in MS-Access im Eigenschaftsfenster des Unterformulars unter „Datenherkunft“). Er lautet:

```
SELECT  K_P.K_Id, K_P.P_Id, Person.Name, Person.Vorname,
        Person.Abt_id, Abteilung.Abt
FROM    (Abteilung
        INNER JOIN Person
        ON Abteilung.Abt_id = Person.Abt_id)
        INNER JOIN K_P ON Person.P_id = K_P.P_Id;
```

Analog der reflexiven Beziehung werden also die dem Kunden zugehörigen Mitarbeiter zunächst als K_ID aus der Beziehungstabelle K_P geholt und dann mit den passenden K_ID's die Namen, Vornamen etc. aus der Tabelle Person hinzugefügt.
Man kann übrigens für die Benutzeransicht alle ID's sowohl im Hauptformular als auch im Unterformal völlig unterdrücken, da diese ID's für ihn uninteressant sind. Es ist jedoch in der Testphase oftmals hilfreich, diese ID's noch zu sehen. Nach Abschluss der Testphase würde ich dann diese ID-Spalten alle unterdrücken; das ist leicht zu machen, man braucht dafür in der Entwickleransicht lediglich die entsprechenden Spaltenbreiten auf den Wert *0 cm* zu setzten.
Damit ist das Kapitel „Anwendungsdesign" für diesen Kompaktkurs abgeschlossen. Es könnte allein darüber ein dickes Buch geschrieben werden, doch mit dem hier präsentierten Beispiel sind meiner Erfahrung nach die häufigsten Fälle abgedeckt und es können darauf aufbauend andere Anwendungsfälle in der Regel realisiert werden.
Es gibt natürlich viele weitere Methoden und Verfahren, wie z.B. Xtreme-Programming, Unified Processes (UP), Fusion etc. um nur ein paar zu nennen.

Übungen zum Selbsttest

1. Entwickeln Sie passende Bildschirmmasken zu dem UML-Diagramm aus Abb. 5.14.

Lösungen der Übungen zum Selbsttest

Kapitel 2:

1. a) Initialisierungsphase
   b) Fachkonzept
   c) Fachkonzept
   d) Fachkonzept für semantisches Datenmodell, DV-Entwurf für logisches Datenmodell
   e) Planung im Fachkonzept, Durchführung in der Testphase.

Kapitel 3:

1. Software-Architekturen sollen einen Standard bieten, mit dem man die Softwareentwicklung einerseits vereinheitlichen will und andererseits einmal benutzte Komponenten (nicht nur Programm-Module, sondern auch Entwicklungs-Muster auf konzeptioneller Ebene) wieder verwenden möchte. Unter einer *Software-Architektur* verstehen wir eine Spezifikation über die Teile des zu entwickelnden Systems, welche ihre Konnektoren sowie die Regeln für die Interaktion dieser Konnektoren mit den Systemteilen enthält. Ein Konnektor umfasst dabei z.B. alle Schnittstellen (auch auf verschiedenen Abstraktionsebenen).

2. Das Ziel einer modellgetriebenen Architektur (MDA) besteht letztlich in der automatischen Generierung des Quellcodes einer Applikation aus dem Entwurfsmodell heraus. Die Idee ist dabei, dass Änderungen nie an der Applikation selbst vorgenommen werden sollen, sondern immer am Modell und dann neu generiert wird. Dadurch wird Konsistenz zwischen dem Modell und der Applikation erzwungen.

3. Das Hauptproblem in MDA besteht darin, dass ein abstraktes Modell in eine konkrete Applikation übergeführt werden soll. Um jedoch eine konkrete Applikation zu erzeugen, sind gewisse Parameter wie z.B. die Software-Plattform (Betriebssystem etc.) notwendig, die in der abstrakten Form so nicht nötig sind. In MDA betrifft dies den Übergang vom plattformunabhängigen Modell (PIM) zum plattformspezifischen Modell (PSM). Das Mapping führt genau diesen Übergang durch.

Kapitel 4:

1. Ein semantisches Datenmodell muss den Ist- und Sollzustand sowie eine Beschreibung der Datenstrukturen beinhalten, und zwar so, dass sowohl Kunde wie auch Softwareentwickler es verstehen kann. Dies geschieht daher in der Regel verbal. Es ist Teil des Pflichtenhefts, und dieses ist Teil des DV-Fachkonzepts, also der 2. Phase des Wasserfall-modells.

2. a) Beispiel für ein Angebot:

Fa. Nimmersatt GmbH & Co. KG, in der Abzock 8, 60321 Frankfurt am Main

An Herrn August 1.2.2007
Abt. DAU
Loser GmbH & Co KG
Ernst-August-von-Halmackenreuther-Str. 11
43233 Buhl

Angebot Entwicklung des Programms zur elektronischen Verwaltung von Mitarbeitern

Sehr geehrter Herr August,

Bezugnehmend auf Ihre Anfrage vom 22.10.1992 unterbreite ich Ihnen folgendes **Angebot**:

Tätigkeit	**Aufwand (MT)**
Problemanalyse und Erstellung eines Pflichtenhefts zur Beschreibung der geplanten Anwendungsentwicklung	2
Entwurf der erweiterten Datenstrukturen und Algorithmen/Funktionen gemäß des erstellten Pflichtenhefts	2
Implementierung der Software incl. Modultest	2

Der Mann-Tag wird mit EUR 1.000,- abgerechnet.

Entwurfs- bzw. Implementierungsfehler, welche unsererseits verursacht wurden, werden innerhalb einer Anwendungstestphase durch Ihre Anwender bis zu 4 Wochen nach Programmübergabe von uns kostenfrei umgehend beseitigt.

Bei Auftragserteilung vor dem 01.04.2007 können die Arbeiten innerhalb von 6 Wochen erledigt werden.

Ich hoffe, Ihnen mit diesem Angebot gedient zu haben.

Mit freundlichen Grüßen

Egon Nimmersatt, Geschäftsführer.

b) Beispiel für ein einfaches Pflichtenheft:

<Projektlogo>

Pflichtenheft

Einführung einer Verwaltungssoftware

Projektnummer : 0815

Auftraggeber : Global AG

Hugo Ernst, Telefonnr. 9876 / 54 32 10

Auftragnehmer : Software GmbH

Mike Dettmann, Telefonnr. 0123 / 45 67 89

Stand: 22.07.06

1. Einleitung

Im Zuge der Neustrukturierung der Global AG werden auch die Verantwortlichkeiten der Hauptabteilung Personal neu geregelt.
Um dieses Vorhaben zu realisieren, muss eine neue Verwaltungssoftware eingesetzt werden.

2. Beteiligte Organisationen/ Firmen

Organisationseinheit	Firma	Ansprechpartner
HA Personal	Global AG	Hugo Ernst
Softwareentwicklung	Software GmbH	Mike Dettmann

3. Die Analyse des Hauptprozesses

3.1 IST – Situation

Nach fachlicher Prüfung durch die Verantwortlichen von Personal werden Personaldaten in Excel-Tabellen eingegeben. Diese Eingaben werden dezentral in den Hauptabteilungen „Personal" der Organisationseinheiten vorgenommen und auf dezentrale Datenträger gespeichert.
Auf Anfrage bzw. nach Terminvorgaben werden Listen nach den gewünschten Inhalten manuell erstellt.

4. Ziele des Projekts

4.1 Fachliche Ziele

Unter dem Aspekt eines straffen Zeitplans besteht ein wesentliches, fachliches Hauptziel in der möglichst schnellen Einführung eines zentralen Personalverwaltungssystems, dass die Anforderungen der Global AG ausreichend berücksichtigt.

4.2 SOLL – Situation

Zugriffsberechtigungen

Mitarbeiter von Personal müssen Zugriffsberechtigungen und Ansichtsberechtigungen erhalten, damit sie Personaldaten qualifiziert festlegen und eingeben können. Die Zugangsberichtigungen müssen flexibel sein, damit sie von Key-Usern geändert werden können, falls dies erforderlich sein sollte.
Es werden die 4 Hierarchiestufen (Geschäftsbereich, Hauptabteilung, Abteilung, Arbeiter) berücksichtigt.
Individuelle Abfragen ermöglichen eine Zusammenstellung nach Auswahlkriterien (z.B. Außen-/Innendienst; freie/feste Mitarbeiter; Personal- und Einsatzdaten)

Zum Geschäftsjahresende erfolgt automatisch die Erhebung der gesetzlichen Statistik und der Archivierung ausgeschiedener Mitarbeiter

Für jeden Geschäftsbereich ist die Software in der Landessprache und nach den gesetzlichen Vorschriften anzupassen.

4.3 Einsatzstufen

Einsatztermin für die Verwaltungssoftware ist der 01.01.2007. Ende des Projektes ist der 31.12.2006

5. Einweisung, Schulung

Schulungsdokumente mit Berücksichtigung von Benutzer- und Systemdokumentation werden durch den Auftragnehmer erstellt. Eine ca. 30 minütige Einweisung der Benutzer erfolgt in Kursen mit max. 10 Teilnehmer.

6. Sicherheit

- Zuordnung der Zugriffsrechte zu den Anwenderklassen, insbesondere die Unterscheidung zwischen Endbenutzer und Systembetreuer
- Schutz vor unbefugtem Zugriff
- Organisation des Notbetriebs, Fehler- und Störungshandhabung
- Backup-Strategie

7. Weiteres Vorgehen

7.1 Aufgaben des Auftragnehmers

Hauptaufgaben des Auftragnehmers liegen vor allem in der Konzeption von Schnittstellen, der Altdatenübernahme und Bereinigungen. Auch die Beratung über den Funktionsumfang der neuen Verwaltungssoftware sowie notwendiges Customizing zählen zu den primären Aufgaben des Auftragnehmers.

7.2 Aufgaben des Auftraggebers

Der Auftraggeber stellt für die Tätigkeiten vor Ort die Kennwörter und Zugänge zu den betreffenden Geräten, Räumlichkeiten, Applikationen, Systemen und Unternehmensnetzwerken sowie insbesondere zum Internet zur Verfügung.

Für etwaige Bereinigungen von Altdaten in den Personaltabellen zeichnet ebenfalls der Auftraggeber verantwortlich. Auch die Archivierung der Altdaten in den Altsystemen ist Aufgabe des Auftraggebers.

Für die Projektdauer werden den im Unternehmen des Auftraggebers eingesetzten Mitarbeitern, falls notwendig, Fremdfirmenausweise zur Verfügung gestellt.

Zu den Aufgaben des Auftraggebers gehören darüber hinaus die Beratung und Mitarbeit bei der Erstellung und Umsetzung eines Berechtigungskonzepts sowie die Erstellung eines Archivierungskonzeptes (unter Einhaltung aktueller gesetzlicher Vorschriften).

8. Zukünftige Supportorganisation

Die für das Personal-Verwaltungs-System verantwortliche Supportorganisation muss definiert werden. Eine Konzeption / einen Vorschlag für Betrieb und Support liefert der Auftragnehmer. Für die Umsetzung und Realisierung der zukünftigen Strukturen ist der Auftraggeber verantwortlich.

9. Projektplanung

Endtermin ist der 31.12.2006. Die Inbetriebnahme erfolgt am 01.01.2007
Die Tests sind bis zum 10.12.2006 und der Rollout bis zum 20.12.2006 abzuschliessen,
Teilabnahmen sind am 15. des laufenden Monats fällig. In diesem Zusammenhang werden der Statusbericht oder anderer Leistungsnachweise aktualisiert.
Änderungswünschen sind schriftlich zu vereinbaren.

6. Unterschriften

Auftragnehmer: __________ ________________________________
(Datum)

Auftraggeber: __________ ________________________________
(Datum)

Auszug aus dem
Semantisches Datenmodell (am Beispiel orientiert)

Der Konzern „Global AG“ ist in mehrere Geschäftsbereiche (GB) strukturiert, welche durch ihren Namen und Ihren Sitz (Land, Straße, PLZ, Ort, Telefon, Amtssprache, Homepage) gekennzeichnet sind. Jeder Geschäftsbereich unterteilt sich in Hauptabteilungen, welche wiederum in Abteilungen gegliedert sind. Jede Hauptabteilung und jede Abteilung hat einen Namen, ein Kürzel und eine Tätigkeitsbeschreibung. Jede Abteilung enthält einen oder mehrere Mitarbeiter. Jeder Mitarbeiter ist durch Personalnummer, Name, Telefon, Eintrittsdatum in das Unternehmen und seine Anschrift(Straße, PLZ, Ort) gekennzeichnet. Weitere enthaltene Attribute der Mitarbeiter sind der Einsatz im Innen- bzw. Aussendienst, feste bzw. freie Mitarbeit und die Führungspositionen durch Vorgesetzte bzw. Untergebene.

c) Phasenmodell mit kurzer Erläuterung am Beispiel der Global AG

Zur Darstellung wähle ich das V-Modell aufgrund seiner linearen Nachvollziehbarkeit für Informatiker und Kunden. So wird z.B. zwischen konstruktiven Tätigkeiten und Tätigkeiten zur Qualitätssicherung unterschieden.

Phase 1 Spezifikation
Hier erfolgt die Analyse des IST-Zustandes durch eine Problembeschreibung. Weiterhin wird mit dem SOLL-Zustand das eigentliche Ziel des Projektes „Verwaltungssoftware“ durch die SWG der Global AG inform eines Angebotes vorgeschlagen. Dazu werden Ziele festgelegt, Kosten abgeschätzt und die Projektplanung beschrieben.
Nach Einigung wird ein Pflichten(Lasten-)heft erstellt, welches die Basis des Projektes bildet und sowohl von der Global AG als auch von der SWG unterschrieben wird.

Phase 2 Systementwurf
Die SWG entwirft das detaillierte Gerüst der Verwaltungssoftware basierend auf dem Pflichtenheft. Der Entwurf wird der Global AG präsentiert und von dieser bestätigt. Dies ist der erste Teil der eigentlichen Entwicklung der Verwaltungssoftware.

Phase 3 Komponentenentwurf
Ähnlich dem Systementwurf wird jetzt die Verwaltungssoftware anhand der Funktionsablaufpläne analysiert und strukturiert. Der Entwurf wird der Global AG präsentiert und von dieser bestätigt.

Phase 4 Implementierung
Aufbauend auf die Entwürfe aus den Phasen 2 und 3 wird jetzt programmiert. Das Resultat wird der Global AG als ausführbares Programm präsentiert und bestätigt.

Phase 5 Komponententest
Die Komponententests werden durch die SWG durchgeführt. Dabei werden einzelne Programmkomponenten auf logische Konsistenz und Übereinstimmung mit dem DV-Design geprüft.

Phase 6 Integrationstest
Nach erfolgreicher Implementierung und Tests der Komponenten werden die sprachspezifischen Versionen der GB mit unterschiedlicher Amtssprache von Vertretern der jeweiligen GB geprüft und bestätigt.

Phase 7 Validierung
Nach Abschluss der integrativen Tests erfolgt die Installation in einer produktionsnahen Umgebung. Nach Freigabe durch die Global AG wird die Verwaltungssoftware in den Standorten ausgerollt (installiert).

3. Gesamtarbeitszeit : 8n Stunden.

Verbrauchte Gesamtarbeitszeit für einen „Kommunikationsevent“ (2 Personen reden miteinander): 2 x 6 Min. = 12 Min = 0,2 Std.

Anzahl Mitarbeiter	**Anzahl Kommunikationsevents**
1	0
2	0 + 1 = 1
3	0 + 1 + 2 = 3
4	0 + 1 + 2 + 3 = 6
...	...
n	0 + 1 + 2 + ... + (n-1) = $\frac{(n-1)n}{2}$

a) Verbrauchte Zeit für n Mitarbeiter = Gesamtarbeitszeit , d.h.

$$\frac{(n-1)n}{2} \bullet 0{,}2 = 8n$$

Lösung: n=81, d.h. ab 81 Mitarbeitern wird nur noch geredet.

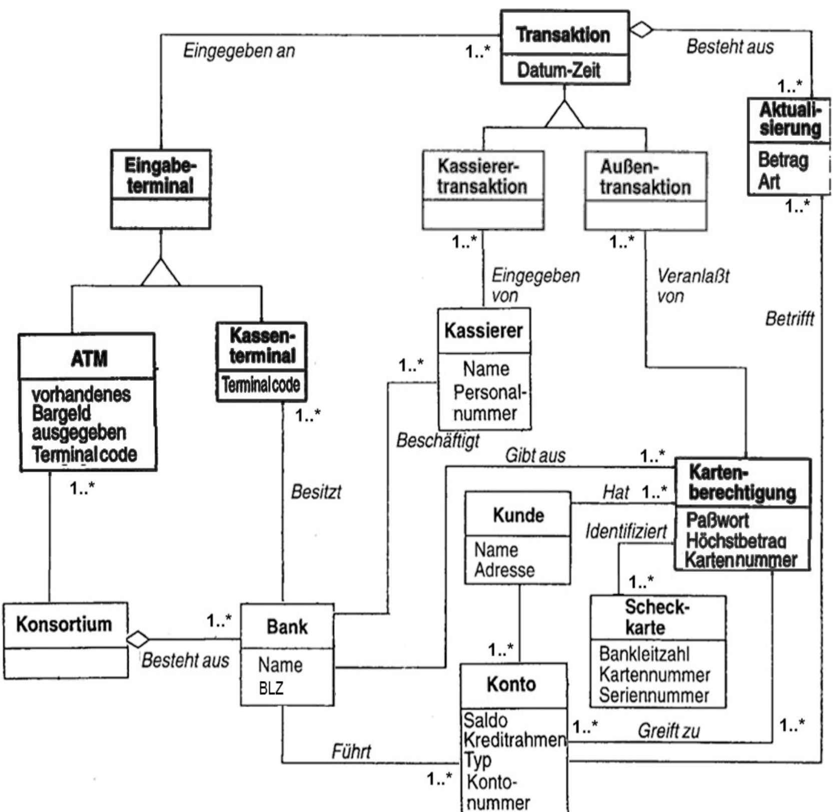
b) Die Differenz aus verbrauchter Zeit für n Mitarbeiter und der Gesamtarbeitszeit wird minimiert. D.h. dass diese Differenz abgeleitet und Null gesetzt wird (MinMax-Problem):

$$\frac{d}{dn}[\frac{(n-1)n}{2} \bullet 0{,}2 - 8n] = 0$$

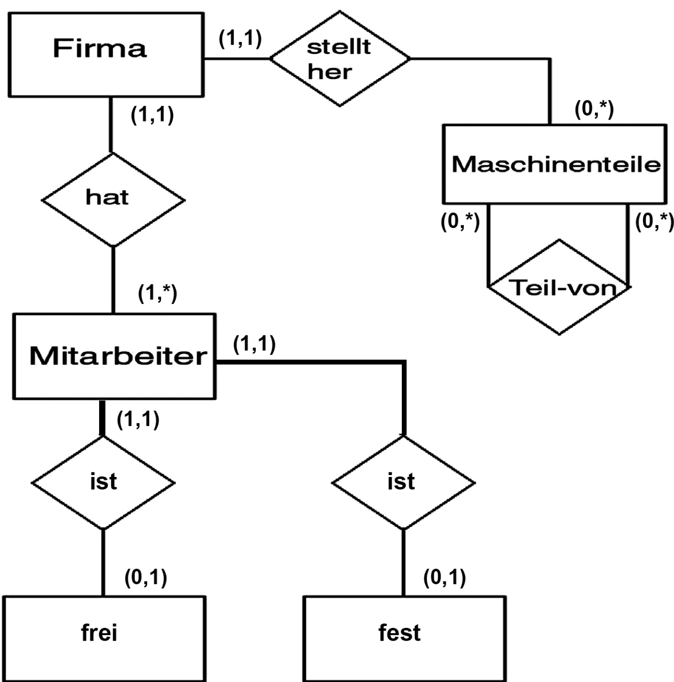
Lösung: n = 40 (gerundet), d.h. bei 40 Mitarbeitern wird ca. genauso viel geredet wie gearbeitet.

Kapitel 5:

1. Eine Möglichkeit wäre (Methoden wurden weggelassen):



2. Umsetzung des UML-Diagramms nach „rein relationales“ ERD:



3. In allen 4 Fällen werden jeweils die Basisklasse und alle beteiligten Subklassen durch Tabellen repräsentiert. Dabei werden alle Attribute so übernommen, wie sie in den jeweiligen Klassen angegeben sind. Alle Subklassentabellen erhalten als Fremdschlüssel den Primärschlüssel der Basisklasse. In den Subklassentabellen wird dieser Fremdschlüssel dann selbst zum Primärschlüssel erhoben. Dadurch ist gewährleistet, dass eine (1,1):(0,1)-Beziehung zwischen Basisklasse und jeder Subklasse vorliegt. Des weiteren kann man z.B. folgendes vornehmen:

{incomplete, overlapping}
Hier muss nichts weiter gemacht werden

Für die nachfolgenden 3 Fälle wird für jede Subklassentabelle in der Basistabelle je ein weiteres „Diskriminator-Attribut“ hinzugefügt. Der Wertebereich beträgt {0,1}. Als Defaultwert wird überall 0 eingetragen. Wenn nun ein Datensatz der Basistabelle von einem Datensatzes einer

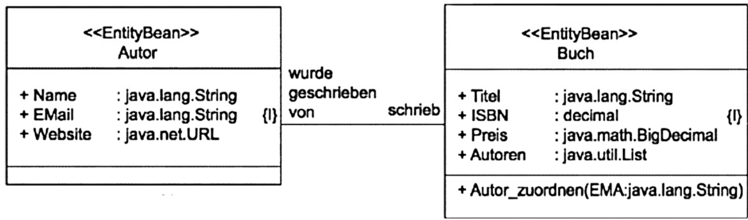
(oder mehrerer) Subtabellen referiert wird, dann wird der Wert des Diskriminator-Attributs dieses Datensatzes in der Basistabelle auf 1 gesetzt.

{complete, overlapping}
Hier muss offenbar jeder Datensatz der Basistabelle einen zugeordneten Datensatz in einer oder mehreren Subtabellen besitzen. Daher muss hier als Bedingung geltet: Die Summe aller Diskriminator-Attributswerte der zugeordneten „Sub-Datensätze“ muss größer als 0 sein.

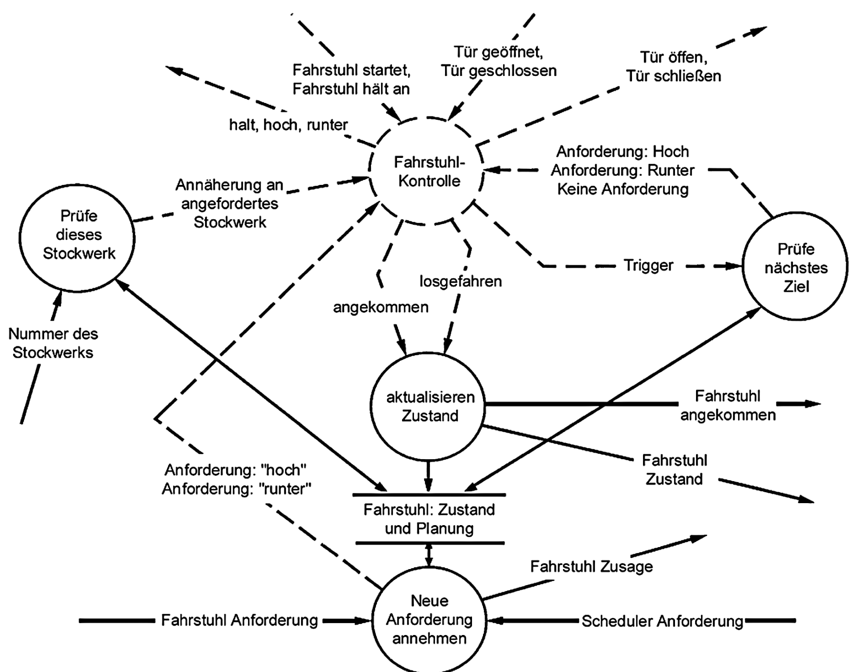
{incomplete, disjoint}
Hier darf maximal ein Datensatz in einer der Subtabellen einem Basis-Datensatz zugeordnet sein. Daher muss jetzt die Summe aller Diskriminator-Attributswerte der zugeordneten „Sub-Datensätze“ kleiner oder gleich 1 sein

{complete, disjoint}
Hier muss jeder Datensatz der Basistabelle einen zugeordneten Datensatz in genau einer Subtabelle besitzen. Daher muss hier die Summe aller Diskriminator-Attributswerte der zugeordneten „Sub-Datensätze“ genau gleich 1 sein

4. Das UML-Diagramm aus der Aufgabenstellung ist (normalerweise) einem weitaus geringerem Wandel unterlegen als das in der unteren Abbildung, wo das PIM aus der ursprünglichen Abbildung plattformabhängig - über entsprechende Annotationen - an die Plattformen Java bzgl. der Sprache und EJB (Enterprise Java Beans) als Middleware gebunden wurde:



5. Eine Möglichkeit wäre:



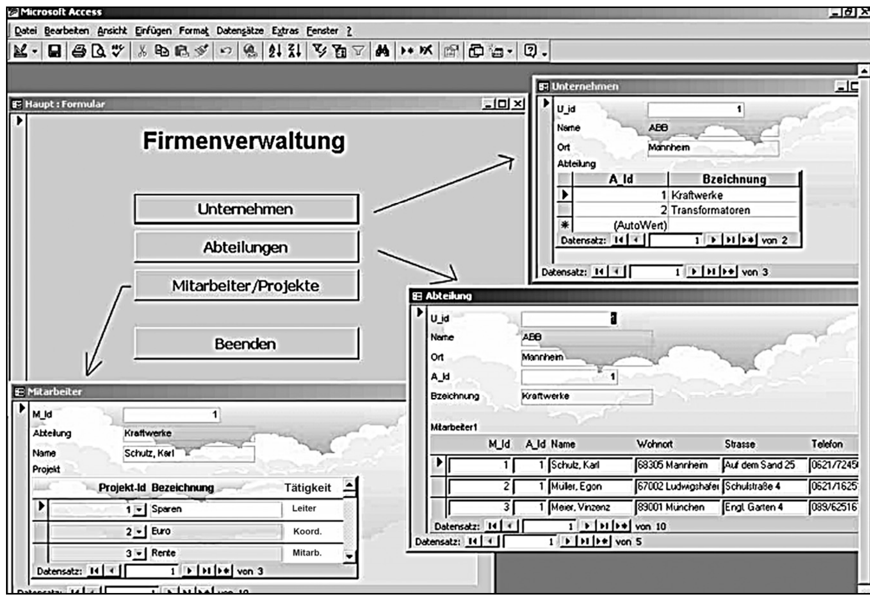
Kapitel 6:

1. Ein Entwurfsmuster beschreibt eine bewährte Schablone für ein Entwurfsproblem. Es stellt damit eine wiederverwendbare Vorlage zur Problemlösung dar. Entstanden ist der Ausdruck in der Architektur, von der er für die Softwareentwicklung übernommen wurde

2. Die Abbildungen 6.1 und 6.2 zeigen, worauf es bei der Verwendung von Entwurfsmustern in der UML-Klassenbildung ankommt: Es sollten die Klassen so gewählt werden, dass –ähnlich wie bei einer Vererbung auf Attributsebene- gleiche Objektstrukturen in einer eigenen Klasse zusammengefasst werden, sodass, wie im Beispiel der genannten Abbildungen, nur noch in den sich wirklich unterscheidenden Komponenten (hier: Gehaltsermittlung) eigene Klassen angelegt werden sollten.

Kapitel 7:

1. Eine Möglichkeit wäre:



Weiterführende Literatur:

Allgemein:

Zöller-Greer, Peter: *Software-Architekturen: Grundlagen und Anwendungen,* Verlag composia, 2010.
König, W. u.a.: *Taschenbuch der Wirtschaftsinformatik und Wirtschaftsmathematik.* Verlag Harri Deutsch, Frankfurt am Main, 1999
Rumbaugh, J. u.a.: *Object Oriented Modeling and Design*, Prentice-Hall Inc., 1991

Für Realzeitsystme:

Gomaa, H.: *Software Design Methods for Concurrent and Real-Time-Systems.* Addison-Wesley 1999

Für Entwurfsmuster:

Gamma, E. Helm, R., Johnson, R. and Vlissides, J.: *Design Pattern*, Addison-Wesley 1995/2006

Für Datenbanken:

Schubert, M.: *Datenbanken*, Teubner, Stuttgart, 2007

Für Künstliche Intelligenz:

Zöller-Greer, Peter: *Künstliche Intelligenz*, Composia Verlag, Wächtersbach 2010

Index:

abgeleitete Klassen 74
abstrakte Klassen 77
Aggregation 79
Analyse 36
Anforderungsanalyse 38
antisymmetrisch 79
Anwendungsfall-Diagramm 42
Assoziationen 69
Assoziationsklasse 72
Assoziationstyp 47, 59
atomare Klasse 56
Attachments 66
Attribut 47, 48

Basisklasse 74
Benutzertest 15
Beziehung 48
Beziehungsklassen 58
BMM 65
Boeing-Hatleys-Methode 81
Booch Modeling Method 65

CASE 7
complete 76
Controls 81
CORBA 34

Data Dictionary 81
Data Repository 81
Datenformat 55
Datenkapselung 55
Datenmodell 55
Datenmodellierung 54
Datenstruktur 55
Datentyp 57
Design 36
disjoint 76
disjunkt 76
Diskriminator 76

Domain 47, 57
Domainklasse 57
DV-Entwurf 13
DV-Konzept 13
dynamische Datenmodellierung 52

Echtzeit-Design 82
Echtzeit-Strukturierte-Analyse 81
elementare Attribute 57
endlicher Zustandsmaschinen 81
Entität 48, 58
Entitätsklasse 58
Entity 47
Entity Relationship Diagramm 47
Entity set 47
Entwicklungszyklus 6
Entwurfsmuster 105
ERD 47
ER-Diagramm 47,60
ERM 47
Exemplare 56

Fachkonzept 38
Feinkonzept 13
Funktion 59
  mengenwertige 59
  partiell definierte 59
  total definierte 59

Generalisierung 74
Grobkonzept 13, 38

Hierarchien 62

Icon 66
Identifikator 58
Implementierung 14
incomplete 76
Inhalte 66

Initialisierung 11
Installation 15
Instanzen 56
Instanzendiagramm 67
Integrität 55
Integritätsbedingungen 55
inverse Rolle 61

Kardinalitätsbeschränkungen 58, 59, 69
Katalog-Aggregation 80
Klasse 54, 56
Klassendiagramme 67
Klassenoperationen 67
Komponenten 79
Komponentengruppe 79
Komposition 80
Konstruktionssystematik 8

Lastenheft 13, 38
Link-Attribut 72
Links 69

Masken 105
MDA 24
Mehrfachvererbung 78
Menüs 105
Methoden 55, 68
MOF 27
Multiplizität 58
Murphys Gesetze 7

Oberklasse 74
Object Modeling Technique 65
Object Oriented Software Engineering 65
Objekt 54
Objektattribute 68
Objektdatentyp 103
objektorientierte Systemanalyse 54
Objektvariable 104
OMT 65
OOA 54
OOSE 65
Operation 68
overlapping 76

Pfad 66
Pflichtenheft 5, 36, 60
physikalische Aggregation 80
polymorph 68
Primary Key 61
Problemanalyse 11
Produktionssystematik 8
Programmtest 14
Projektion 69
Projektplanung 36
Prototyp 17
Prototyping 18, 105
Prozesse 81
Pseudoattribute 72

referentielle Integrität 102
reflexive Beziehung 63
Relation 48
Relationenklassen 58
Relationship 47
Relationship set 47
Rolle 61
Rollen 69
RTSA 81
Rumbaugh, Jim 65

Schema 55
semantische Datenmodellierung 60
semantisches Datenmodell 13, 36, 39, 54
Sequenzdiagramme 49
SOA 20
Software Engineering 8
Software-Analyse 36
Software-Architekturen 20

Software-Design 36
Softwareergonomie 105
Spezialisierung 74
spezifische Attribute 74
Spiralmodell 17
State-Transition-Diagram 85
Steuerungen. 81
String 66
Subklassen 74
Superklasse 74
Systemanalyse 8, 38
Systemanalytiker 6

Test 14
Transaktions-Analyse 82
Transformationen 81
Transformations-Analyse 82
transitive Instanz 74
Transitivitätseigenschaft 79

U-Case-Diagramme 43
überlappend 76
UML 27, 65
Unterklassen 74
unvollstänidige Generalisierung 76
User-Case-Diagramme 43

Vererbung 74, 77
Verknüpfungen 66
vollständige Generalisierung 76

Ward-Mellor-Methode 81
Wartung 15
Wasserfallmodell 10
Wertebereich 47

XMI 28

Zugriffsoperationen 55
Zustandsdiagramme 51, 74
Zustandübergangs-Diagramm 85