

# ***Software-Entwicklung***

---

## Grundlagen

**Autor**

Guido Rau

## 0. Inhaltsverzeichnis

0. Inhaltsverzeichnis.....	0-2
1. Einführung.....	1-5
2. Überblick.....	2-6
2.1. Lastenheft.....	2-7
2.2. Pflichtenheft.....	2-7
2.3. Analyse.....	2-7
2.4. Entwurf (Design).....	2-8
2.5. Implementierung.....	2-8
2.6. Testphase.....	2-8
2.7. Dokumentation.....	2-8
2.8. Abnahme- und Einführungsphase.....	2-9
2.9. Wartungs- und Pflegephase.....	2-9
3. Lastenheft.....	3-10
3.1. Begriffsdefinition.....	3-10
3.2. Beschreibung.....	3-10
3.3. Beispiel für ein Lastenheft.....	3-10
3.4. Übung.....	3-12
4. Die Analyse.....	4-13
4.1. Begriffsdefinition.....	4-13
4.2. Beschreibung.....	4-13
4.3. Die objektorientierte Analyse (OOA).....	4-13
4.4. Pflichtenheft.....	4-14
4.4.1. Begriffsdefinition.....	4-14
4.4.2. Beschreibung.....	4-14
4.4.3. Aufbau eines Pflichtenheftes.....	4-15
4.4.4. Beispiel.....	4-18
4.4.5. Übung.....	4-20
4.5. Das OOA-Modell.....	4-20
4.5.1. Das statische Modell.....	4-21
4.5.2. Das dynamische Modell.....	4-21
4.6. Der Prototyp.....	4-21
4.7. Erstellung des OOA-Modells.....	4-21
4.8. Konzepte und Notation des OOA-Modells.....	4-22

4.8.1. Das Basiskonzept.....	4-22
4.8.2. Das statische Konzept.....	4-31
4.8.3. Das dynamische Konzept.....	4-37
5. Gestaltung der Benutzungsoberfläche.....	5-52
5.1. Einführung in die Software Ergonomie.....	5-52
5.1.1. Das GUI-System.....	5-52
5.1.2. Das Gestaltungsregelwerk.....	5-52
5.2. Dialoggestaltung.....	5-52
5.2.1. modale und nicht-modale Dialoge.....	5-52
5.2.2. SDI und MDI-Anwendungen.....	5-53
5.2.3. Ergonomische Anforderungen für Bürotätigkeiten mit Bildschirmgeräten (Teil 10:Grundsätze der Dia- loggestaltung ISO 9241-10 : 1996).....	5-53
5.2.4. Alternativen der Dialoggestaltung.....	5-54
5.3. Fenster.....	5-54
5.3.1. Anwendungsfenster.....	5-54
5.3.2. Unterfenster.....	5-54
5.3.3. Dialogfenster.....	5-54
5.3.4. Mitteilungsfenster.....	5-55
5.4. Menüs.....	5-55
5.4.1. Aktionsmenü.....	5-55
5.4.2. Eigenschaftenmenü.....	5-55
5.4.3. Menübalken.....	5-55
5.4.4. drop-down-Menü.....	5-55
5.4.5. pop-up-Menü.....	5-55
5.4.6. Beschleunigte Menüauswahl.....	5-55
5.4.7. Symbolbalken.....	5-56
5.4.8. Gestaltungsregel für Menüs.....	5-56
5.5. Vom Klassendiagramm zur Dialogstruktur.....	5-57
5.5.1. Das Erfassungsfenster.....	5-57
5.5.2. Das Listenfenster.....	5-57
5.5.3. Assoziationen.....	5-58
5.5.4. 1-Assoziation.....	5-58
5.5.5. many-Assoziation.....	5-59
5.5.6. Vererbungsstrukturen in Dialogen.....	5-60

5.6.	Interaktionselemente.....	5-61
5.7.	Gestaltung von Fenstern.....	5-62
5.7.1.	Gestaltungsregeln für Fenster.....	5-63
5.7.2.	Gestaltungsregeln für das Hervorheben.....	5-63
5.7.3.	Gestaltungsregeln für die Farbwahl.....	5-63
5.7.4.	harmonische Gestaltung.....	5-64
6.	Der objektorientierte Entwurf (Design).....	6-67
6.1.	Entwurfsziel.....	6-67
6.2.	Das statische Modell des OOD.....	6-69
6.3.	Das dynamische Modell des OOD.....	6-69
6.4.	Konzept und Notation des objektorientierten Entwurfs.....	6-69
6.4.1.	Objekte / Klassen.....	6-70
6.4.2.	Attribute.....	6-74
6.4.3.	Operationen.....	6-76
6.4.4.	Assoziation.....	6-79
6.4.5.	Polymorphismus.....	6-85
6.4.6.	Vererbung.....	6-85
6.4.7.	Paket.....	6-85
6.4.8.	Szenario.....	6-85
6.4.9.	Zustandsautomat.....	6-85
7.	Implementierung.....	7-86
8.	Softwaretestverfahren.....	8-107
9.	Glossar.....	8-127
10.	Index	9-128

## **1. Einführung**

Im Rahmen dieses Trainings soll der Teilnehmer die einzelnen Entwicklungsphasen eines Software-Produktes und die in diesen Phasen angewendeten Techniken kennen lernen und einsetzen können.

Nach einem globalen Überblick über die einzelnen Phasen werden deren Einzelheiten vorgestellt. Anhand von Beispielen wird die Anwendung erläutert.

Im Anschluss an die Erläuterung jeder Phase setzt der Teilnehmer die neu erlernten Techniken im Rahmen eines Projektes um, das sich als roter Faden durch das gesamte Training zieht.

## 2. Überblick

Die Software-Entwicklung durchläuft mehrere Phasen. Die erste dieser Phasen ist die Planungsphase. Hier werden vor Beginn der Entwicklung eines Software-Produktes zunächst eine Voruntersuchung oder Durchführbarkeitsuntersuchung durchgeführt. Am Ende dieser Untersuchung steht die Entscheidung, ob ein Projekt fachlich, ökonomisch und personell umgesetzt werden kann und somit weitergeführt wird oder nicht.

Die Planungsphase besteht aus mehreren Einzelaktivitäten:

Auswählen des Produktes

### 1. Trendstudie

- Marktanalyse
- Forschungsergebnisse
- Kundenanfragen
- Vorentwicklungen (bei bereits vorhandenen Vorgängerversionen)

### 2. Voruntersuchungen zum Projekt

- u. U. Ist-Aufnahme wenn bereits ein Vorgängermodell vorhanden ist
- Festlegen der Hauptanforderungen
  - Festlegen der Hauptfunktionen
  - Festlegen der Hauptdaten
  - Festlegen der Hauptleistungen
  - Festlegen der wichtigsten Aspekte der Benutzerschnittstelle
  - Festlegen der wichtigsten Qualitätsmerkmale

### 3. Durchführbarkeitsuntersuchung

- Prüfen der fachlichen Durchführbarkeit (softwaretechnische Realisierbarkeit, Verfügbarkeit geeigneter Entwicklungs- und Zielmaschinen)
- Prüfen der alternativen Lösungsvorschläge (z. B. Kauf und Anpassung von Standardsoftware vs. Individualsoftware)
- Prüfen der personellen Durchführbarkeit (Verfügbarkeit geeigneter Fachkräfte für die Entwicklung)
- Prüfen der Risiken

### 4. Prüfen der wirtschaftlichen Durchführbarkeit

- Aufwands- und Terminschätzung
- Wirtschaftlichkeitsrechnung

Nach der Durchführung dieser Vorüberlegungen erfolgt die „Durchführbarkeitsstudie“, die sich in folgende Einzelschritte aufteilt.

5. Lastenheft
6. Projektkalkulation
7. Projektplan

Das Hauptanliegen der Planungsphase ist es also zu prüfen, ob ein Produkt entwickelt werden soll oder nicht. Gegenstand des Trainings sind die der Planungsphase nachgeordneten Phasen. Es wird also im weiteren Verlauf des Trainings davon ausgegangen, dass diese Vorüberlegungen bereits abgeschlossen sind.

### **Lastenheft**

*Das Lastenheft enthält eine Zusammenfassung aller fachlichen Basisanforderungen, die das zu entwickelnde Software-Produkt aus der Sicht des Auftraggebers erfüllen muss.*

Das Lastenheft stellt quasi eine grobes Pflichtenheft dar, in dem die wesentlichen Anforderungen an ein Software-Produkt beschrieben werden.

### **Pflichtenheft**

*Das Pflichtenheft ist eine Beschreibung dessen, was das zu realisierende Produkt leisten soll.*

Das Pflichtenheft bildet den ersten Schritt der Systemanalyse. Es beschreibt sehr viel detaillierter die Anforderungen an das System. Hierbei hat es zwei Aufgaben zu erfüllen. Zum ersten ist das Pflichtenheft das Einstiegsdokument in das Projekt, das als Grundlage zur Einarbeitung für spätere Wartungsarbeiten dient. Zum zweiten ist es die Grundlage zur Erstellung der objektorientierten Analyse. Das Pflichtenheft stellt das Produkt nicht so detailliert dar, wie das Ergebnis der OOA und ist daher nicht die Grundlage der Implementierung.

### **Analyse**

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

Bei der Analyse wird ein Modell des Fachkonzeptes unter Berücksichtigung der im Pflichtenheft festgehaltenen Anforderungen erstellt, das konsistent, eindeutig und realisierbar ist. Bei der Systemanalyse werden alle Aspekte der Implementierung unberücksichtigt gelassen.



## Entwurf (Design)

*Aufgabe des Entwurfs ist es, die in der OOA spezifizierte Anforderung an ein Software-Produkt unter Berücksichtigung der äußeren Rahmenbedingungen und der geforderten technischen Randbedingungen zu realisieren.*

Der Entwurf findet hierbei noch auf einem höheren Abstraktionsniveau statt als die Implementierung. Im Entwurf entwickelte Klassen zu einem Projekt können jedoch aufgrund der starken Verzahnung direkt implementiert werden.

Auch werden in der Entwurfsphase die Bibliotheken festgelegt, die für das Projekt herangezogen werden sollen.

## Implementierung

*Die Implementierung ist die Umsetzung der Ergebnisse der Entwurfsphase in ein Software-Produkt mittels einer Programmiersprache.*

In der Implementierungsphase wird der eigentliche Quellcode der Software einschließlich der integrierten Dokumentation des Quelltextes (Kommentare) erstellt. Darüber hinaus werden die Testmethoden geplant und vorbereitet.

## Testphase

*Die Testphase überprüft und dokumentiert die Ergebnisse der Implementierungsphase*

Während der Testphase werden die Einzelmodule eines Software-Produktes sowie deren Zusammenwirken auf Funktionstüchtigkeit, Ergonomie und Sicherheit überprüft. Sämtliche Testweisen und Testergebnisse werden in der Projektdokumentation festgehalten.

## Dokumentation

Die Dokumentation eines Software-Produktes ist die schriftliche Niederlegung aller zum Produkt gehörenden Anforderungen und Ergebnisse.

Zur Dokumentationen gehören nicht nur die Benutzer-Handbücher, sondern vielmehr gehört zur Dokumentation in der Hauptsache die schriftliche Niederlegung aller entscheidungswichtigen Ergebnisse der einzelnen Phasen wie:

- Ergebnisse der Trendstudie in der Planungsphase
- Ergebnisse der Voruntersuchungen in der Planungsphase
- Ergebnisse der Durchführbarkeitsuntersuchung in der Planungsphase

- Das Lastenheft
- Ergebnisse der Analyse
- Das Pflichtenheft
- Ergebnisse der Designphase
- Kommentierung der Implementierung
- Testbeschreibungen und Ergebnisse in der Testphase
- Die Benutzerhandbücher

## **Abnahme- und Einführungsphase**

*Die Abnahme- und Einführungsphase ist die Installation und Inbetriebnahme des fertiggestellten Produktes beim Anwender oder Auftraggeber.*

Diese Phase beinhaltet nicht nur die Installation und Inbetriebnahme des Produktes beim Anwender, sondern eventuell auch die Schulung der Mitarbeiter. In den Bereich der Inbetriebnahme fällt auch eine eventuelle Übernahme der Daten aus dem alten System.

## **Wartungs- und Pflegephase**

*Wartung und Pflege ist die Betreuung des Produktes nach der Einführung*

In diese Phase gehört die Beseitigung von Fehlern, die trotz umfangreicher Tests im täglichen Betrieb noch auftreten. Ebenso fällt hierunter die Betreuung bei Veränderungen in der Systemlandschaft und Reaktion auf veränderte Anforderungen bedingt durch die Geschäftsabläufe beim Kunden.

Die beiden Punkte 2.8 und 2.9 sollen im Rahmen dieses Scripts nur der Vollständigkeit halber genannt werden. Sie gehören nicht unmittelbar zum Inhalt dieses Trainings.

### 3. Lastenheft

#### Begriffsdefinition

*Das Lastenheft enthält eine Zusammenfassung aller fachlichen Basisanforderungen, die das zu entwickelnde Software-Produkt aus der Sicht des Auftraggebers erfüllen muss.*

#### Beschreibung

Die Formulierung „Basisanforderungen“ bedeutet, dass sich die Beschreibung im Lastenheft lediglich auf die fundamentalen Eigenschaften des Produktes bezieht. Sie befindet sich auf einem Abstraktionsniveau, das die Anforderungen ausreichend beschreibt, ohne dabei zu konkrete Anforderungen zu formulieren. Der Inhalt beschreibt das „WAS“ nicht das „WIE“.

Eine sinnvolle Gliederung könnte z. B wie folgt aussehen:

1. Zielbestimmung
  - Beschreibung, welche Ziele durch das Produkt erreicht werden sollen.
2. Produkteinsatz
  - Festlegung des Anwendungsbereiches sowie der Zielgruppe für das Produkt
3. Produktfunktionen
  - Hier werden die Hauptfunktionen des Produktes festgelegt. Hierbei handelt es sich nur um die Kernfunktionen und nicht um sekundäre Funktionen. Jede Funktion wird durch eine Zahl mit vorangesetztem LF (**L**astenheft **F**unktion) gekennzeichnet. Die Buchstaben-Zahlen-Kombination wird von Schrägstrichen eingeschlossen (/LFnn/). Die Zahlen folgen hierbei Zehner-Schritten.
4. Produktdaten
  - Hierunter werden die Hauptdaten festgelegt, die permanent gespeichert werden müssen (/LDnn/).
5. Produktleistung
  - Diese Angaben sind optional, und werden nur gemacht, wenn besondere Leistungsanforderungen bezüglich Datenumfang oder Genauigkeit an das Produkt gestellt werden (/LLnn/).
6. Qualitätsanforderungen
  - Die wichtigsten Anforderungen sollten genannt werden, wie z. B. Zuverlässigkeit, gute Ergonomie, und Effizienz
7. Ergänzungen

- Hier werden spezielle Anforderungen beschrieben, wie z. B. besondere Anforderungen an die Benutzeroberfläche oder Schnittstelle zu anderen Produkten

### Beispiel für ein Lastenheft

- 1 Zielbestimmung  
Für die Firma EDV-TRANS soll ein Verwaltungsprogramm für das Hochlager erstellt werden. Die Verwaltung dieses Lagers soll damit vereinfacht werden. Außerdem soll die Disposition und die Auftragsabteilung mit eingebunden werden.
- 2 Produkteinsatz  
Das Programm soll bei der Firma EDV-TRANS in den Bereichen Hochlager, Auftragsbearbeitung und Disposition eingesetzt werden. Zielgruppe sind die Mitarbeiter dieser Abteilungen.
- 3 Produktfunktionen  
 /LF10/ Erfassung, Änderung und Stornierung von Bestellungen  
 /LF20/ Erfassung, Änderungen, Löschungen des Lagerbestandes.  
 /LF30/ Drucken von Bestelllisten  
 /LF40/ Drucken von Lagerlisten  
 /LF50/ Abfrage von Lagerbeständen, Auftragsständen und Lieferterminen  
 /LF60/ Erstellung von Lieferscheinen
- 4 Produktdaten  
 /LD10/ Es müssen relevante Daten über Artikel gespeichert werden.  
 /LD20/ Es müssen relevante Daten über Kunden gespeichert werden.  
 /LD30/ Es müssen relevante Daten über Lagerorte gespeichert werden.
- 5 Produktleistung  
 /LL10/ Die Antwortzeit der Funktion /LF50/ darf maximal 15 Sekunden betragen.
- 6 Qualitätsanforderungen

Produkt-qualität	Sehr gut	Gut	Normal	Nicht relevant
Funktionalität		X		
Zuverlässigkeit			X	
Benutzbarkeit		X		
Effizienz		X		
Änderbarkeit		X		

---

Übertrag- barkeit
----------------------

X
---

## Übung

Charlies Never-Come-Back Airline benötigt ein neues Programm für die Gepäck- und Passagierabfertigung seiner Fluggäste.

**Erstellen Sie ein Lastenheft!**

## 4. Die Analyse

### Begriffsdefinition

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

### Beschreibung

Ziel der Analyse ist es, aus den Wünschen und Anforderungen des Auftraggebers ein Fachkonzept zu erstellen. Hierbei werden alle Aspekte der Implementierung ausgeklammert. Außerdem wird bei der Analyse von einer „perfekten“ Technik ausgegangen. Einflussfaktoren wie sie in der Realität vorkommen bleiben unberücksichtigt. D. h. es wird ein System unterstellt, in dem keine Fehler auftreten, der Speicher unendlich groß ist und keine Verzögerungen oder anderweitige Beeinflussungen stattfinden. Es sollen bei der Analyse die „wahren“ Anforderungen des Auftraggebers modelliert werden. Das Finden dieser „wahren“ Anforderungen ist ein kontinuierlicher Prozess, bei dem sich das Sammeln, Filtern und Dokumentieren von Informationen ständig wiederholt, bis ein endgültiges Modell zur Verfügung steht.

### Die objektorientierte Analyse (OOA)

Bei der objektorientierten Analyse (OOA) wird von Objekten der realen Welt ausgegangen. Die können Personen, Artikel oder aber auch Aufträge oder Lieferscheine sein. Durch Modellbildung entsteht aus diesen Objekten eine geeignete Abstraktion in ein objektorientiertes Modell. Dabei gehören Objekte mit gleichen Eigenschaften gleichen Klassen an. Das Ziel der OOA ist, das Problem zu verstehen und nicht, das WIE der Darstellung, Speicherung oder Selektierung darzustellen.

Zur Präsentation des OOA-Modells beim Auftraggeber empfiehlt sich der Einsatz eines Prototypen der Benutzungsoberfläche. Anhand dieser kann der Auftraggeber dann seine Änderungswünsche darlegen.

Im Rahmen der Produktanalyse sind folgende Produkte zu erstellen, auf die im folgenden näher eingegangen wird:

- Pflichtenheft
- OOA-Modell

Statisches Modell

Dynamisches Modell

- Prototyp

## Pflichtenheft

### 4..1. Begriffsdefinition

*Das Pflichtenheft ist eine detaillierte Beschreibung dessen, was das zu realisierende Produkt leisten soll.*

### 4..2. Beschreibung

Nach der groben Festlegung der Anforderungen an das Software-Produkt im Lastenheft wird nun im Pflichtenheft die detaillierte Darstellung der Anforderungen festgehalten. Das Pflichtenheft stellt bei Aufträgen zur Erstellung von Individual-Software die Grundlagen des Vertrages dar. Hierzu muss es einige Anforderungen erfüllen.

#### Aufgaben des Pflichtenheftes

Das Pflichtenheft enthält eine Zusammenfassung aller fachlichen Anforderungen, die das zu entwickelnde Software-Produkt aus der Sicht des Auftraggebers erfüllen muss.

#### Adressaten des Pflichtenheftes

Zu den Adressaten des Pflichtenheftes gehören die Vertreter des externen oder internen Auftraggebers, und das Entwicklerteam des Auftragnehmers. Seitens des Auftraggebers zählen hierzu die Vertreter der Fachabteilung und Benutzerrepräsentanten oder ausgewählte Benutzer. Auf der Seite des Auftragnehmers sind dies der Projektleiter sowie Systemanalytiker, Designer und Vertreter der Qualitätskontrolle.

#### Inhalt des Pflichtenheftes

Der Inhalt des Pflichtenheftes beschreibt den fachlichen Funktions-, Daten-, Leistungs- und Qualitätsumfang. Dabei wird nicht beschrieben, WIE die Anforderungen umzusetzen sind, sondern WAS umzusetzen ist. Da das Pflichtenheft als Grundlage eines Vertrages dient, stellt es die vertragliche Vereinbarung des **Lieferumfangs** dar. Demzufolge ist es gleichzeitig auch die Basis für die spätere Abnahme des Produktes durch den Auftraggeber. Ohne ein Pflichtenheft ist eine korrekte Abnahme nicht möglich. Sollte sich während des Projektablaufs die Notwendigkeit zur Änderung des Pflichtenheftes ergeben, so sind diese Änderungen vom Auftraggeber zu bestätigen. Hierzu erhält das Pflichtenheft eine Versionsnummer.

Bei der Erstellung des Pflichtenheftes ist darauf zu achten, dass die beschriebenen Anforderungen realisierbar sind ohne dabei Entscheidungen vorwegzunehmen, die den Entwurf oder die Implementierung einschränken könnten.

#### Sprache des Pflichtenheftes

Das Pflichtenheft enthält die Anforderungen in detaillierter verbaler Form, in der die einzelnen Anforderungspunkte durch Nummerierung von einander getrennt sind. Diese Nummerierung ist erforderlich, um



in späteren Dokumentationen auf die entsprechenden Punkte des Pflichtenheftes verweisen zu können.

### **Didaktik des Pflichtenheftes**

Da das Pflichtenheft u. a. zur Einarbeitung in das Projekt dienen soll, empfiehlt sich ein Gliederungsschema, das das Pflichtenheft gut lesbar macht.

### **Erstellungszeitpunkt des Pflichtenheftes**

Nach Abschluss der Planungsphase ist das Pflichtenheft das erste Dokument, das erstellt wird.

## **4..3. Aufbau eines Pflichtenheftes**

Der Aufbau eines Pflichtenheftes unterteilt sich in zehn Kapitel.

### **1. Zielbestimmung**

Dieses Kapitel beschreibt, welche Ziele durch das Projekt erreicht werden sollen. Hierbei wird unterschieden in Kriterien, die erfüllt sein müssen und Kriterien, die wünschenswert wären.

#### **1.1. Musskriterien**

Hier werden die Kriterien aufgeführt, die unabdingbarer Bestandteil des Produktes sein sollen, um die Verwendbarkeit zu gewährleisten.

#### **1.2. Wunschkriterien**

Hierzu gehören Kriterien, die zwar nicht unabdingbar für das Projekt sind, deren Vorhandensein jedoch angestrebt werden soll.

#### **1.3. Abgrenzungskriterien**

Um eine klare Definition des Produktes zu erhalten, werden an dieser Stelle die Kriterien aufgeführt, die nicht Bestandteil des Produktes sein sollen.

### **2. Produkteinsatz**

Da der Einsatz des Produktes Auswirkungen auf den Umfang der Funktionalität und die Qualitätsmerkmale hat, werden in diesem Unterpunkt die nachfolgenden Abschnitte festgehalten.

#### **2.1. Anwendungsbereiche**

z. B. Textverarbeitung, Buchführung oder CAD-Anwendung

## 2.2. Zielgruppe

z. B. Sekretärinnen, Schreibkräfte, Buchhalter, technische Zeichner. Es kann sinnvoll sein, hier auch die Voraussetzung beim Benutzer für den Gebrauch der Software festzuhalten (z. B. DV-unkundig).

## 2.3. Betriebsbedingungen

Physikalische Umgebung des Systems, tägliche Betriebszeit und ob das System durch den Bediener ständig beobachtet wird oder ein unbeaufsichtigter Betrieb erforderlich ist.

## 3. Produktumgebung

### 3.1. Software

Beschreibung der Software-Systeme auf der Zielmaschine (Betriebssysteme, Datenbanksystem u. s. w.)

### 3.2. Hardware

Beschreibung der Hardwareumgebung der Zielmaschine (CPU, Peripheriegeräte wie z. B. Drucker, Bildschirm)

### 3.3. Orgware

Beschreibung der organisatorischen Randbedingungen unter denen das Produkt eingesetzt werden soll. Ein E-Mail-System lässt sich nur dann sinnvoll nutzen, wenn der Empfänger in das elektronische Postsystem eingegliedert ist. Es ist also eine entsprechende LAN-Verbindung erforderlich.

### 3.4. Produkt-Schnittstellen

Soll das Produkt in eine bestehende Produktfamilie eingegliedert werden oder mit anderen Software-Produkten zusammenarbeiten, müssen entsprechende Schnittstellen beschrieben werden (z. B. für Ferndiagnose-Systeme, Interaktion mit bestehenden Produkten).

## 4. Produktfunktionen

Hier erfolgt die funktionale Beschreibung der Produkt-Funktionen aus der Sicht des Benutzers. Für jede Funktion des Programms sollte ein eigener Abschnitt eingerichtet werden. Jede Einzelanforderung einer Funktion wird – wie im Lastenheft – durch ein F gefolgt von einer Zahl gekennzeichnet. Die Kombination wird zwischen Schrägstriche gesetzt (z. B. /F10/). Die Nummerierung erfolgt in Zehnerschritten um Ergänzungen leichter einfügen zu können. Handelt es sich bei der Funktion um ein Wunschkriterium, wird an die Nummerierung ein „W“

angefügt (/F20W/). Die Festlegung der Funktionen erfolgt unabhängig vom Bildschirmlayout und der Tastaturbelegung. Diese werden erst im Kapitel 7 des Pflichtenheftes berücksichtigt.

## 5. Produktdaten

In diesem Kapitel werden alle langfristig zu speichernden Daten aus Benutzersicht beschrieben. Die Nummerierung erfolgt analog zu den Funktionen: /D10/ u. s. w.

## 6. Produktleistung

Hier werden alle leistungsbezogenen Anforderungen des Produktes beschrieben. Hierzu gehören z. B. Antwortzeiten spezieller Funktionen (Datenbankabfragen) oder die Genauigkeit numerischer Werte. Auch hier erfolgt die Nummerierung wieder analog zu den beiden vorhergehenden Kapiteln mit z. B. (/L10/).

## 7. Benutzungsoberfläche

Dieses Kapitel beschreibt die grundlegenden Anforderungen an die Benutzungsoberfläche, wobei je nach Produkt folgende Gesichtspunkte zu berücksichtigen sind:

- Bildschirmlayout
- Drucklayout
- Tastaturbelegung
- Dialogstruktur und -aufbau

Im Rahmen dieses Kapitels sollten nur die produktspezifischen Bedingungen festgelegt werden, die nicht durch die Qualitätszielbestimmungen in Kapitel 8 behandelt werden.

## 8. Qualitätsbestimmungen

Hier werden die Qualitätsanforderungen an das Produkt aufgeführt. Hierzu zählen auch die Aspekte der Ergonomie der Benutzungsoberfläche, soweit sie nicht produktspezifisch sind und somit bereits unter Punkt 7 behandelt wurden.

## 9. Testszenarien / Testfälle

Kapitel 9 beinhaltet eine Zusammenstellung aller anwendungsbezogenen Testfälle, die mehrere Funktionen des Produktes betreffen. Während die Testfälle für einzelne Funktionen des Programms innerhalb der Funktionsanforderungen behandelt werden, sollten hier die globalen Testfälle aufgeführt werden, die

mehrere Funktionen des Produktes betreffen. Dieser Test bildet dann die Grundlage für den Abnahmetest.

## 10. Entwicklungsumgebung

In diesem Kapitel ist die Entwicklungsumgebung des Produktes zu beschreiben. Es muss festgelegt werden, welche Konfiguration bezüglich Soft- und Hardware inklusive eventueller Peripheriegeräte und Orgware (z. B. LAN-Umgebung) benötigt wird. Seitens der Software wird der verwendete Compiler sowie andere eventuell benötigte Entwicklungswerkzeuge aufgeführt.

## 11. Ergänzungen

Hierunter fallen alle nicht unter den Punkten 1 – 10 behandelten Anforderungen. Beispielsweise sind hier Installationsbedingungen baulicher oder räumlicher Natur zu nennen oder die Bereitstellung von Testdaten und Hilfspersonal durch den Auftraggeber. Weiterhin können Normen, Vorschriften, Lizenzen und Patente, die Bestandteil des Projektes sein sollen, hier benannt werden. Auch kann es sinnvoll sein im Pflichtenheft verwendete Fachbegriff an dieser Stelle zu definieren um Missverständnisse auszuschließen.

### 4..4. Beispiel

- 1 Zielbestimmungen

Für die Firma EDV-TRANS soll ein Verwaltungsprogramm für das Hochlager erstellt werden. Die Verwaltung dieses Lagers soll damit vereinfacht werden. Außerdem soll die Disposition und die Auftragsabteilung mit eingebunden werden.
- 1.1 Musskriterien
  - Verwaltung des Lagers (Ein- und Auslagern von Artikel)
  - Verwaltung von Aufträgen / Erstellung von Lieferscheinen
  - Verwalten von Kunden
  - Abfragen:
    - Welcher Artikel ist mit welchem Bestand am Lager?
    - Wann werden bestellte Artikel wieder verfügbar sein?
- 1.2 Wunschkriterien
  - Erweiterte Abfragemöglichkeiten
  - Weiterleitung der Lieferscheindaten an die Buchhaltung zur Rechnungslegung
  - Warenumschlagstatistik
- 1.3 Abgrenzungskriterien

- Keine integrierte Buchhaltung
- 2 Produkteinsatz
  - Das Programm soll bei der Firma EDV-TRANS in den Bereichen Hochlager, Auftragsbearbeitung und Disposition eingesetzt werden.
- 2.1 Anwendungsbereich
  - Lagerverwaltung und Disposition
- 2.2 Zielgruppe
  - Zielgruppe sind die Mitarbeiter dieser Abteilungen.
- 2.3 Betriebsbedingungen
  - Büro (Disposition) und Hochlager
- 3 Produktumgebung
  - Das Produkt läuft auf Arbeitsplatzrechnern, die Daten werden in einer Datenbank auf dem Server hinterlegt.
- 3.1 Software
  - Betriebssystem: Windows NT 4.0 SP 6a
- 3.2 Hardware
  - PC
- 3.3 Orgware
  - Netzwerkverbindung zur Buchhaltung und Disposition.

### 3.4 Produktschnittstellen

Lieferscheindaten werden in die Datenbank eingetragen, auf die auch die Buchhaltung zugreift. Neue Lieferscheinnummern werden automatisch an die Datenbank weitergegeben.

## 4 Produktfunktionen

### 4.1 Lagerverwaltung

/F10/ Erfassung, Änderung und Stornierung von Bestellungen

/F20/ Erfassung, Änderungen, Löschungen des Lagerbestandes.

/F30/ Drucken von Bestelllisten

/F40/ Drucken von Lagerlisten

/F50/ Abfrage von Lagerbeständen, Auftragsständen und Lieferterminen

/F60/ Erstellung von Lieferscheinen

### 4.2

## 5 Produktdaten

/D10/ Es müssen relevante Daten über Artikel gespeichert werden.

/D20/ Es müssen relevante Daten über Kunden gespeichert werden.

/D30/ Es müssen relevante Daten über Lagerorte gespeichert werden.

## 5 Produktleistung

/L10/ Die Antwortzeit der Funktion /F50/ darf maximal 15 Sekunden betragen.

## 6 Qualitätsanforderungen

Produkt- qualität	Sehr gut	Gut	Normal	Nicht relevant
Funktio- nalität		X		
Zuver- lässigkeit			X	
Benutzbar- keit		X		
Effizienz		X		

---

Änder- barkeit	X	
Übertrag- barkeit		X

#### 4..5. Übung

Entwickeln Sie aus dem im vorangegangenen Kapitel erstellten Lastenheft von Charlies Never-Come-Back Airline ein Pflichtenheft!

#### Das OOA-Modell

Das OOA-Modell stellt die fachliche Lösung des zu realisierenden Systems dar. Man spricht von einem Fachkonzept. Die OOA teilt sich in zwei Bereiche. Zum einen in die statische Analyse, zum anderen in die dynamische Analyse. Welcher dieser beiden Bereiche für die jeweilige Anwendung der wichtigere ist, hängt im wesentlichen von der besonderen Art des Produktes ab. Der statische Bereich der OOA ist bei Anwendungen vorrangig, die schwerpunktmäßig typische Datenbank-Anwendungen sind. Der dynamische Bereich der OOA ist für im wesentlichen interaktiv orientierte Anwendungen wie z. B. Echtzeitanwendungen zur Maschinensteuerung von stärkerer Bedeutung. Die nachfolgende Abbildung zeigt, wie die objektorientierten Konzepte auf diese Modelle abgebildet werden können.

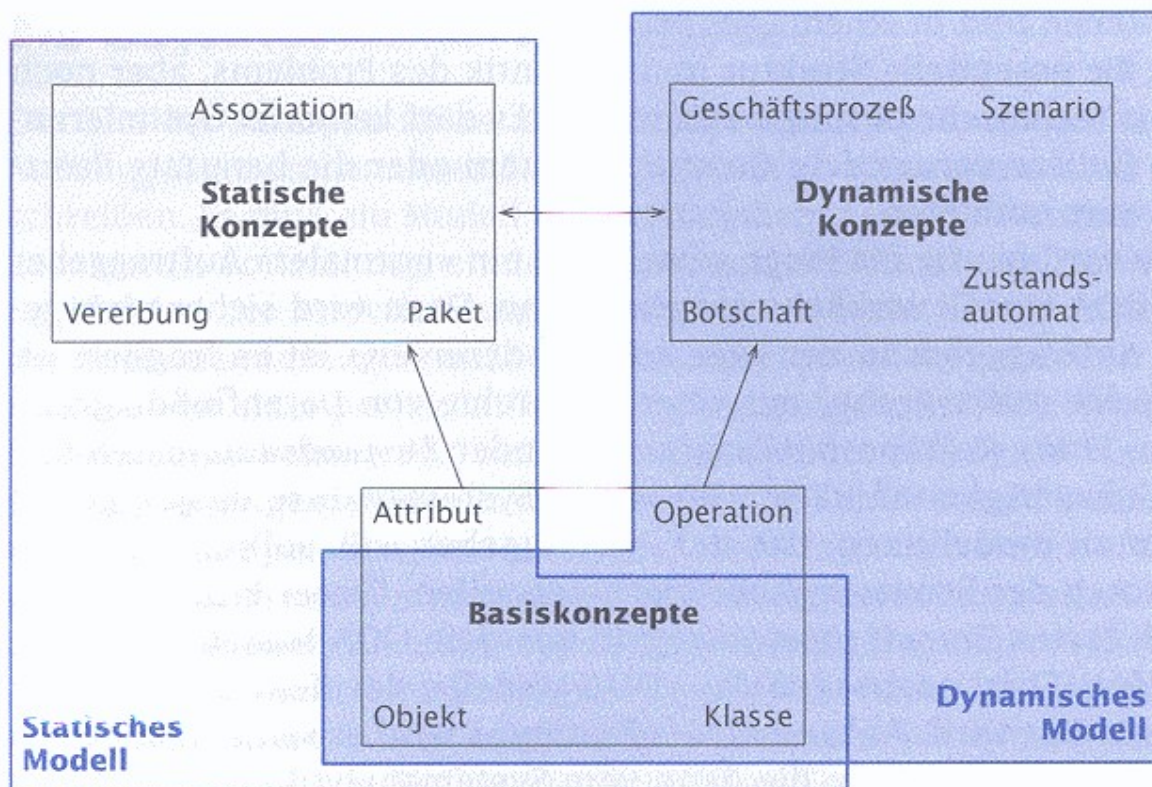


Abbildung 4.1 Untergliederung des objektorientierten Modells



#### **4..6. Das statische Modell**

Das statische Modell beschreibt die Klassen des Systems, die Assoziationen (Beziehung) der Klassen untereinander und die Vererbungsstrukturen. Außerdem sind im statischen Modell die Daten des Systems enthalten (Attribute). Der besseren Übersicht wegen können auch Teilsysteme innerhalb eines Systems gebildet werden. Diese Teilsysteme werden als Pakete bezeichnet.

#### **4..7. Das dynamische Modell**

Im Gegensatz zu dem statischen Modell befasst sich das dynamische Modell mit den Funktionsabläufen innerhalb eines Systems. Die Geschäftsprozesse zeigen die durchzuführenden Aufgaben auf einem sehr hohen Abstraktionsniveau. Innerhalb dieses Modells zeigen Szenarios die Kommunikation zwischen den einzelnen Objekten, die zur Erledigung der verschiedenen Aufgaben notwendig ist. Die Lebenszyklen der einzelnen Objekte, d. h. die Reaktion der Objekte auf verschiedene Ereignisse, wird durch Zustandsautomaten beschrieben.

Das Modell des Fachkonzeptes enthält alle Informationen, die zur Erstellung des Prototypen der Benutzungsoberfläche erforderlich sind. Hieraus kann also der Prototyp der Benutzungsoberfläche abgeleitet werden. Dieser Prototyp sollte darüber hinaus eine entsprechend ergonomische Gestaltung aufweisen. Er dient zur Evaluierung des Produktes durch den Auftraggeber.

#### **Der Prototyp**

Der Prototyp einer Benutzungsoberfläche zu einem Projekt stellt ein ablauffähiges Programm dar, das alle Attribute des Projektes auf der Oberfläche abbildet. Er enthält jedoch weder irgendwelche Anwendungsfunktionen, noch ist eine Datenspeicherung möglich. Der Prototyp besteht aus den für das Projekt erforderlichen Fenstern, Menüs und Dialogen. Er dient dazu dem späteren Benutzer das zu entwickelnde Programm zur Evaluierung vorzustellen. Die Benutzungsoberfläche sollte vollständig sein, Lassen sich nicht alle Informationen innerhalb der Oberfläche darstellen (z. B. Konzepte der Zugriffsberechtigungen), sollte der Prototyp durch eine entsprechende Dokumentation ergänzt werden. Da sich im Lebenszyklus eines Software-Programms im allgemeinen die Benutzungsoberfläche schneller ändert als die Funktionalität des Fachkonzeptes, ist die Trennung dieser beiden Bereiche in der Entwicklung ein Grundprinzip. Hierbei legt das Fachkonzept fest, welche Informationen auf dem Bildschirm dargestellt werden, die Benutzungsoberfläche dagegen, wie bzw. in welchem Format die Informationen auf dem Bildschirm dargestellt werden sollen.

#### **Erstellung des OOA-Modells**

Das OOA-Modell wird durch ein Team erstellt, das sich aus circa zwei bis fünf Personen zusammensetzt. Hieran sind neben dem Systemanalytiker auch Fachexperten und die zukünftigen Benutzer oder deren Repräsentanten beteiligt. Z. B. würde bei der Erstellung eines

Buchhaltungsprogramms der Buchhalter als Fachexperte und die Angestellte des Steuerberaters als zukünftige Benutzerin zum Team gehören. Bei größeren Projekten sollte sich auch ein Experte für Software-Ergonomie im Entwicklungsteam befinden

Die Erstellung eines OOA-Modells ist jedoch keineswegs ein Prozess, in dem jeder Schritt einmal durchlaufen wird. Vielmehr entsteht das OOA-Modell durch Iterationen. So wird beispielsweise zunächst ein OOA-Modell erarbeitet. Hieraus wird dann ein passender Prototyp entworfen, der dem Benutzer zur Evaluierung vorgelegt wird. Aus dieser Besprechung ergeben sich wiederum notwendige Änderungen, die zu einem modifizierten Prototyp führen, der dann seinerseits wieder dem Benutzer zur Auswertung vorgelegt wird. Dieser Vorgang wird solange wiederholt, bis eine optimale Lösung für die Umsetzung des Projektes gefunden wird. Der Prototyp muss immer aus dem OOA-Modell abgeleitet werden. Der umgekehrte Weg führt insbesondere bei nicht-trivialen Problemstellungen zu schwer verständlichen Dialogabläufen.

### **Konzepte und Notation des OOA-Modells**

Nachfolgend werden die Begriffe innerhalb der einzelnen Konzepte sowie deren Darstellung innerhalb der Modelle behandelt. Hierbei erfolgt die Darstellung in der Notation der UML (**U**nified **M**odeling **L**anguage)

## **4..8. Das Basiskonzept**

### **Objekt**

#### **4..8..1 Begriffsdefinition**

*Ein Objekt ist ein Gegenstand des Interesses einer Beobachtung, Untersuchung oder Messung. Es kann sich hierbei um Dinge (z. B. Lagerartikel, Bestelllisten), Personen oder Begriffe (z. B. Programmiersprache, ?) handeln.*

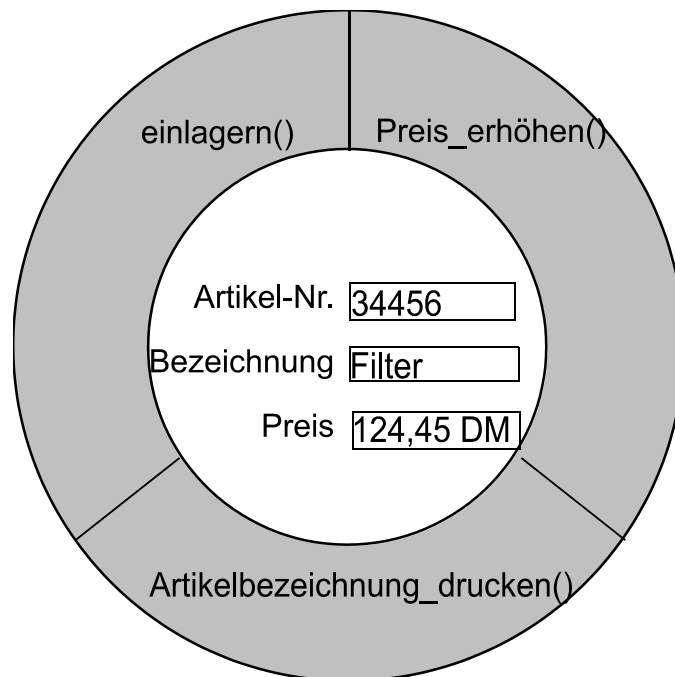
#### **4..8..2 Beschreibung**

In der objektorientierten Programmierung besitzt ein Objekt einen bestimmten Zustand und reagiert auf seine Umgebung mit einem bestimmten vordefinierten Verhalten. Jedes Objekt besitzt darüber hinaus eine Identität, die eine eindeutige Unterscheidung zu anderen Objekten zulässt. Ein Objekt kann mehrere andere Objekte kennen. In dem Fall sprechen wir von Verbindungen zwischen den Objekten.

Der Zustand eines Objektes wird durch die aktuellen Werte seiner Attribute sowie durch seine Verbindungen zu anderen Objekten beschrieben. Die Attribute sind unveränderliche Merkmale eines Objektes. Deren Werte hingegen können sehr wohl Änderungen unterliegen.

Das Verhalten eines Objektes wird durch seine Operationen bestimmt. Eine Änderung oder Abfrage eines Attributwertes ist nur über Operationen möglich.

Ein Artikel besitzt eine Artikelnummer, eine Bezeichnung und einen Preis (Attribute). Er kann in ein Lager eingelagert werden, der Preis kann erhöht oder verringert werden und die Daten über den Artikel können ausgedruckt werden (Operationen). Die Attribute werden hierbei von der Außenwelt verborgen und können nur durch die Operationen des Objektes verändert werden.



**Abbildung 4.2 Artikel-Objekt**

#### 4..8..3 Notation

Ein Objekt wird in der UML-Notation durch ein horizontal geteiltes Rechteck dargestellt. Im oberen Teil befindet sich der Objektname und die Klasse, zu der das Objekt gehört. Beide Angaben werden durch einen Doppelpunkt getrennt (Abbildung 4.3)



**Abbildung 4.3 Notation von Objekten nach UML**

Für die Beschriftung des oberen Feldes gibt es drei Möglichkeiten

1. :Klasse bei anonymen Objekten wird nur der Klassenname angegeben
2. Objekt:Klasse wenn das Objekt über einen Klassennamen angesprochen werden soll

3. Objekt                    wenn der Objektname zu eindeutigen Identifizierung  
ausreicht

Die Bezeichnung eines Objektes wird immer unterstrichen. Anonyme Objekte werden verwendet, wenn irgendein Objekt dieser Klasse angesprochen werden soll. Objektnamen beziehen sich immer auf ein bestimmtes Objekt einer Klasse. Sie dienen zur Benennung des Objektes für den Systemanalytiker.

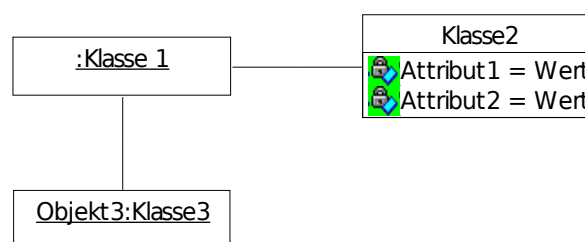
Im unteren Feld werden optional die Attribute des Objektes eingetragen, die im jeweiligen Kontext relevant sind. Hier sind seitens der UML folgende Alternativen vorgesehen:

1. Attribut:Typ = Wert
2. Attribut = Wert diese Weise empfiehlt sich, da der Typ des Attributes bereits bei der Klasse definiert wird.
3. Attribut                    sinnvoll, wenn der Wert eines Attributes nicht relevant ist.

Die Operationen, die ein Objekt ausführen kann, werden nicht mit angegeben. Diese Angabe ist ebenfalls bereits bei der Klassendefinition enthalten.

#### 4..8..4 Objektdiagramm

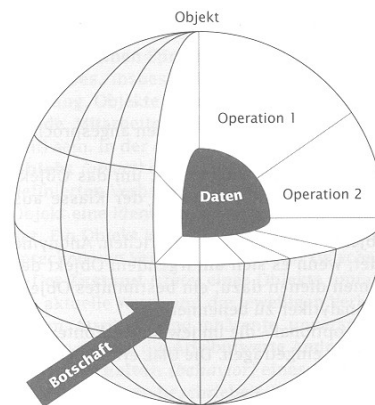
Das Objektdiagramm zeigt die Verbindungen der Objekte untereinander auf (Abbildung 4.4). Die Beschreibung der Objekte, Attribute und Verbindungen beziehen sich dabei auf einen bestimmten Zeitpunkt. Man könnte Objektdiagramme auch als Momentaufnahmen des Systems ansehen.



**Abbildung 4.4 Notation des Objektdiagramms**

#### 4..8..5 Geheimnisprinzip

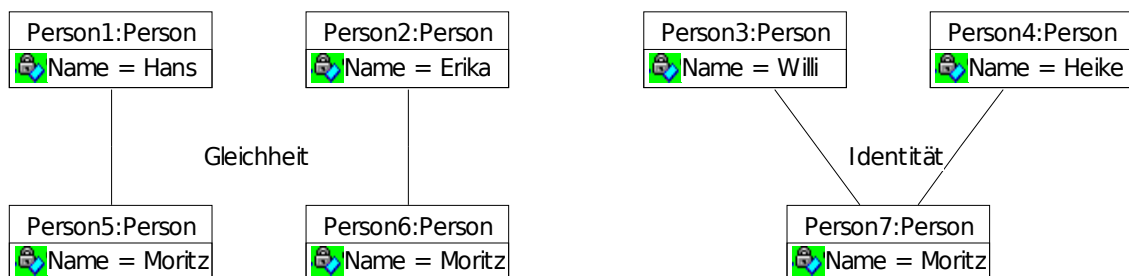
Wesentlicher Inhalt der Objektorientierung ist die Bildung einer Einheit von Zustand und Verhalten eines Objektes. In der objektorientierten Programmierung spricht man hierbei von der Kapselung der Daten. D. h. Zustände eines Objektes können nur durch Operationen des Objektes gelesen oder geschrieben werden. Die Daten eines Objektes bleiben also nach außen verborgen. Das Objekt erfüllt das Geheimnisprinzip.



**Abbildung 4.5 Realisierung des Geheimnisprinzips eines Objektes**

#### 4..8..6 Objektidentität

Die Objektidentität ist die Eigenschaft, die ein Objekt eindeutig von allen anderen Objekten unterscheidet. Auch Objekte, die zufällig identische Attributwerte haben, können somit aufgrund ihrer Identität unterschieden werden. Die Identität eines Objektes kann nicht geändert werden. Eine Gleichheit zwischen Objekten entsteht, wenn ihre Attributwerte dieselben sind. Haben z. B. die Personen Hans und Erika beide ein Kind mit dem Namen Moritz (Gleichheit), so besteht Gleichheit zwischen den beiden Personen „Moritz“. Heike und Willi dagegen haben ein gemeinsames Kind mit dem Namen Moritz. Hier besteht Identität (Abbildung 4.6).



**Abbildung 4.6 Gleichheit und Identität von Objekten**

#### 4..8..7 Externe und interne Objekte

Bei der Objektorientierung muss zwischen internen und externen Objekten unterschieden werden. Bei den externen Objekten handelt es sich um in der realen Welt existierende Objekte, die internen Objekte hingegen sind nur für die Systementwicklung interessant. Der Kunde Meier einer Bank ist ein begeisterter Segler. Für ein Softwaresystem, das Überweisungsaufträge bearbeiten soll, ist diese Eigenschaft jedoch vollkommen unwichtig. Das interne Objekt „Meier“ würde diese Eigenschaft also nicht besitzen. Sollte das Softwareprodukt jedoch bei einem Sportausstatter zur Versendung von Werbung eingesetzt werden, wäre diese Eigenschaft auch für das interne Objekt wichtig. Interne Objekte reduzieren die Eigenschaften der externen Objekte auf die für das System entscheidenden Merkmale.

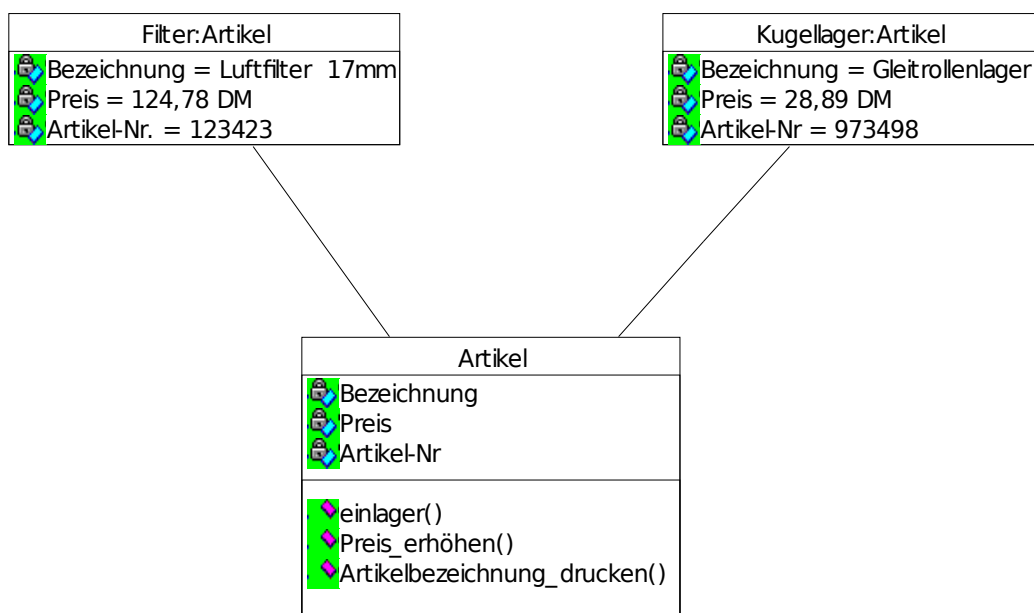
## Klasse

### 4..8..8 Begriffsdefinition

*Eine Klasse definiert Attribute, Operationen und Beziehung, die für mehrere gleichartige Objekte Gültigkeit haben. Klassen verfügen über je einen Mechanismus um ein Objekt dieser Klasse zu erzeugen und wieder zu zerstören.*

### 4..8..9 Beschreibung

Jedes Objekt gehört zu genau einer Klasse. Unter den Beziehungen einer Klasse versteht man ihre Assoziationen und Vererbungsstrukturen. Hierauf wird im Script weiter unten noch näher eingegangen. Eine Klasse bzw. deren Objekte reagiert auf Botschaften vom Programm. Die Fähigkeit, hierauf zu reagieren, bestimmt das Verhalten der Klasse. Durch jede Botschaft des Programms wird eine Operation aktiviert, die den gleichen Namen trägt wie die Botschaft. In der Abbildung 4.7 gehören sowohl der Filter als auch das Kugellager zur Klasse Artikel.



**Abbildung 4.7 Mehrere Objekte der Klasse Artikel**

### 4..8..10 Notation

Die vollständige Notation der Klassen enthält den Klassennamen im oberen Feld. Der Name sollte immer ein Substantiv sein und wird groß geschrieben. Im mittleren Feld stehen die Attribute und im unteren Feld werden die Operationen eingetragen. Je nach Notwendigkeit kann eines der beiden unteren Felder oder auch beide in der Darstellung entfallen. Der Klassenname im oberen Feld wird zentriert und fettgedruckt angegeben.

Die Klassensymbole werden zusammen mit weiteren Symbolen in das Klassendiagramm eingetragen. In großen Systemen kann es notwendig sein, mehrere Klassendiagramme zu erstellen.

#### 4..8..11 Abstrakte Klasse

Wie bereits in der Definition für Klassen ausgeführt, besitzt jede Klasse einen Mechanismus zur Erzeugung von Objekten eben dieser Klasse. Bei der Vererbung ist es aber zuweilen sinnvoll mit Klassen zu arbeiten, von denen keine Objekte herstellbar sind. Diese Klassen werden als abstrakte Klassen bezeichnet. Sie werden entweder durch einen kursiv geschriebenen Klassennamen oder das Merkmal *{abstract}* kenntlich gemacht.

#### 4..8..12 Kurzbeschreibung der Klasse

Da jede Klasse einen bestimmten Zweck im System erfüllt, sollte ihr eine kurze Beschreibung hinzugefügt werden, aus der ihr Zweck im System erkennbar ist.

Klasse Artikel      Artikel der im Hochregallager abgelegt ist

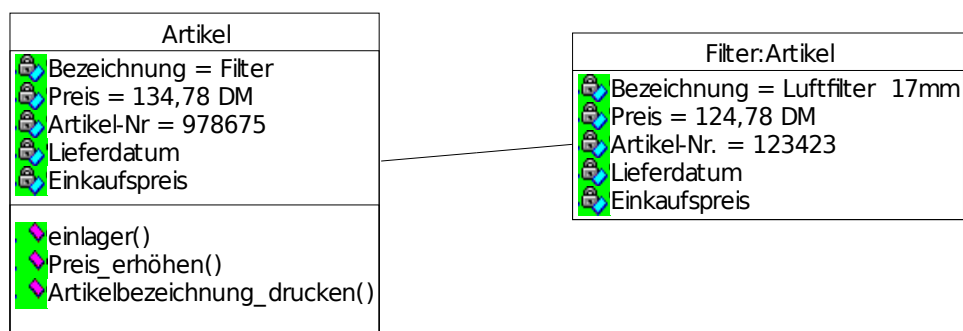
### Attribut

#### 4..8..13 Begriffsdefinition

*Die Attribute einer Klasse beschreiben die Daten der Objekte sowie deren Typ. Alle Objekte einer Klasse haben gleiche Attribute. Die Attributwerte hingegen können sowohl gleich als auch unterschiedlich sein.*

#### 4..8..14 Beschreibung

Wie in Abbildung 4.8 zu sehen ist beschreibt die Klasse welche Attribute ein Objekt besitzt, das Objekt dagegen legt fest, welche Werte diese Attribute haben. Der Attributwert darf auch leer sein. Das Attribut erhält also nicht bei der Erzeugung sondern erst zu einem späteren Zeitpunkt einen definierten Wert (eventuell auch nie).



**Abbildung 4.8 Klasse Artikel und Artikel „Filter“**

#### 4..8..15 Restriktionen

In machen Fällen ist es notwendig, dass Beziehungen zwischen den Werten eines Objektes zur Laufzeit des Programms erhalten bleiben

müssen. Man spricht in diesem Fall von einer Restriktion. Eine Restriktion muss immer wahr sein, sie ist invariant.

Z. B. muss der Verkaufspreis des Artikels „Filter“ immer das 1,5-fache des Einkaufspreises betragen.

Bei der Implementierung muss in diesem Fall sicher gestellt werden, dass diese Restriktion eingehalten wird.

#### 4..8..16 Klassenattribute

Ein Attribut eines Objektes hat normaler Weise für jedes Objekt einen anderen Attributwert. Attribute sind also Objekt bezogen. Will man z. B. eine Attribut haben, mit dem man festhalten kann, wie viel Objekte einer Klasse bereits erzeugt wurden, lässt sich dies nicht mit einem Attribut umsetzen, das für jedes Objekt einen eigenen Attributwert hat. Hier kann man sogenannte Klassenattribute einsetzen. Dies sind Attribute, bei denen nur ein Attributwert für alle Objekte einer Klasse existiert.

Angenommen Sie wollen wissen wie viel verschiedene Artikel es im Hochlager gibt. Für jeden Artikel wird ein neues Objekt der Klasse Artikel erzeugt. Zählen Sie nun bei jedem Erzeugen des Artikel-Objektes ein entsprechendes Klassenattribut hoch und beim Löschen eines Objektes herunter, dann enthält das Klassenattribut immer die Anzahl der verschiedenen Artikel im Lager.

#### 4..8..17 Abgeleitete Attribute

Ein weiterer Typ sind sogenannte abgeleitete Attribute. Hierunter versteht man Attribute eines Objektes, die sich aus anderen Objektattributen errechnen lassen. Diese Attribute werden in der Notation durch einen vorangestellten Schrägstrich gekennzeichnet (Abbildung 4.9). Ein abgeleitetes Attribut darf nicht geändert werden.

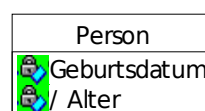


Abbildung 4.9 Abgeleitetes Attribut

#### 4..8..18 Attributtypen

Jedes Attribut einer Klasse / Objektes ist von einem bestimmten Datentyp. Hierfür kommen die verschiedenen numerischen sowie alpha-numerischen Datentypen in Frage. In der Notation wird der Typ hinter dem Attributnamen angegeben getrennt durch einen Doppelpunkt (Attributname:Typ).

#### 4..8..19 Elementare Klasse

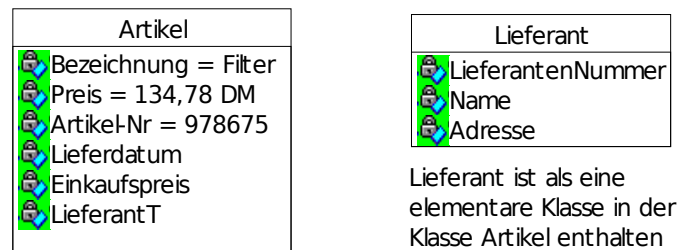
Der Typ eines Attributes kann aber auch wieder eine Klasse sein. Bei solchen als Typ verwendeten Klassen spricht man von elementaren Klassen. Um nicht in Konflikt mit den Attributnamen einer Klasse zu geraten, empfiehlt es sich, den Namen einer elementaren Klasse besonders kenntlich zu machen. Dies kann z. B. durch ein angehängtes



T erfolgen (Abbildung 4.10). Eine Vorgabe gibt es seitens der UML hierfür nicht.

#### 4..8..20 Komplexe Objekte

Besitzt ein Objekt Attribute, die selber wieder Objekte sind, werden diese Objekte als komplexe Objekte bezeichnet (Abbildung 4.10). Dieses (Unter-) Objekt kann wiederum ein komplexes Objekt sein. Ist das Unterobjekt aus der gleichen Klasse gebildet wie das Objekt selber spricht man von einem rekursiven Objekt.



**Abbildung 4.10 Elementare Klassen als Typ in einer anderen Klasse**

#### 4..8..21 Attributspezifikation

Der Prototyp der Benutzungsoberfläche soll aus dem Ergebnis der OOA abgeleitet werden. Es ist daher notwendig, die einzelnen Attribute der Klasse zu spezifizieren. Hierfür werden folgende Angaben zu den Attributen gemacht.

Name

1. Typ
2. Anfangswert
3. Muss-Attribut (mandatory)
4. Schlüssel (key)
5. Attributwert nicht änderbar (frozen)
6. Einheit
7. Beschreibung

Die ersten drei Spezifikationen wurden bereits erläutert. Ein Muss-Attribut (4) ist ein Attribut, das beim Erzeugen des Objektes einen Anfangswert erhalten muss. Schlüssel ist ein Attribut dann, wenn sich das Objekt über dieses Attribut eindeutig innerhalb einer Klasse identifizieren lässt (5). Es können auch mehrere Attribute einer Klasse Schlüsselattribute sein. In diesem Fall handelt es sich um sogenannte zusammengesetzte Schlüssel. Attribute, deren Wert nach einer einmaligen Definition nicht mehr geändert werden kann, werden mit „nicht änderbar“ oder {frozen} spezifiziert. Die Spezifikation „Einheit“ wird zur Gestaltung der Benutzungsoberfläche verwendet. Die Bedeutung eines Attributes kann noch - falls notwendig - durch eine Beschreibung ergänzt werden. Bei abgeleiteten Attributen enthält die Beschreibung die Ableitungsregel.

In der Notation wird folgende Schreibweise angewendet:

Attribut : Typ = Anfangswert

{mandatory, key, frozen, Einheit: ..., Beschreibung}

Auf das Beispiel des Hochregallagers angewandt, könnte dies z. B. so aussehen:

```
Bezeichnung : String(60) {mandatory}
Preis :      Float      {mandatory}
Artikel-Nr. : Long      {mandatory, key, frozen}
Lieferdatum : Date {Beschreibung : Termin der nächsten Lieferung}
Einkaufspreis : Float {Beschreibung : Preis bei letzter Lieferung}
Lieferant : LieferantT
            {Beschreibung : günstigster Lieferant für diesen Artikel}
```

## Operation

### 4..8..22 Begriffsdefinition

*Eine Operation ist eine ausführbare Tätigkeit. Die Operationen einer Klasse werden von allen Objekten dieser Klasse verwendet. Die Menge aller Operationen einer Klasse wird als Verhalten bzw. Schnittstelle bezeichnet.*

### 4..8..23 Beschreibung

Bezogen auf die Klasse Artikel kann also jedes Objekt auf die Operationen einlagern(), Preis\_erhöhen() und Artikelbezeichnung\_drucken() zugreifen.

Artikel
Bezeichnung = Filter Preis = 134,78 DM Artikel-Nr = 978675 Lieferdatum Einkaufspreis LieferantT
einlagern() Preis_erhöhen() Artikelbezeichnung_drucken()

**Abbildung 4.11 Attribute und Operationen der Klasse Artikel**

### 4..8..24 Notation

Analog zu den Attributen einer Klasse werden die Operationen einer Klasse in der Notation eingefügt (Abbildung 4.11).

Man kann drei Arten von Operationen unterscheiden

1. Objektoperationen
2. Konstruktoroperationen

### 3. Klassenoperationen

### 4. Verwaltung

#### **4..8..25     Objektoperationen**

Objektoperationen, auch kurz Operationen genannt, werden immer auf ein bestimmtes Objekt angewendet. Z. B.  
Artikelbeschreibung\_drucken() druckt immer die Beschreibung des Artikels, auf den sich das Objekt bezieht.

#### **4..8..26     Konstruktoroperationen**

Konstruktoroperationen sind Operationen, die innerhalb der Klasse beschrieben werden und die ein Objekt dieser Klasse neu erzeugen.

#### **4..8..27     Klassenoperationen**

Klassenoperationen sind Operationen, die sich nicht wie die Objektoperationen auf genau eine Objekt beziehen, sondern sich mehrere Objekte dieser Klasse oder sogar auf alle beziehen, aber in der Klasse implementiert sind. Dies könnte z. B. eine Operation Drucke\_alle\_Lagerartikel() sein, die nicht nur Informationen zu einem Lagerartikel ausdruckt, sondern Informationen zu allen Artikeln bzw. Objekten der Klasse Artikel. Diese Klassenoperationen sind analog zu den weiter oben besprochenen Klassenattributen zu sehen. Eine Klassenoperation Artikel\_anlegen() könnte z. B. bei jeder Neuanlage eines Artikels das Klassenattribut „Anzahl\_Lagerartikel“ um eins hochzählen.

#### **4..8..28     Verwaltungsoperationen**

Verwaltungsoperationen sind Operationen, die grundlegende Operationen innerhalb einer Klasse ausführen. Hierzu gehören im Wesentlichen die Operationen, die zum Erzeugen und Löschen eines Objektes benötigt werden (new(), delete()) oder die zum Lesen oder Setzen von Attributwerten erforderlich sind (setAttribut(), getAttribut()).

#### **4..8..29     Externe und interne Operationen**

Man unterscheidet bei den Operationen zwischen externen und internen Operationen. Extern ist eine Operation, wenn sie direkt von der Benutzungsoberfläche aus aktiviert wird. Dabei kann durch die externe Operation der Aufruf weiterer interner Operationen erfolgen. Interne Operationen werden im Gegensatz hierzu nur aus dem System selber aufgerufen. Für die Analyse ist es nun wichtig, die externen Operationen ausfindig zu machen und diese in das Klassendiagramm aufzunehmen. Eine Aufnahme der internen Operationen in das Klassendiagramm erfolgt nur, wenn es für das Verständnis notwendig ist. Die Namen der Operationen sollten so gewählt werden, dass sich hieraus die Aufgabe der Operation erkennen lässt (z. B. drucke\_Artikelbezeichnung()).

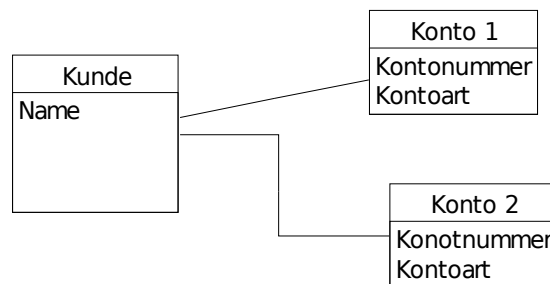
## 4..9. Das statische Konzept

Nach der Behandlung des Basiskonzeptes für die Notation, das in allen Bereichen benötigt wird, werden im Folgenden die für das statische Konzept der OOA benötigten Begriffe der Notation erläutert.

### Assoziation

Eine Assoziation modelliert eine Verbindung zwischen Objekten einer oder mehrerer Klassen, wobei die Modellierung stets zwischen den Objekten der Klassen und nicht zwischen den Klassen selber erfolgt. Zwar ist zuweilen von der Assoziation zwischen Klassen die Rede, dies sollte jedoch nicht darüber hinweg täuschen, dass tatsächlich eine Assoziation zwischen den Objekten dieser Klasse gemeint ist. Eine Assoziation, die zwischen Objekten der gleichen Klasse besteht, nennt man eine reflexive Assoziation.

Ein Kunde einer Bank eröffnet zwei Konten mit den Nummern 3445234 und 1234556. Jedes Konto lautet auf den Namen des Kunden. Es existiert also eine Verbindung zwischen dem Kunden und den beiden Konten ( Abbildung 4 .12).



**Abbildung 4.12 Assoziation zwischen Objekten**

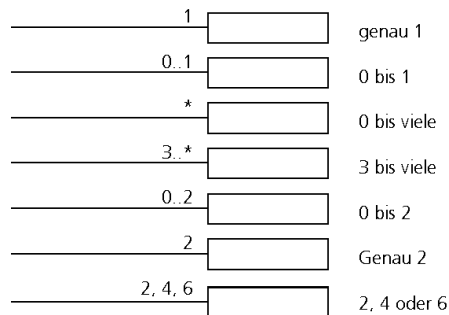
Ein Kunde ist mehreren Konten zugeordnete. Die Menge aller Verbindungen zwischen den Objekten einer Klasse wird als Assoziation bezeichnet. In der Systemanalyse sind diese Assoziationen bidirektional, d. h. jeder Kunde kennt seine Konten und jedes Konto kennt „seinen“ Kunden. Die UML kennt binäre und höherwertige Assoziationen. Im Rahmen dieses Scripts werden wir jedoch nur die binären – also die Assoziationen zwischen zwei Objekten - betrachten. Sie wird in der UML durch eine Linie zwischen einer oder zwei Klassen beschrieben. An den Enden wird die jeweilige Kardinalität angegeben (siehe unten).

### 4..9..1 Kardinalität

Im obigen Beispiel kann man sehen, dass sich ein Objekt (der Kunde) auf mehrere andere Objekte (die Konten) beziehen kann. Umgekehrt bezieht sich aber jedes Konto auf genau einen Kunden. Dieser Sachverhalt wird durch die Kardinalität der Assoziation beschrieben. Hierbei sagt die Assoziationslinie zunächst nur aus, dass sich die Objekte der beteiligten Klassen kennen. Die Kardinalität hingegen gibt nun an, wie viele Objekte ein bestimmtes anderes Objekt kennen.

#### 4..9..2 Notation der Kardinalität

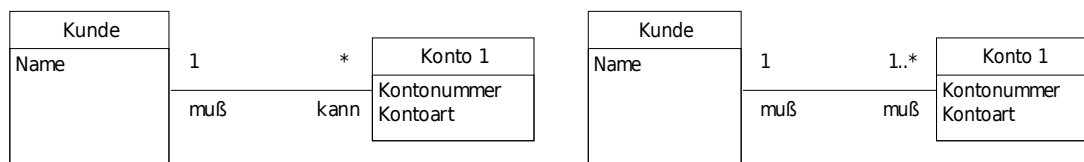
Die Beschreibung der Kardinalität erfolgt nun durch Beschriftung neben den Klassen wie in Abbildung 4.13. gezeigt. Wobei in dieser Abbildung die Klassen durch einfache Rechtecke dargestellt sind. Die nachfolgende Abbildung zeigt einige mögliche Kardinalitäten.



**Abbildung 4.13 Notation der Kardinalität**

#### 4..9..3 Kann- und Muss-Assoziation

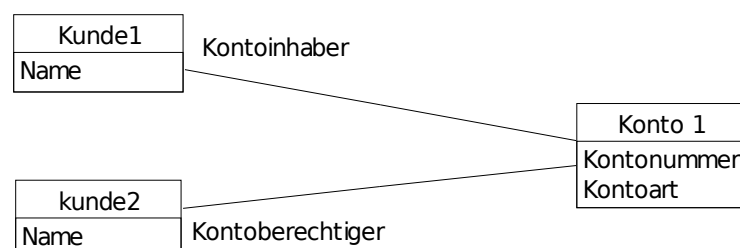
Neben der Kardinalität unterscheidet man Kann- und Muss-Assoziation. Im Beispiel kann die Bank einen Kunden haben, der aber kein Konto haben muss. Ein Konto muss jedoch genau einem Kunden zugeordnet werden (Abbildung 4.14 links). Andererseits kann es auch sein, dass einem Kunden mindestens ein Konto zugeordnet sein muss (Abbildung 4.14 rechts).



**Abbildung 4.14 Kann- und Muss - Assoziation**

#### 4..9..4 Rolle oder Rollename

Sind einer Klasse mehrere Objekte einer anderen Klasse zugeordnet, sollte zum besseren Verständnis in das Klassendiagramm eingetragen werden, in welchem Zusammenhang das jeweilige Objekt steht. Hierzu verwendet man den Rollennamen oder kurz die Rolle. In Abbildung 4.15 sind Kontoinhaber und Kontoberechtigter die „Rollen“, die das jeweilige Objekt dieser Klasse im Zusammenhang des Klassendiagramms spielt.



**Abbildung 4.15 Rollennamen**

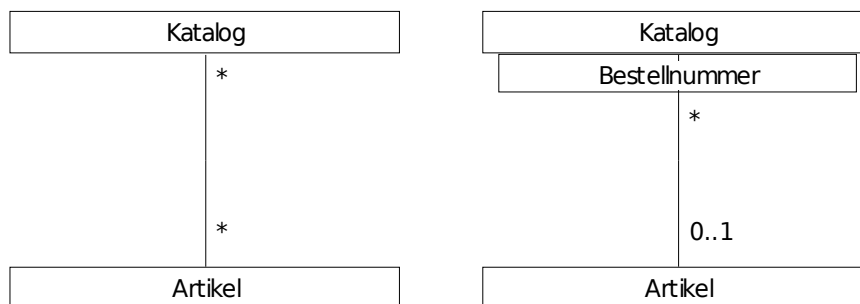
#### 4..9..5 Restriktionen

Es können für Assoziationen auch Restriktionen formuliert werden. Diese ergänzen die Assoziationen und können frei formuliert werden. Sollen im obigen Beispiel Kontoinhaber und Kontoberechtigter nicht identisch sein, könnte eine entsprechende Restriktion wie folgt formuliert werden:

{Kunde.Kontoinhaber <> Kunde.Kontoberechtigter}

#### 4..9..6 Qualifizierte Assoziation

Eine qualifizierte Assoziation liegt vor, wenn eine normale Assoziation durch eine Qualifikationsangabe die Menge der Objekte auf der anderen Seite der Assoziation in eine Teilmenge zerlegt. Hierdurch wird eine Anzahl Objekte auf der anderen Seite der Assoziation ausgewählt. In Verbindung mit einer Qualifikationsangabe besitzt die Angabe der Kardinalität eine entsprechend abgewandelte Bedeutung. 0..1 bedeutet es gibt kein Objekt auf der anderen Seite oder mehrere, 1 bedeutet es gibt genau ein Objekt oder \* bedeutet es gibt auf der anderen Seite der Assoziation eine Teilmenge von Objekten, die durch die Qualifikationsangabe selektiert werden. In Abbildung 4.16 ist dies am Beispiel eines Versandhauskataloges aufgezeigt. Im linken Teil besteht eine Assoziation zwischen dem Katalog und den Artikeln im Katalog. Auf der rechten Seite wird diese Assoziation spezifiziert durch die Bestellnummer, die entweder zu genau einem Artikel gehört (1) oder zu keinem Artikel des Kataloges (0).

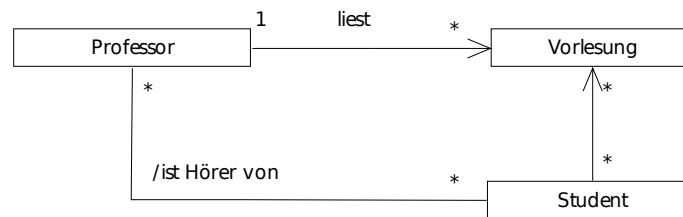


**Abbildung 4.16 Unqualifizierte und Qualifizierte Assoziation**

#### 4..9..7 Abgeleitete Assoziation

Sind Abhängigkeiten zwischen Objekten bereits beschreiben, so kann zur Erläuterung eine weitere Abhängigkeit dargestellt werden, die das gleiche aussagt, jedoch einen anderen Weg beschreibt, um diese Abhängigkeit aufzuzeigen. Solche Assoziationen nennt man abgeleitete Assoziationen. In der UML wird eine solche abgeleitete Assoziation durch einen vorangestellten Schrägstrich gekennzeichnet (Abbildung 4.17 „/ist Hörer von“). Die gleiche Assoziation (Ein Student hört die Vorlesung eines Professors) ist auch über die Assoziationen zwischen Professor und Vorlesung in Verbindung mit der Assoziation zwischen Student und Vorlesung gegeben. Die Assoziati-

on „ist Hörer von“ ist somit redundant und damit eine abgeleitete Assoziation.



**Abbildung 4.17 abgeleitete Assoziation**

#### 4..9..8 Aggregation

Existiert zwischen zwei Objekten der beteiligten Klassen eine Rangordnung derart, dass das eine Objekt Teil des anderen ist, dann sprechen wir von einer Aggregation. Für die Aggregation gilt: Wenn Objekt A Teil von Objekt B ist, kann Objekt B nicht Teil von Objekt A sein. Bei einer Aggregation kann Objekt A aber gleichzeitig noch Teil von einem Objekt C sein. Im Fall von Objekt B und C spricht man von Aggregatobjekten.

#### 4..9..9 Komposition

Für eine Komposition gelten im Grunde die gleichen Regel wie auch für die Aggregation. Im Unterschied bzw. darüber hinaus gilt aber auch:

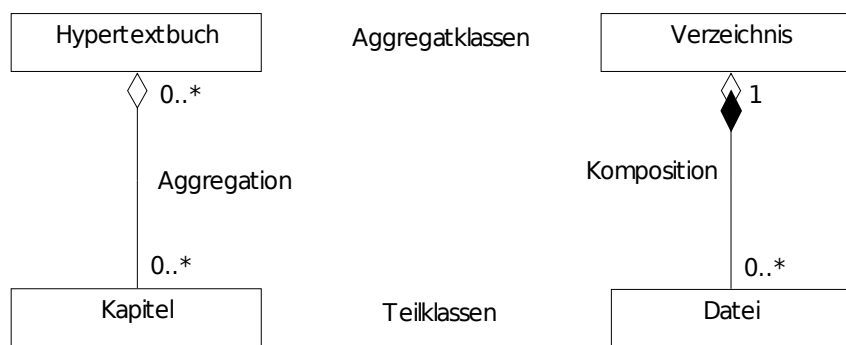
- Jedes Objekt der Teilklasse kann nicht gleichzeitig mehreren Aggregatobjekten angehören.

Wird das Aggregatobjekt kopiert oder gelöscht, so wird auch das Teilobjekt kopiert bzw. gelöscht.

Die Abgrenzung zwischen Aggregation und Komposition ist in der Praxis nicht immer ganz einfach. Auch findet man zu dieser Abgrenzung in der Literatur unterschiedliche Denkansätze.

Um den Unterschied zu erläutern betrachten wir folgendes Beispiel. Bei einem Online-Dokument kann zum Beispiel ein Kapitel über entsprechende Verweise auch zu einem anderen Dokument gehören. Eine Datei kann zum gleichen Zeitpunkt aber immer nur in einem Verzeichnis enthalten sein. In anderen Verzeichnisse können sich allenfalls Kopien dieser Datei befinden.

In der Notation wird eine Aggregation durch eine Verbindungslinie dargestellt, bei der das Ende an dem Aggregatobjekt mit einer transparenten Raute versehen ist. Bei einer Komposition ist die Raute ausgefüllt.



**Abbildung 4.18 Aggregation und Komposition**

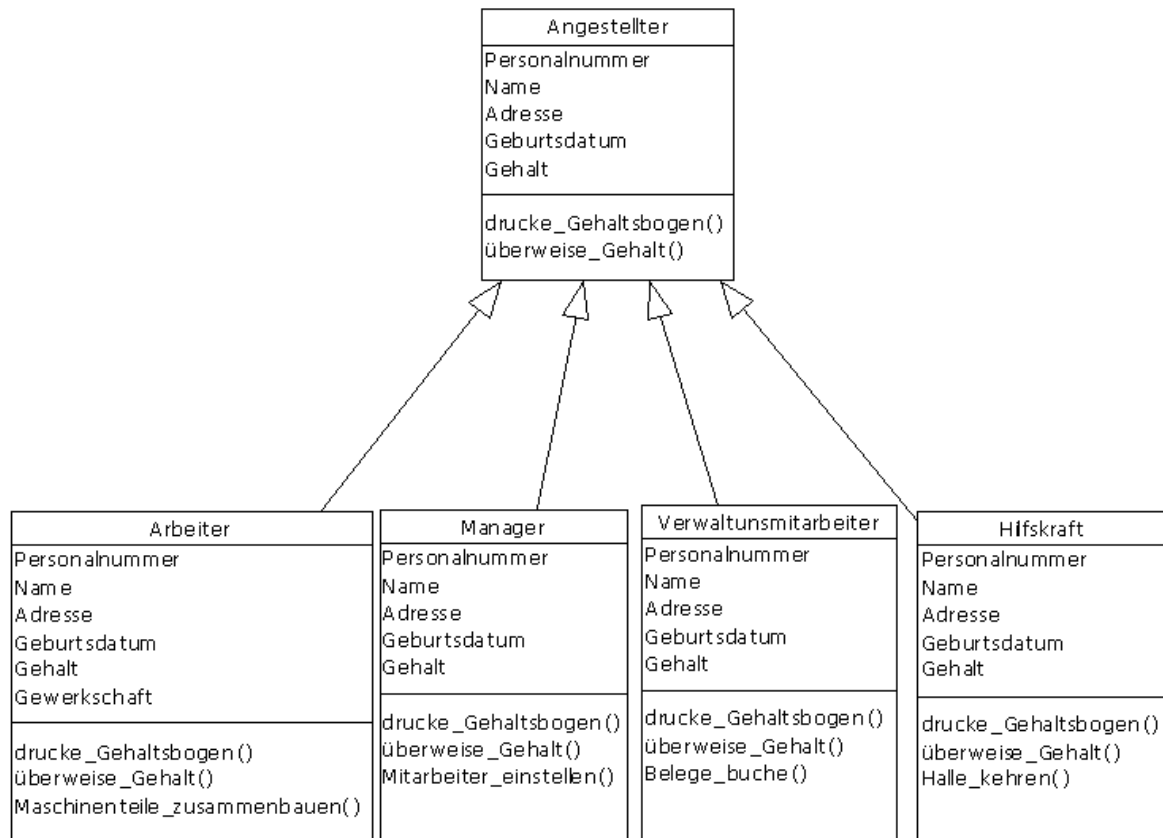
## Vererbung

### 4..9..10 Beschreibung

Ausgehend von einer Basisklasse kann eine spezialisierte Klasse abgeleitet werden. Hierbei ist die spezialisierte Klasse konsistent mit der Basisklasse, enthält aber zusätzliche Informationen in Form von weiteren Attributen, Operationen oder Assoziationen. Bei diesem Vorgang sprechen wir von Vererbung. Die spezialisierte Klasse kann überall eingesetzt werden wo auch die Basisklasse eingesetzt werden kann. Man spricht von der Oberklasse (super class) und der Unterklasse (sub class). Bei der Vererbung erfolgt also eine Spezialisierung der generellen Basisklasse.

Betrachtet man zum Beispiel eine Klasse Angestellter, so könnten die Klassen Arbeiter, Verwaltungsmitarbeiter, Manager und Hilfskraft entsprechende Vererbungen dieser Klasse sein.





**Abbildung 4.19 Beispiel einer Vererbungsstruktur**

#### 4..9..11 abstrakte Klasse

Eine besondere Form von Basisklassen sind Klassen, von denen keine Objekte gebildet werden können, aber von denen andere Klassen abgeleitet werden. Diese Klassen nennt man abstrakte Klassen. In der Notation werden sie gekennzeichnet indem der Klassenname kursiv gedruckt wird oder durch den Zusatz {abstract}.

#### 4..9..12 Notation

Im Diagramm wird die Vererbung mittels einer Verbindungslinie zwischen Basisklasse und spezialisierter Klasse zum Ausdruck gebracht. Die Verbindungslinie hat in diesem Fall am Berührungspunkt mit der Basisklasse ein transparentes Dreieck, das zur Basisklasse hin zeigt.

#### 4..9..13 Was wird vererbt?

Von der Oberklasse zur Unterklasse werden alle Attribute, Operationen, Klassenattribute und Assoziationen weitergegeben. Die Attributwerte hingegen werden nicht von der Oberklasse zur Basisklasse vererbt. Eine Ausnahme bilden hier die Werte der Klassenattribute. Diese werden mit den Attributen an die Unterklasse weitergereicht.

#### 4..9..14 Einfach- und Mehrfachvererbung

Neben der oben vorgestellten Einfachvererbung, bei der eine Unterklasse eine Oberklasse besitzt, gibt es noch eine sogenannte Mehrfachvererbung. Bei dieser besitzt eine Unterklasse zwei oder mehr Basisklassen. Die Vererbungsform wird jedoch in der OOA im allge-

meinen nicht benötigt und wird deshalb erst bei dem OOD besprochen werden.

#### **4..9..15 Vor- und Nachteile der Vererbung**

Aus der Vererbung ergeben sich wesentliche Vorteile. Es können mit sehr geringem Aufwand neue Klassen erzeugt werden, die auf den bereits existierenden Klassen aufbauen. Hieraus folgt, dass Änderungen in der Basisklasse automatisch Auswirkungen auf alle abgeleiteten Klassen haben, soweit innerhalb der Basisklassen die Attribute oder Operationen nicht überschrieben werden. Ein Nachteil ist, dass dies sich immer so verhält, auch wenn es nicht gewünscht ist. Außerdem wird durch die Vererbung das Geheimnisprinzip der Objektorientierung durchbrochen, nachdem keine Klasse die Attribute einer anderen Klasse sehen sollte. Diese Vererbung führt dazu, dass bei einer Änderung der Oberklasse möglicherweise auch die Unterklasse geändert werden muss.

#### **Paket**

Pakete sind zusammengefasste Modellelemente (z. B. Klassen). Ein Paket kann selber wieder Pakete beinhalten. Das Konzept des Paketes lässt sich sinnvoll zur Gruppierung der einzelnen Elemente des gesamten Softwaresystems nutzen.

In der Notation wird ein Paket als ein Rechteck mit einem Reiter in der linken oberen Ecke dargestellt. In der Notation kann der Inhalt des Paketes im Rechteck angezeigt werden, muss es aber nicht. Wird der Inhalt angezeigt, steht der Name des Pakets in dem Reiter, ansonsten wird er in das Rechteck geschrieben.

Abhängigkeiten zwischen Paketen werden in der Notation durch gestrichelte Linien mit einem Pfeil am Ende gezeigt. Eine Abhängigkeit zwischen Paketen bedeutet, dass bei Änderungen in dem einen Paket möglicherweise auch Änderungen innerhalb des anderen Paketes notwendig sind. Pakete werden in der UML im Klassendiagramm eingetragen. Diagramme, die nur Pakete enthalten, nennt man Paketdiagramme.

#### **4..10. Das dynamische Konzept**

Auch das dynamische Konzept hat seine speziellen Begriffe und eine eigene Notation in der UML.

## Geschäftsprozess

### 4..10..1 Begriffsdefinition

*Ein Geschäftsprozess (use case, workflow, business process) ist eine aus mehreren zusammenhängenden Aufgaben bestehende Einheit.*

### 4..10..2 Use case im Informationssysteme

Ein use case ist eine Sequenz zusammengehörender Transaktionen, die von einem Akteur im Dialog mit dem System ausgeführt werden. Dabei ist eine Transaktion eine Menge von Bearbeitungsschritten, von denen entweder alle oder keiner ausgeführt wird. Die Summe aller use cases eines Informationssystems dokumentiert die Möglichkeiten der Benutzung des Systems.

### 4..10..3 Use case im Unternehmen

Bezogen auf ein Unternehmen ist ein use case eine Anzahl unternehmensinterner Aktivitäten, die durchgeführt werden, um die Wünsche eines Kunden zu befriedigen.

### 4..10..4 Akteur

Ein Akteur ist eine Rolle, die ein Benutzer im System spielt, wobei er einen gewissen Einfluss auf das System hat. Es kann sich bei einem Akteur entweder um eine Person handeln, die das System benutzt, es kann aber auch eine Organisationseinheit des Unternehmens bzw. ein externes System wie z. B. ein anderes Unternehmen sein. Akteure befinden sich aber stets außerhalb des Systems.

Stellt das System ein Handelshaus dar, dann sind Kunde und Lieferant Akteure. Die Buchhaltung ist dagegen kein Akteur des Handelshauses, denn sie befindet sich innerhalb dieses Systems. Bei dem Softwaresysteme, das Auftrags- und Bestellwesen unterstützt, ist dagegen die Buchhaltung ein Akteur, denn diese Abteilung ist außerhalb dieses Systems, muss jedoch mit dem Softwaresystem kommunizieren. Die Aufgaben des Kunden und Lieferantensachbearbeiters können durchaus von der gleichen Person ausgeübt werden, bezogen auf das System handelt es sich aber dennoch um zwei Akteure, die zwei unterschiedliche Rollen spielen. Bei der Modellierung von Softwaresystemen sind also die Akteure diejenigen, die die Ergebnisse dieses Systems erhalten.

### 4..10..5 Spezifikation

Geschäftsprozesse können semiformal oder informal also umgangssprachlich beschrieben werden. Ein Grundprinzip der Modellierung ist, die Funktionalität des Systems von der Benutzungsoberfläche zu trennen. Die Geschäftsprozesse müssen demnach ebenfalls ohne Bezüge zur Benutzungsoberfläche beschrieben werden.

Der von vielen Faktoren abhängige Geschäftsprozess der Auftragsbearbeitung in einem Versandhaus wird unter dem Geschäftsprozess

Auftrag ausführen zusammengefasst. Gleichgültig ob es sich um Neukunden handelt und ob alle Artikel direkt geliefert werden können.

Geschäftsprozess Auftrag ausführen:

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

Beteiligte an diesem Geschäftsprozess sind: Kundensachbearbeiter, Lagersachbearbeiter und Buchhaltung.

#### 4..10..6 Notation

Während sich einfache Geschäftsprozesse leicht in umgangssprachlichen Formulierungen abfassen lassen, kann bei umfangreichen Geschäftsprozessen folgende Schablone als Checkliste eingesetzt werden:

Geschäftsprozess: Name z. B. Auftrag ausführen (Was wird getan?)

Ziel: globale Zielsetzung bei erfolgreicher Ausführung

Kategorie: primär, sekundär, optional

Vorbedingung: Zustand bei Beginn des Geschäftsprozesses

Nachbedingung: Erfolg: Zustand bei erfolgreicher Beendigung des Prozesses

Nachbedingung: Misserfolg: Zustand wenn Ziel nicht erreicht wurde

Akteure: Rolle der beteiligten Personen oder anderer Systeme, die den Geschäftsprozess auslösen oder daran beteiligt sind

Auslösendes Ereignis: Wenn dieses Ereignis eintritt, wird der Prozess initiiert.

Beschreibung: 1 erste Aktion  
2 zweite Aktion

Erweiterung: 1a Erweiterung des Funktionsumfanges von 1

Alternativen: 1a Alternative Ausführung zur ersten Aktion

1b Weitere Alternative zur ersten Aktion

Ein Geschäftsprozess kann nur ausgeführt werden, wenn die Vorbedingung erfüllt ist. Hierbei kann aber die Nachbedingung eines Prozesses die Vorbedingung eines anderen sein, wodurch die Reihenfol-

ge der Geschäftsprozesse festgelegt wird. Bei der Formulierung der Geschäftsprozesse werden die einzelnen Aufgaben in der Beschreibung nummeriert. Hierbei wird der Standardfall zuerst beschrieben. Seltener vorkommende Fälle werden als Erweiterungen hinzugefügt, wenn sie zusätzlich zu einer Aktion der Standardverarbeitung ausgeführt werden. Ersetzen Aktionen die Standardaktionen, werden sie unter Alternativen aufgenommen.

Die Kategorie eines Geschäftsprozesses bestimmt die Häufigkeit, mit der das notwendige Verhalten benötigt wird.

primär:	häufig benötigtes, notwendiges Verhalten
sekundär:	selten benötigtes, notwendiges Verhalten
optional:	nicht notwendiges, aber das System unterstützendes Verhalten

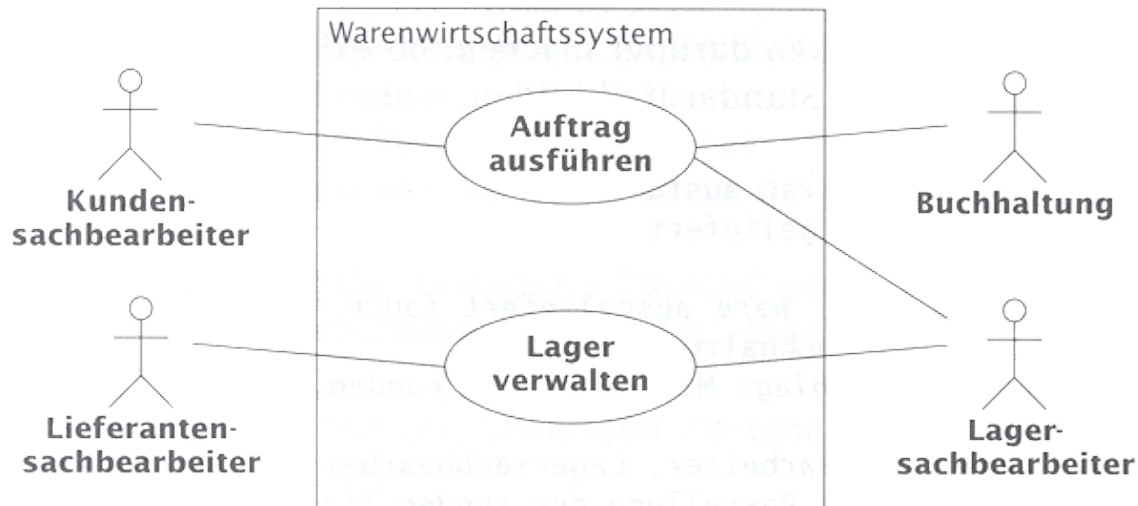
Beispiel des obigen Geschäftsprozesses:

Geschäftsprozess:	Auftrag ausführen
Ziel:	Ware an Kunden liefern
Kategorie:	primär
Vorbedingung:	-
Nachbedingung:	Erfolg: Ware ausgeliefert (auch Teilmenge), Rechnung an Buchhaltung gegeben
Nachbedingung:	Misserfolg: Mitteilung an Kunden, dass nichts lieferbar ist
Akteure:	Kundensachbearbeiter, Lagersachbearbeiter und Buchhaltung
Auslösendes Ereignis:	Bestellung des Kunden liegt vor.
Beschreibung:	1 Kundendaten abrufen 2 Lieferbarkeit prüfen 3 Rechnung erstellen 4 Auftrag vom Lager ausführen lassen 5 Rechnungskopie an Buchhaltung weitergeben
Erweiterung:	1a Kundendaten aktualisieren
Alternativen:	1a Neukunden aufnehmen 3a Rechnung mit Nachnahme erstellen 3b Rechnung mit Bankeinzug erstellen

#### 4..10..7 Geschäftsprozessdiagramm

Einen guten Überblick über die Zusammenhänge der Geschäftsprozesse untereinander liefert das Geschäftsprozessdiagramm. In ihm wird das Zusammenspiel zwischen den Akteuren und den Geschäftsprozessen dargestellt. In der Notation werden die Akteure durch

Strichmännchen gekennzeichnet die Geschäftsprozesse durch Ovale. Die Kommunikation zwischen Akteuren und Geschäftsprozessen wird durch eine entsprechende Verbindungslinie dargestellt.



**Abbildung 4.20 Geschäftsprozessdiagramm für ein Informationssystem**

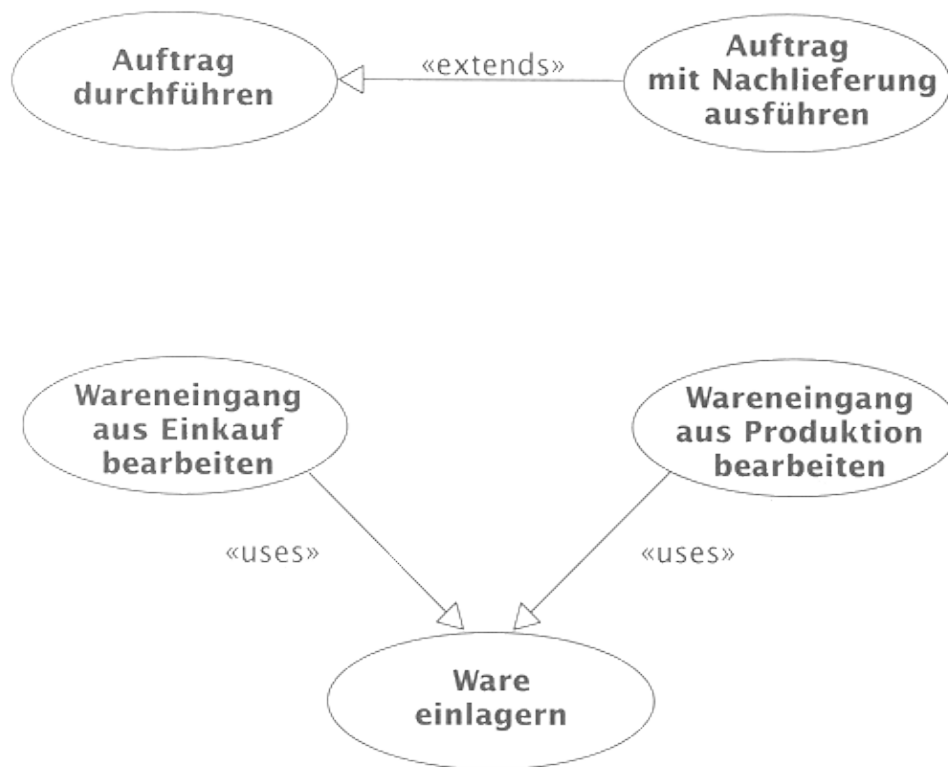
#### **4..10..8 Extends-Beziehung**

Die zwischen den Geschäftsprozessen bestehenden Beziehungen können in zwei Arten aufgeteilt werden. Eine Extends-Beziehung liegt vor, wenn ein Prozess B die Funktionalität des Prozesses A erweitert, aber im übrigen auch die Funktionalität von Prozess A hat. Die Verbindungslinie zwischen diesen Prozessen wird in diesem Fall mit <<extends>> beschriftet. Hierdurch können komplexe Prozesse zunächst vereinfacht spezifiziert werden und die Sonderfälle in die Erweiterung verlagert werden.

Sind nicht alle bestellten Artikel an Lager, wird der Kunde entsprechend informiert und erhält eine Teillieferung. Der Geschäftsprozess „Auftrag ausführen“ wird also durch „den Prozess mit Nachlieferung“ ausführen erweitert.

#### **4..10..9 uses-Beziehung**

Eine uses-Beziehung liegt vor, wenn zwei Prozesse B1 und B2 ein gemeinsames Verhalten haben, das in einem Prozess A spezifiziert ist. Prozess A wird hierbei von den Prozessen B1 und B2 quasi wie ein Unterprogramm aufgerufen. Hierdurch wird eine redundante Beschreibung des gleichen Verhaltens vermieden. In einer solchen Beziehung wird die Verbindungslinie zwischen den Prozessen mit <<uses>> beschriftet.



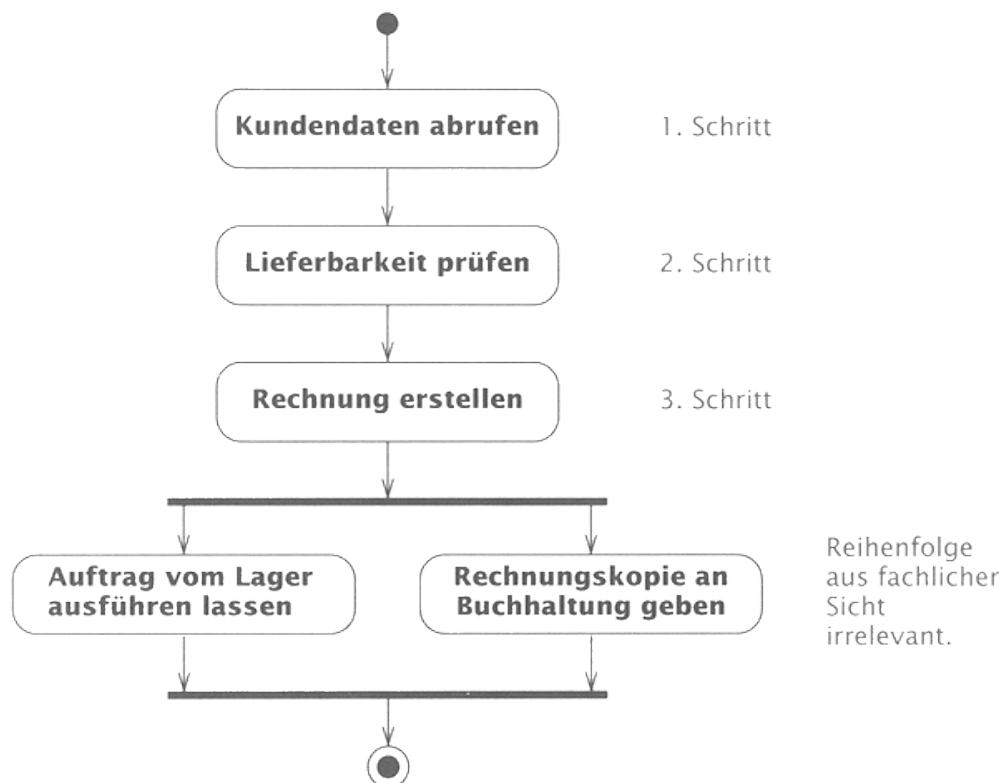
**Abbildung 4.21 extends- und uses-Beziehung**

#### **4..10..10 Konkrete und abstrakte Geschäftsprozesse**

In der oben gezeigten Abbildung sind die beiden Geschäftsprozesse „Auftrag durchführen“ und „Auftrag mit Nachlieferung durchführen“ zwei Prozesse, die sich aus konkreten Anforderungen des Ein- bzw. Verkaufs ableiten lassen. Es sind sogenannte konkrete Geschäftsprozesse. Der Prozess Ware einlagern hingegen ist ein künstliches Gebilde, das nur zur Unterstützung der beiden anderen Prozesse existiert. In dem Fall spricht man von einem abstrakten Geschäftsprozess.

#### **4..10..11 Aktivitätsdiagramm**

Mit der oben vorgestellten Schablone lassen sich Geschäftsprozesse einfach aber effektiv darstellen. Nicht darstellbar ist innerhalb dieser Schablone allerdings die Sequenz der im Rahmen eines Geschäftsprozesses durchzuführenden Schritte. Hierfür bietet die UML die Notation des Aktivitätsdiagramms. Es ist ein Sonderfall eines Zustandsdiagramms, das weiter unten behandelt wird.



**Abbildung 4.22 Aktivitätsdiagramm Auftragsbearbeitung**

## Botschaft

*Eine Botschaft ist die Aufforderung eines Senders (client) an einen Empfänger (server, supplier) eine Dienstleistung zu erbringen.*

Hierbei wird die Botschaft vom Empfänger interpretiert und eine Operation ausgeführt. Der Sender weiß nicht wie die zur Botschaft gehörende Operation ausgeführt wird. Die Summe aller Botschaften, auf die ein Objekt einer Klasse reagieren kann, wird als Protokoll der Klasse bezeichnet. Botschaft und Operation haben den gleichen Namen. Das Verhalten eines objektorientierten Systems wird somit durch die Botschaften beschrieben, über die die Objekte miteinander kommunizieren.

Empfängt ein Objekt in einer Vererbungsstruktur eine Botschaft, für die es in seiner eigenen Liste der Operationen keine Entsprechung findet, sucht es nach einer entsprechenden Operation in der Liste der Oberklasse. In der Literatur wird für den Begriff Botschaft auch häufig der Begriff „Nachricht“ verwendet. Er entspricht dem in der englischen Literatur verwendeten Begriff „message“.



## Szenario

### 4..10..12 Definition

*Unter einem Szenario versteht man eine Sequenz von Verarbeitungsschritten, die unter bestimmten Bedingungen ausgeführt werden, um das Hauptziel des Akteurs zu erreichen. Sie beginnen mit dem auslösenden Ereignis und enden mit dem Ziel des Akteurs.*

Aus dem vorhergehenden Beispiel eines Geschäftsprozesses zur Auftragsbearbeitung lassen sich drei Szenarios ableiten:

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

Szenarios werden durch Interaktionsdiagramme modelliert. Die UML bietet zwei verschiedene Arten an: das Sequenzdiagramm und das Kollaborationsdiagramm.

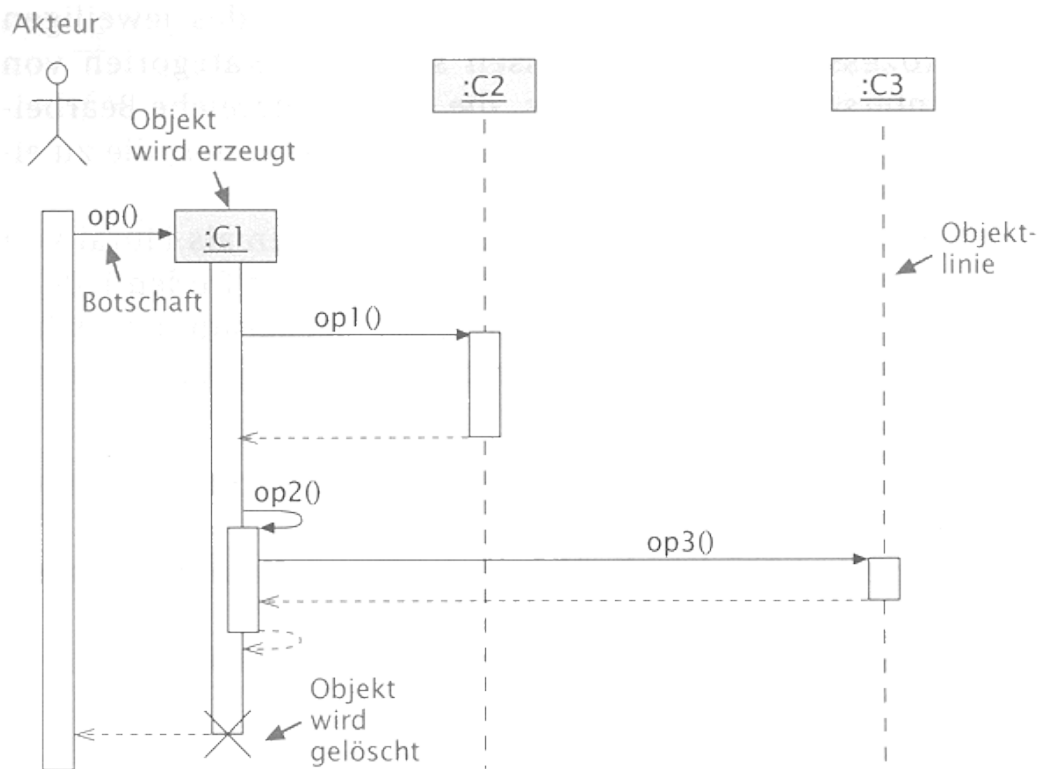
### 4..10..13 Das Sequenzdiagramm

Ein Sequenzdiagramm besitzt zwei Dimensionen. Die Vertikale repräsentiert die Zeit, wohingegen auf der Horizontalen die Objekte eingetragen werden. Jedes Objekt wird durch eine senkrechte gestrichelte Linie – der Objektlinie – dargestellt. Durch diese Linie wird die Existenz des Objektes während eines Zeitraums repräsentiert. Die Linie beginnt mit der Entstehung des Objektes und endet mit dem Löschen desselben. Das Löschen eines Objektes wird durch ein großes X gekennzeichnet. Existiert ein Objekt über den gesamten Ausführungszeitraum eines Szenarios, so wird dies durch eine von oben nach unten durchgehende gestrichelte Linie gekennzeichnet.

Am oberen Ende einer Linie wird ein Objektsymbol eingetragen. Da es sich im allgemeinen nicht um spezielle Objekte handelt, wird statt des Objektnamens in der UML der Name der Klasse angegeben (:Klassenname).

In horizontaler Richtung werden die Botschaften eingetragen, die an die jeweiligen Objekte gesendet werden. Jede Botschaft wird mit einem Pfeil mit geschlossener Pfeilspitze vom Sender zum Empfänger eingetragen und mit dem Namen der Operation beschriftet. Wird ein Objekt zu diesem Zeitpunkt erzeugt, weist die Pfeilspitze auf dieses Objekt ansonsten auf die Objektlinie. Die Operation selber wird als schmales Rechteck auf der Objektlinie eingetragen. Nach Beenden einer Operation führt eine blaue gestrichelte Linie von der Operation zum Sender, die an ihrem zum Sender weisenden Ende eine offene

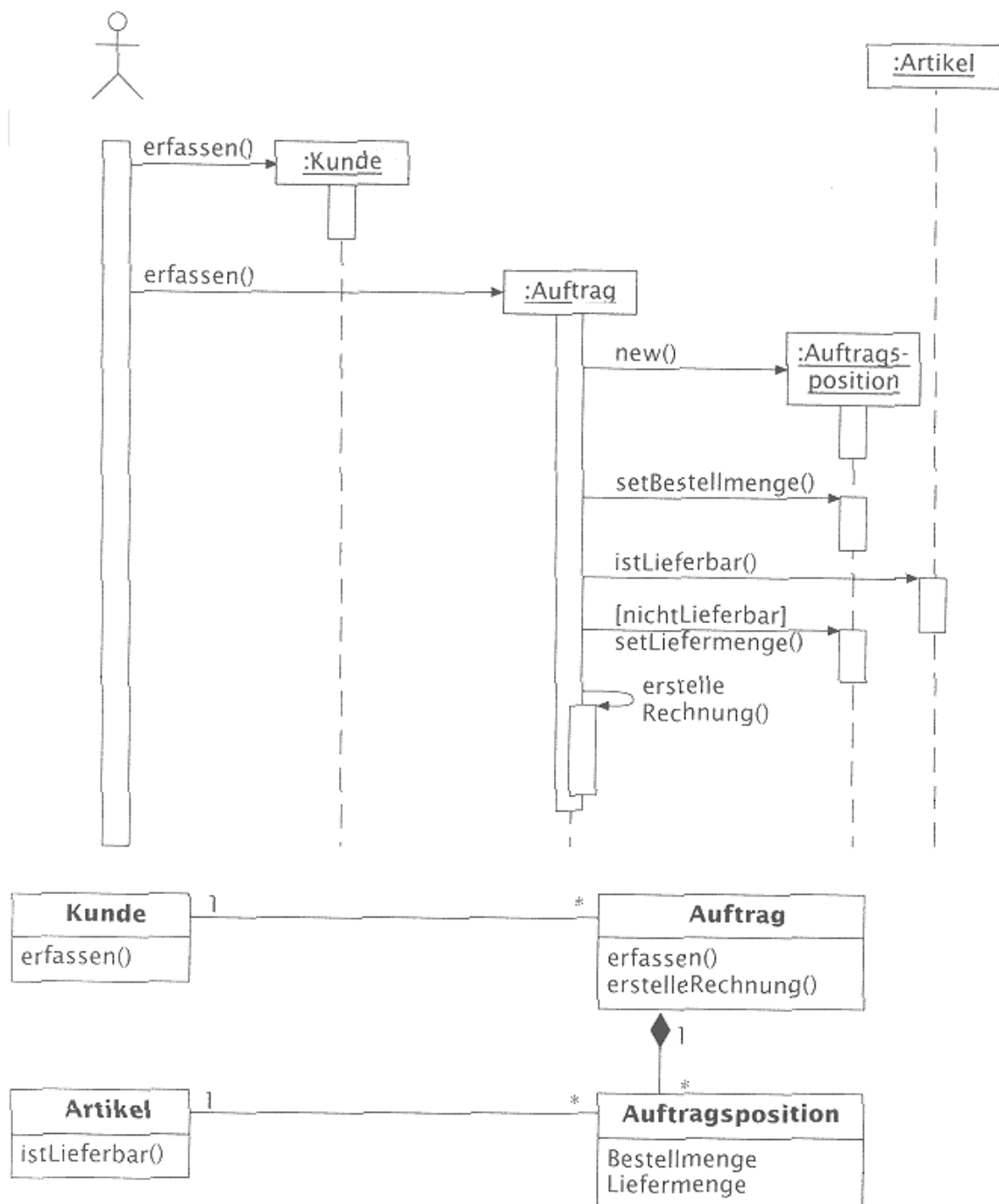
Pfeilspitze hat. Dies ist insbesondere bei paralleler Datenverarbeitung oder asynchronen Botschaften wichtig und sollte hier nicht weggelassen werden.



**Abbildung 4.23 Notation Sequenzdiagramm**

In der Analyse werden Sequenzdiagramme verwendet, um Abläufe so präzise zu beschreiben, dass deren fachliche Korrektheit diskutiert werden kann. Hierbei kann kein absoluter Detaillierungsgrad genannt werden. Dieser kann nur im Einzelfall festgelegt werden.

Die UML sieht auch die Angabe von Bedingungen oder Wiederholungen innerhalb eines Sequenzdiagramms vor. Hierbei wird die Bedingung der Operation in eckigen Klammern [ ] vorangestellt. Wiederholungen erhalten in der Notation einen Stern vor die Bedingung `*[Bedingung] operation()`. Wird die Anzahl der Wiederholungen hierbei nicht angegeben, so bedeutet dies, dass die Anzahl unbestimmt ist.



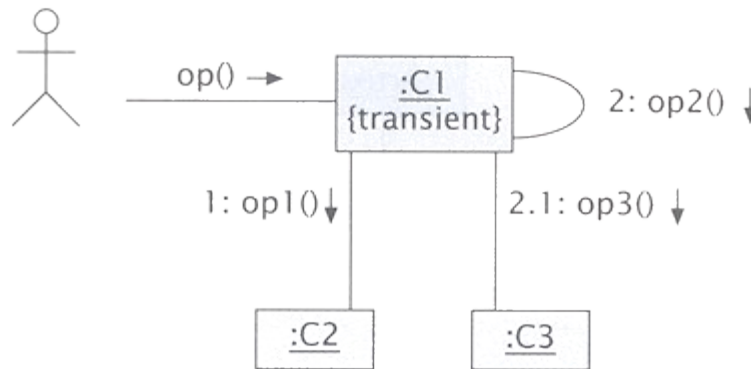
**Abbildung 4.24 Bedingungen und Wiederholungen in der Notation**

Ein Sequenzdiagramm muss mit dem Klassendiagramm konsistent sein. Alle Botschaften, die an ein Objekt gesendet werden, müssen in der Operationsliste der Klasse vorhanden sein. Verwaltungsoperationen werden im Sequenzdiagramm eingetragen, um die Kommunikation vollständig zu beschreiben. Im Klassendiagramm hingegen müssen sie nicht explizit modelliert werden.

#### 4..10..14 Das Kollaborationsdiagramm

Eine Alternative zum Sequenzdiagramm ist das Kollaborationsdiagramm. Es beschreibt – wie das Sequenzdiagramm – die Objekte jedoch auch die Verbindung zwischen diesen Objekten. An jede Ver-

bindung kann eine Botschaft in Form eines Pfeil angetragen werden. In Abbildung 4.25 sendet der Akteur eine Botschaft `op()` an ein neu erzeugtes Objekt der Klasse `:C1`. Dieses Objekt aktiviert danach zunächst `op1()` und dann `op2()`, die wiederum ihrerseits die Operation `op3()` aufruft. Die Reihenfolge wird hierbei durch die Nummerierung 1, 2 und 2.1 bestimmt.



**Abbildung 4.25 Notation des Kollaborationsdiagramms**

Wird ein Objekt beim Senden der Botschaft neu erzeugt, wird es mit `{new}` gekennzeichnet. Wird es im Laufe der Ausführung gelöscht, erfolgt die Kennzeichnung durch `{destroyed}`. Dies gilt für die Verbindungen analog. Wird ein Objekt innerhalb einer Ausführung sowohl erzeugt als auch wieder zerstört, wird es als `{transient}` gekennzeichnet.

Bei den Verbindungen unterscheidet man zwischen permanenten Verbindungen und temporären Verbindungen. Besteht zwischen zwei Objekten nur für die Dauer der Kommunikation eine Verbindung, so gilt diese als temporär. Diese Verbindung kann auch ohne Vorliegen einer Assoziation bestehen. Die permanenten Verbindungen entsprechen den Assoziationen zwischen den Objekten. In der Notation werden temporäre Verbindungen durch `<<temp>>` ausgewiesen, um sie von den Assoziationen zu unterscheiden.

Sequenzdiagramme heben den zeitlichen Aspekt des dynamischen Verhaltens hervor und sind deshalb gut geeignet, die Reihenfolge und die Verschachtelung der Operationen zu erkennen. Sie eignen sich daher insbesondere für die Modellierung komplexer Szenarios, da sie die Reihenfolge der Botschaften transparent wiedergeben. Es lassen sich mehrere externe Operationen, die durch einen Akteur angestoßen werden, eintragen. Das Kollaborationsdiagramm betont im Gegensatz hierzu die Verbindungen zwischen den Objekten. Die Reihenfolge der Operationen sowie die Verschachtelung der Objekte wird hier durch eine hierarchische Nummerierung angegeben. Dies hat jedoch den Nachteil, die Reihenfolge weniger überschaubar darzustellen. Für den Analytiker bringt es jedoch den Vorteil, sich nicht gleich auf eine Ausführungsreihenfolge festlegen zu müssen, sondern sich zunächst auf die Beschreibung der Kommunikation zwischen den Objekten zu konzentrieren und erst in einem weiteren

Schritt die Reihenfolge hinzuzufügen. Mit Kollaborationsdiagrammen lassen sich sehr gut die Wirkungen komplexer Operationen beschreiben. Allerdings ist im Gegensatz zum Sequenzdiagramm für jede externe Operation ein eigenes Kollaborationsdiagramm erforderlich.

## **Zustandsautomat**

### **4..10..15 Definition**

*Ein Zustandsautomat besteht aus Zuständen und Zustandsübergängen (Transitionen). Ein Zustand ist eine Zeitspanne, in der ein Objekt auf ein Ereignis wartet. In einen Zustand gelangt ein Objekt immer durch ein Ereignis. Ein Ereignis hat keine Dauer. Ein Objekt kann nacheinander mehrere Zustände durchlaufen, kann sich aber zu einem Zeitpunkt in nur genau einem Zustand befinden. Tritt innerhalb eines Zustandes ein Ereignis ein, so ist der neue Zustand abhängig zum einen von dem Zustand, in dem das Objekt sich befand als das Ereignis eintrat, zum anderen von dem Ereignis das eintrat.*

Der Zustand eines Objekt enthält damit implizit Informationen, die sich aus den bisherigen Eingaben ergeben haben. In der Objektorientierung dient das Zustandsdiagramm zur Darstellung des Zustandsautomaten.

### **4..10..16 Zustandsname**

Jeder Zustand kann einen Namen haben. Zustände denen kein Name zugeordnet ist, nennt man anonyme Zustände. Ein benannter Zustand kann in einem Zustandsdiagramm mehrfach verwendet werden, um die Lesbarkeit des Diagramms zu erhöhen. Diese Zustände sind alle identisch. Als Zustandsnamen sollte immer ein Adjektiv verwendet werden, auch dann, wenn mit dem Zustand eine Verarbeitung verbunden ist.

### **4..10..17 Aktionen**

Mit einem Zustand können Aktionen oder Aktivitäten verbunden sein. Eine Aktion, die beim Eintritt in einen Zustand – unabhängig durch welche Transition – durchgeführt wird, nennt man eine entry-Aktion. Sie wird immer ausgeführt beim Eintritt in diesen Zustand. Das Gegenstück hierzu ist die exit-Aktion. Sie wird jedes Mal durchlaufen, wenn der Zustand verlassen wird und in einen anderen Zustand übergeht. Eine Aktivität beginnt, wenn ein Objekt einen Zustand einnimmt und endet, wenn es ihn wieder verlässt.

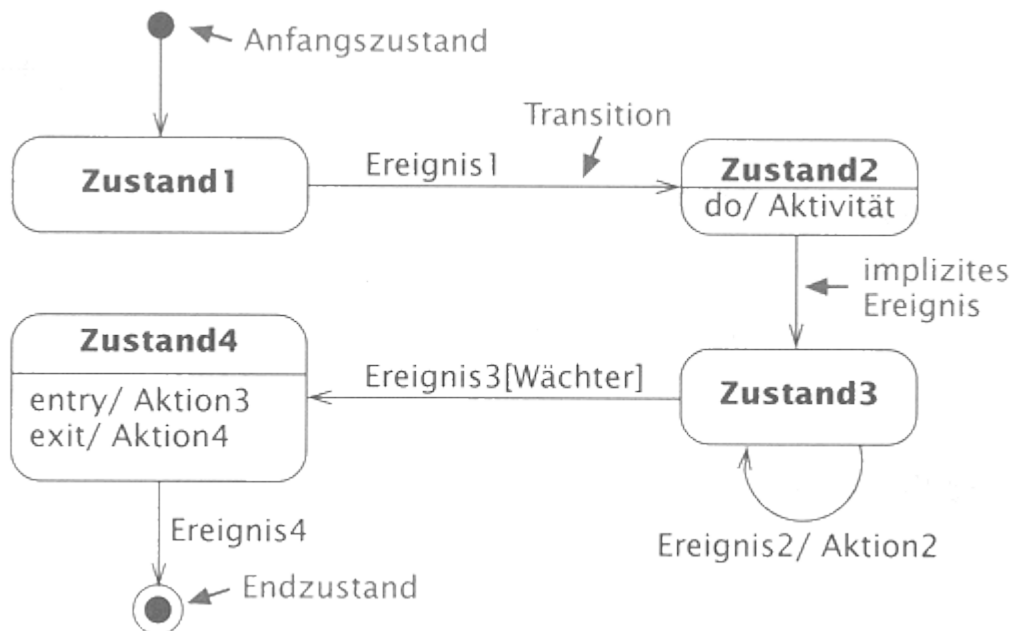
### **4..10..18 Anfangszustand**

Der Anfangszustand wird im Diagramm durch einen kleinen schwarzen Kreis dargestellt. Es handelt sich um einen sogenannten Pseudozustand, der nur durch eine Transition mit einem richtigen Zustand verbunden ist. Der Anfangszustand kann z. B. mit einem Ereignis zum Erzeugen eines Objektes beschriftet werden. Das Objekt selber kann sich nie in diesem Zustand befinden. Ein neu erzeugtes Objekt

befindet sich nach seiner Erzeugung in dem „echten“ Folgezustand des Anfangszustandes. Beim Anfangszustand handelt es sich um einen Scheinzustand. Er stellt ein grafisches Hilfsmittel dar.

#### 4..10..19 Endzustand

Im Endzustand hört ein Objekt auf zu existieren. Aus diesem Zustand führt keine Transition heraus. Auch hier handelt es sich um einen Pseudozustand, der im Diagramm als Bullauge dargestellt wird.



**Abbildung 4.26 Notation des Zustandsdiagramm mit Anfangs- und Endzustand**

#### 4..10..20 Transition

Eine Transition oder Zustandsübergang verbindet zwei Zustände. Sie wird durch einen offenen Pfeil dargestellt. Eine Transition wird stets durch ein Ereignis ausgelöst und kann nicht unterbrochen werden. Tritt ein Ereignis ein, das Objekt befindet sich jedoch in einem Zustand, in dem es auf dieses Ereignis nicht reagieren kann, wird es ignoriert. Sind Ausgangs- und Folgezustand einer Transition identisch, so muss beachtet werden, dass entry-Aktion und exit-Aktion ausgeführt werden.

#### 4..10..21 Ereignisse

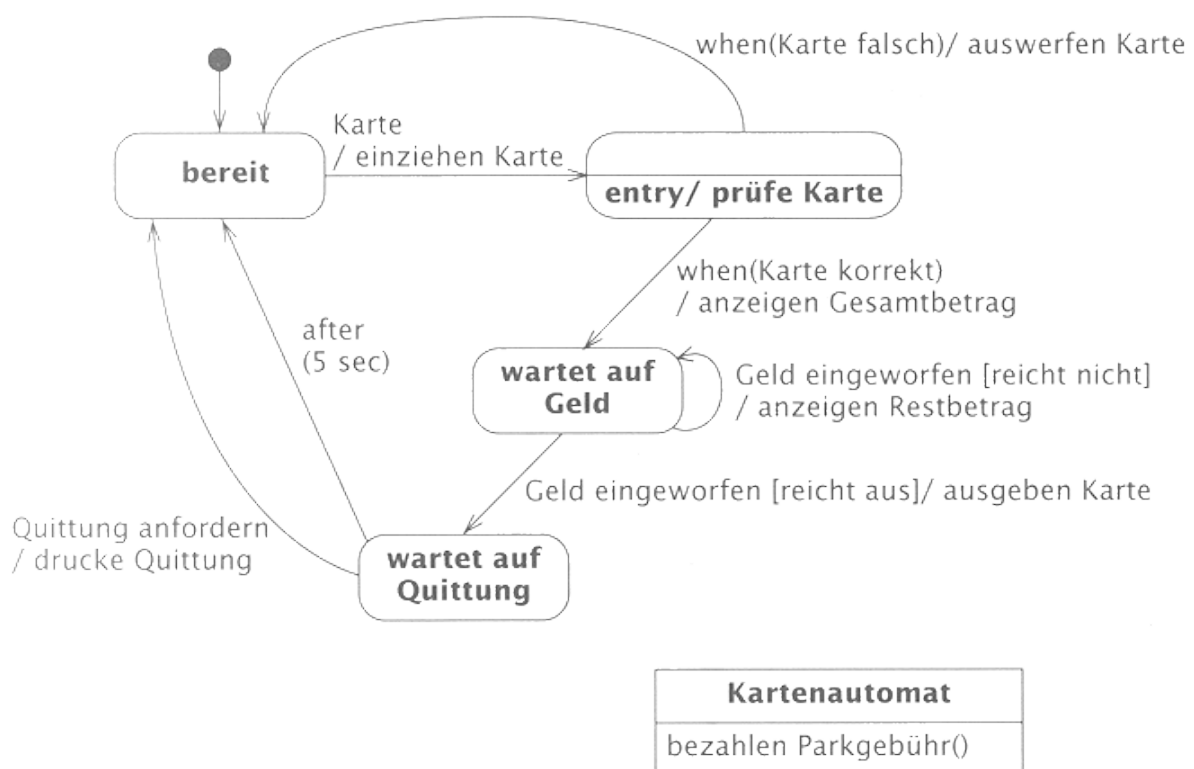
Ereignisse können beispielsweise sein:

- Eine Bedingung die wahr ist
- Ein Signal
- Eine Botschaft (eine Operation die aufgerufen wird)
- Eine verstrichene Zeitspanne
- Ein Zeitpunkt

In den beiden letzten Fällen handelt es sich um zeitliche Ereignisse.

#### 4..10..22 Notation

Ein Transition kann an eine Bedingung geknüpft sein z. B. `when(Temperatur > 100)`. Die Transition wird in diesem Fall immer ausgelöst, wenn die Bedingung erfüllt ist. Botschaften werden durch einen Namen beschrieben und können Parameter haben z. B. `Maus-taste_gedrückt(Mausposition)`. Die Transition wird ausgelöst – man sagt auch feuert –, wenn die Botschaft gesendet wird. Auch eine verstrichene Zeitspanne kann eine Transition auslösen. Dies wird beispielsweise gekennzeichnet durch `after(10 Sekunden)`. In dem Fall feuert die Transition nach Ablauf der Zeitspanne. Ein Zeitpunkt wird in der Notation angegeben durch `when(Datum = 1.1.2000)`.



**Abbildung 4.27 Zustandsautomat eines Kartenautomaten mit Bedingungen**

#### 4..10..23 Wächter

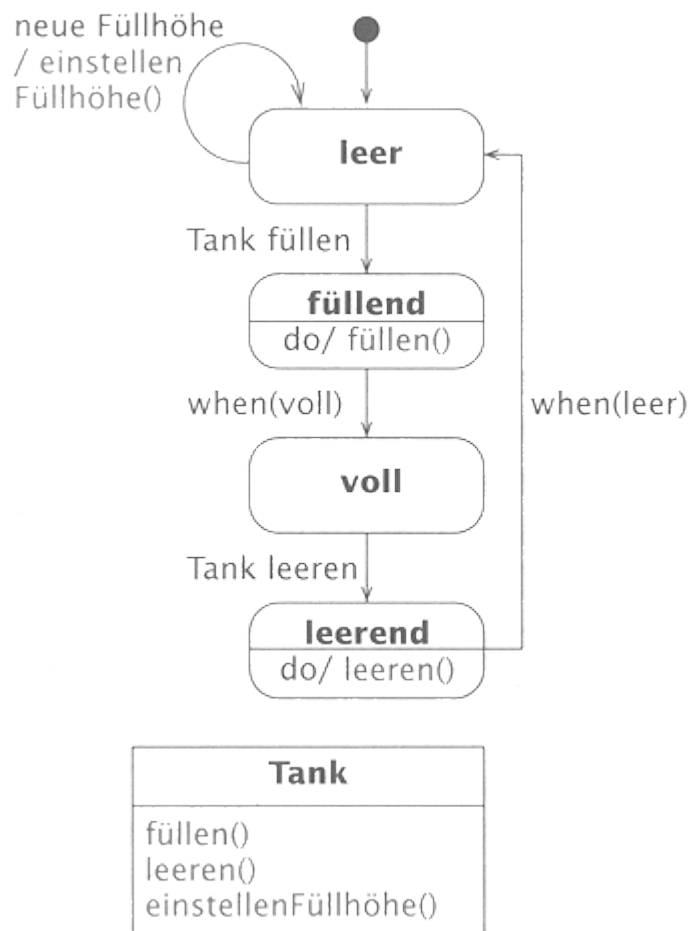
Diese Ereignisse können mit einem sogenannten Wächter (guard condition) kombiniert werden. Auch der Wächter ist eine Bedingung, die sich jedoch von den oben beschriebenen Bedingungen unterscheidet. Tritt ein Ereignis ein, wird zunächst der Wächter ausgewertet. Ist die dort spezifizierte Bedingung erfüllt, feuert die Transition (guarded transition).

#### 4..10..24 Implizite Ereignisse

Jeder Zustand darf eine ausgehende Transition besitzen, die nicht von einem explizit angegebenen Ereignis abhängt. Diese Transition wird ausgeführt, wenn die Verarbeitung, die mit dem Zustand verbunden ist, beendet ist. Ein solches Ereignis nennt man implizites Ereignis.

#### 4..10..25 Lebenszyklus

Häufig werden in objektorientierten Methoden Zustandsautomaten verwendet, um die Lebenszyklen (live cycle) der Objekte einer Klasse zu beschreiben.



**Abbildung 4.28** Zustandsautomaten eines Tanks als Darstellung der Lebenszyklen

#### 4..10..26 Verfeinerung von Zuständen

Zustände lassen sich durch Unterzustände (substates) verfeinern. Dabei schließen sich alle Unterzustände gegenseitig aus. Man nennt einen Zustand, der durch Unterzustände verfeinert wird, auch zusammengesetzten Zustand. Wie der Hauptzustand verfügen auch die Unterzustände über Anfangs- und Endzustände.

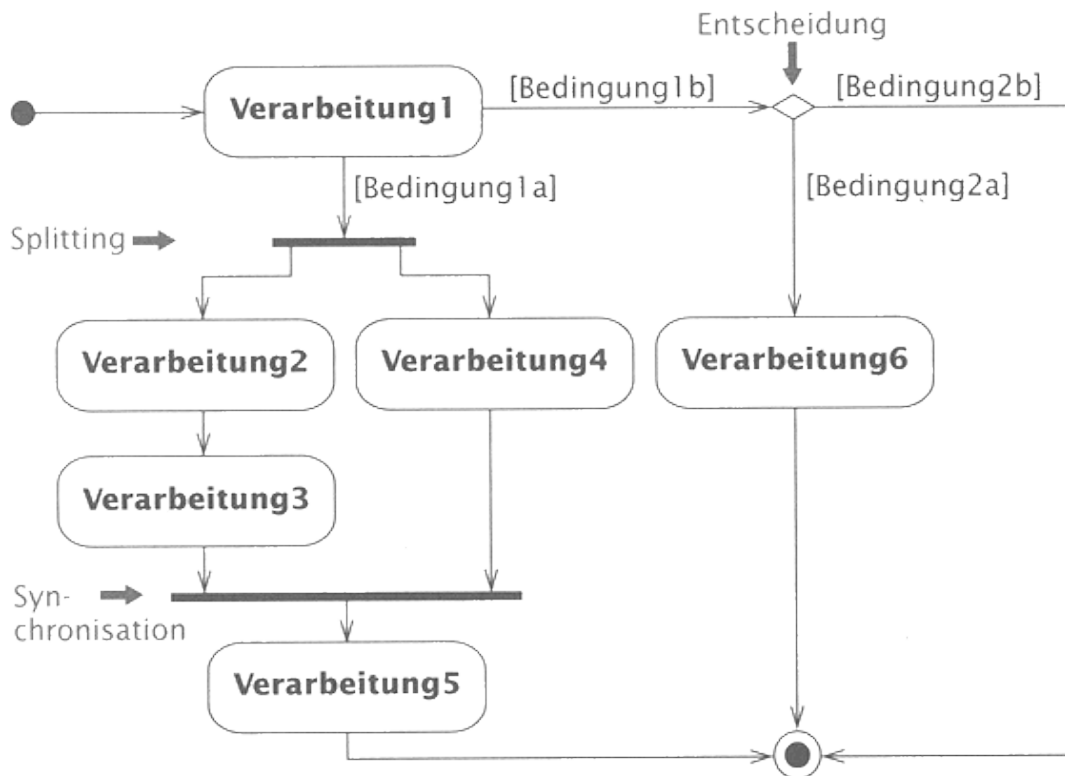
#### 4..10..27 Aktivitätsdiagramm

Ein Sonderfall des Zustandsdiagramms ist das Aktivitätsdiagramm. Es beschreibt nicht wie das Zustandsdiagramm die Reaktion auf Ereignisse, sondern spezifiziert die interne Verarbeitung. Bei diesem Diagramm sind die meisten Zustände mit Verarbeitungen verknüpft. Ein Zustand wird verlassen, wenn die mit ihm verbundene Verarbeitung beendet ist. Dabei kommen weder in einem Zustand noch in einer Transition explizite Ereignisse vor.



#### 4..10..28 Notation des Aktivitätsdiagramms

Abbildung 4.29 zeigt die Notation des Aktivitätsdiagramms. Jeder Zustand modelliert einen Schritt innerhalb der Gesamtverarbeitung. Nach Abschluss der Verarbeitung1 wird entsprechend den Bedingungen [Bedingung1a] und [Bedingung1b] verzweigt. Eine Entscheidung kann hierbei auch mittels einer Raute dargestellt werden.



**Abbildung 4.29 Notation des Aktivitätsdiagramms**

Außerdem kann spezifiziert werden, ob Verarbeitungsschritte nacheinander ausgeführt werden müssen oder ob eine Parallelverarbeitung möglich ist (Splitting). Im Anschluss an eine Parallelverarbeitung erfolgt eine Synchronisation, die sicherstellt, dass alle parallel verlaufenden Verarbeitungen korrekt beendet sind bevor mit dem nächsten Verarbeitungsschritt fortgefahren wird.

## **5. Gestaltung der Benutzungsoberfläche**

Nach Vorliegen des OOA-Modells wird der Prototyp der Benutzungsoberfläche gestaltet. Der Prototyp enthält bereits Fenster, Menüs und die Dialogführung der zu erstellenden Software. Er enthält jedoch keine Daten und keine Dialogführung innerhalb von Datenfeldern also kein scrolling innerhalb einer Liste u .ä.. Die Entwicklung des Prototyps sollte unter Berücksichtigung der Software-Ergonomie geschehen.

### **Einführung in die Software Ergonomie**

Die Software-Ergonomie befasst sich mit der menschengerechten Gestaltung von Softwaresystemen. Mit ihr soll das System an die Eigenschaften und Bedürfnisse des Benutzers angepasst werden.

#### **5..1. Das GUI-System**

Das GUI-System ist die Schnittstelle zwischen dem Benutzer und der Anwendung. Sie enthält zum einen Dialogelemente (Buttons, Optionbuttons, Checkboxes, usw.) für die Bedienungsabläufe und E/A-Komponenten für die Informationsdarstellung (Textfelder, Labels usw.). Idealerweise verwendet man für die Erstellung des Prototyps das gleiche GUI-System, das zur Realisierung der Benutzungsoberfläche im Entwurf angewandt wird. Dadurch ist der Prototyp kein „Wegwerf“-Produkt, sondern kann weiterentwickelt werden.

#### **5..2. Das Gestaltungsregelwerk**

Ein Gestaltungsregelwerk (style guide) schreibt vor, wie die Benutzungsoberfläche zu gestalten ist. Sie bestimmt z. B. das Aussehen von Fenstern, Menüs und anderen Fensterelementen. Diese Festbeschreibung soll dazu dienen, dem Benutzer immer einheitlich gestaltete Benutzungsoberflächen zur Verfügung zu stellen, um eine Einarbeitung zu erleichtern. Das Gestaltungswerk kann zum Beispiel der Style Guide von Windows sein, den Microsoft zur Entwicklung von Applikationen anbietet und der die Windows-typischen Dialoge und Fenster beschreibt. Es kann sich dabei allerdings auch um ein unternehmenseigenes Gestaltungsregelwerk handeln.

### **Dialoggestaltung**

Ein Dialog ist eine Interaktion zwischen einem Benutzer und einem Dialogsystem, um ein bestimmtes Ziel zu erreichen. Ein Benutzer ist ein Mensch, der mit dem Dialogsystem arbeitet /ISO 9241-10 : 1996/.

Aus Sicht des Benutzers lassen sich Primär- und Sekundärdialoge unterscheiden. Hierbei gehören Primärdialoge zur direkten Aufgabenerfüllung. Sie sind erst beendet, wenn das Arbeitsziel erreicht ist. Die Sekundärdialoge dienen dem Benutzer als zusätzliche situationsabhängige Informationsquelle. Nach Beenden eines Sekundärdialogs wird der Primärdialog fortgesetzt.

### **5..3. modale und nicht-modale Dialoge**

Es gibt für Dialoge zwei unterschiedliche Modi, der sogenannte modale Dialog und den nicht-modalen Dialog. Der modale Dialog muss beendet sein, bevor mit einer Anwendung weiter fortgefahren werden kann. Eventuell kann ein solcher Dialog nicht nur anwendungsmodal sondern sogar systemmodal sein. Systemmodal bedeutet, dass der Dialog geschlossen werden muss, bevor überhaupt eine weitere Aktion im System möglich ist. Ein nicht-modaler Dialog hingegen kann geöffnet bleiben, während im ursprünglichen Fenster weitergearbeitet wird.

### **5..4. SDI und MDI-Anwendungen**

Man unterscheidet zwei Anwendungstypen: die SDI-Anwendung und die MDI-Anwendung. Innerhalb einer SDI-Anwendung (Single document interface) kann zu einem Zeitpunkt genau ein Dokument geöffnet sein (z. B. Notepad). Eine MDI-Anwendung (multiple document interface) erlaubt das Öffnen mehrerer Dokumente oder mehrerer Sichten auf ein Dokument gleichzeitig (z. B. Word). Dabei ist ein beliebiger Wechsel von einem zum anderen Dokument möglich.

### **5..5. Ergonomische Anforderungen für Bürotätigkeiten mit Bildschirmgeräten (Teil 10: Grundsätze der Dialoggestaltung ISO 9241-10 : 1996)**

Für die Dialoggestaltung gibt es eine Reihe von Richtlinien, wovon die folgenden sieben Grundsätze als wichtig erkannt wurden:

➤ **Aufgabenangemessenheit**

Ein Dialog ist aufgabenangemessen, wenn er seinen Benutzer unterstützt, seine Arbeitsaufgabe effektiv und effizient zu erledigen (z. B. Positionierung der Eingabemarke auf das erste Feld des Dialoges)

➤ **Selbstbeschreibungsfähigkeit**

Ein Dialog ist selbstbeschreibungsfähig, wenn jeder einzelne Dialogschritt durch Rückmeldung des Systems verständlich ist bzw. dem Benutzer auf Anfrage erklärt wird (z. B. Anfrage zur Bestätigung eines Löschvorgangs, wenn die Löschung nicht rückgängig gemacht werden kann).

➤ **Steuerbarkeit**

Ein Dialog ist steuerbar, wenn der Benutzer in der Lage ist, den Dialogablauf zu starten sowie seine Richtung und Geschwindigkeit zu beeinflussen, bis das Ziel erreicht ist (z. B. Bewegen der Einfügemarke zu den Eingabefeldern und Steuerelementen).

➤ **Erwartungskonformität**

Ein Dialog ist erwartungskonform, wenn er konsistent ist und den Merkmalen des Benutzers entspricht (z. B. Ausgabe von Meldungen)

an der vom Benutzer erwarteten Stelle oder Dialoge werden stets durch Drücken der selben Taste beendet).

➤ **Fehlertoleranz**

Ein Dialog ist fehlertolerant, wenn das beabsichtigte Arbeitsergebnis trotz erkennbarer fehlerhafter Eingaben entweder mit keinem oder mit minimalem Korrekturaufwand seitens des Benutzers erreichbar ist (z. B. wird vom System ein Eingabefehler festgestellt, wird der Cursor in das entsprechende Eingabefeld zurück gesetzt).

➤ **Individualisierbarkeit**

Ein Dialog ist individualisierbar, wenn das Dialogsystem Anpassungen an die Erfordernisse der Arbeitsaufgabe sowie an die individuellen Fähigkeiten und Vorlieben des Benutzers zulässt (z. B. Steuerung der Geschwindigkeit des scrollings).

➤ **Lernförderlichkeit**

Ein Dialog ist lernförderlich, wenn er den Benutzer beim Erlernen des Dialogsystems unterstützt und anleitet (z. B. das Lernen wird dadurch unterstützt, dass der Benutzer ohne die Gefahr potentieller katastrophaler Ergebnisse am System experimentieren kann).

## **5..6. Alternativen der Dialoggestaltung**

Prinzipiell kann man für die Dialoggestaltung zwischen drei verschiedenen Bedienungsarten wählen.

### **objektorientierte Bedienung**

Bei der objektorientierten Bedienung wählt der Benutzer zunächst das Objekt, das bearbeitet werden soll, und danach den Bearbeitungsschritt (Pop-up-Menüs).

### **funktionsorientierte Bedienung**

Bei der funktionsorientierten Bedienung wird zunächst der Bearbeitungsschritt gewählt z. B. durch Auswahl eines Menüpunktes und danach das Objekt, mit dem dieser Bearbeitungsschritt ausgeführt werden soll.

### **direkte Manipulation**

Bei der direkten Manipulation werden die Objekte vom Benutzer in Analogie zur Schreibtischarbeit behandelt. Hierzu gehört zum Beispiel das Verschieben oder Kopieren von Objekten mittels drag & drop.

### **Fenster**

Zur Gestaltung einer Benutzungsoberfläche stehen verschiedene Fensterformen zur Verfügung. Es wird zwischen Primärfenster und Sekundärfenster unterschieden. In den Primärfenstern finden die Hauptaktionen der Benutzer statt, in den Sekundärfenstern werden die Eingaben von Optionen und sekundäre Aktivitäten durchgeführt.

### **5..7. Anwendungsfenster**

Das wichtigste Primärfenster ist das Anwendungsfenster. Es ist sozusagen der Container für die Anwendung oder das „parent window“. Aus ihm heraus werden alle anderen Fenster der Anwendung geöffnet. Dieses Fenster enthält mindestens die Titelleiste, einen Menübalken und den Arbeitsbereich der Anwendung. Bei einer SDI-Anwendung findet die Interaktion mit dem Benutzer vorwiegend in diesem Fenster statt. Bei einer MDI-Anwendung ist dieser Bereich leer.

### **5..8. Unterfenster**

In einer MDI-Anwendung können innerhalb des Anwendungsfensters mehrere Unterfenster (child windows) geöffnet werden. Unterfenster liegen innerhalb des Anwendungsfensters und werden durch das Anwendungsfenster begrenzt. Unterfenster können innerhalb des Anwendungsfensters unterschiedlich angeordnet sein. Sie können überlappend, nebeneinander oder untereinander liegen.

### **5..9. Dialogfenster**

Ebenfalls zu den Sekundärfenstern gehört das Dialogfenster. Es dient für Sekundärdialoge, die häufig modal sind, aber durchaus auch nicht modal sein können. Dialogfenster können verschiebbar oder nicht verschiebbar sein. Ihre Größe kann jedoch anders als bei den Unterfenstern nicht verändert werden. Dialogfenster können – im Gegensatz zu den Unterfenstern – über den Rahmen des Anwendungsfensters hinaus verschoben werden. Dialogfenster sollten so wenig wie möglich von ihrem Hintergrund verdecken.

### **5..10. Mitteilungsfenster**

Eine besondere Form eines modalen Dialogfensters ist das Mitteilungsfenster. Es dient dazu, eine Information des Systems an den Benutzer auszugeben. Der Benutzer kann mit einer Aktion auf diese Mitteilung reagieren. Ein Mitteilungsfenster enthält keine Interaktionselemente zur Datenselektion.

#### **Menüs**

Menüs bestehen aus einer überschaubaren und meist vordefinierten Menge von Optionen, aus denen ein Benutzer auswählen kann.

### **5..11. Aktionsmenü**

Ein Aktionsmenü kann Anwendungsfunktionen auslösen oder in andere Menüs verzweigen. Im zweiten Fall spricht man von Kaskadenmenüs.

### **5..12. Eigenschaftenmenü**

Über ein Eigenschaftenmenü kann ein Benutzer Parameter einstellen, die das Verhalten der Anwendung beeinflussen. Hier können oft mehrere Optionen selektiert werden. Auch hier sind Kaskadenmenüs möglich. Ein Beispiel für ein solches Eigenschaftenmenü ist das Menü „Ansicht“ des Windows-Explorers, in dem die unterschiedliche

Darstellungsweise der Dateien im rechten Fenster des Explorers eingestellt werden kann.

### **5..13. Menübalken**

Der Menübalken enthält alle Menütitel. Ein Anwendungsfenster hat stets einen Menübalken, während Dialogfenster und Mitteilungsfenster keinen Menübalken haben. Ein Unterfenster kann ebenfalls einen Menübalken haben. In diesem Fall wird der Menübalken des Anwendungsfensters vom Menübalken des aktiven Unterfensters überlagert.

### **5..14. drop-down-Menü**

Drop-down-Menüs erscheinen nach Auswahl des zugehörigen Menütitels im Menübalken. Die Menüeinträge haben einen globalen Geltungsbereich. Im momentanen Kontext nicht auswählbare Optionen sind grau dargestellt.

### **5..15. pop-up-Menü**

Pop-up-Menüs erscheinen auf rechten Mausklick an der Position des Mauszeigers und beziehen sich auf das jeweils aktive Objekt. Die Optionen des pop-up-Menüs besitzen einen lokalen Geltungsbereich. Auch hier werden nur die Optionen grau dargestellt, die im momentanen Kontext nicht auswählbar sind.

### **5..16. Beschleunigte Menüauswahl**

Um geübten Benutzern eine schnellere Auswahl zu ermöglichen, als es die Auswahl mit der Maus zulässt, gibt es mehrere Möglichkeiten des schnelleren Zugriffs. Die wohl wichtigste sind die mnemonischen Kürzel. Hiermit werden einzelne unterstrichene Buchstaben der Menünamen oder -optionen bezeichnet. Durch Drücken dieser Buchstaben Tasten in Verbindung mit der ALT-Taste wird die Menüoption aufgerufen. Ein weiteres Hilfsmittel zur schnelleren Bedienung der Menüs sind Tastaturkürzel oder auch short-keys genannt. Durch Drücken der entsprechenden Tastenkombination wird die Anwendungsfunktion aufgerufen, ohne dass sich das Menü öffnet.

### **5..17. Symbolbalken**

Symbolbalken sind ebenfalls ein Hilfsmittel für eine beschleunigten Zugriff auf Anwendungsfunktionen. Oft werden in diesen Symbolbalken die am häufigsten benötigten Funktionen der Anwendung hinterlegt. Teilweise lassen sich diese Symbolbalken durch individuelle Anpassung verändern.

### **5..18. Gestaltungsregel für Menüs**

- Menütitel (von drop-down-Menüs)

Einheitliche Bezeichnung und Anordnung in allen Anwendungen und Fenstern

Kurz, prägnant und selbsterklärend

Einheitlicher grammatikalischer Stil

- Benennung der Menüoptionen

Kurz und prägnant

Einheitlich in allen Anwendungen und Fenstern

Dem Benutzer vertraut

- Gestaltung der Menüoptionen

Linksbündig anordnen

Zufällige Anordnung vermeiden, stattdessen eine alphabetisch Reihenfolge oder eine funktionale Gruppierung verwenden.

Wenn möglich, statt einer rein sprachlichen Darstellung, zusätzlich bildhaft darstellen

- Kaskadenmenüs

Maximal zweistufig (in Ausnahmefällen auch dreistufig)

Breite flache Bäume mit etwa 8 bis 16 Gruppen

Aussagefähige Gruppennamen wählen, aus denen man auf die darunterliegenden Menüoptionen schließen kann

Gruppen möglichst disjunkt

- Positionierung der pop-up-Menüs

Rechts, nahe dem aktiven Objekt, ohne dieses zu überdecken

- Mnemonische Kürzel

Gleichen Menüoptionen in mehreren Menüs die gleichen Kürzel zuordnen

Leicht merkbare Kürzel wählen.

- Abkürzungsregel

Abkürzungen möglichst vermeiden

Streichen einzelner Buchstaben, meist Vokale (z. B. Zchn für Zeichen)

Abschneiden der letzten Buchstaben eines Wortes (z. B. Dir für Directory)

### **Vom Klassendiagramm zur Dialogstruktur**

Aus dem Klassendiagramm kann systematisch eine objektorientierte Dialogstruktur abgeleitet werden. Hierbei geht man davon aus, dass jede Klasse des Analysemodells auf ein Erfassungsfenster und ein Listenfenster abgebildet wird. Der Menübalken erhält je ein drop-down-Menü für die Erfassungsfenster und die Listenfenster. Zuvor muss jedoch geprüft werden, ob für jede Klasse ein Listenfenster sinnvoll ist und ob für die Erfassung ein eigener Dialog erforderlich ist. Die ermittelten Klassen werden in den Menüs „Erfassung“ und Listen aufgeführt. Sind zu viele Klassen vorhanden, empfiehlt sich eine geeignete Gruppierung.

### 5..19. Das Erfassungsfenster

Das Erfassungsfenster bezieht sich auf genau ein Objekt der jeweiligen Klasse. In ihm wird jedes Attribut der Klasse auf ein typgerechtes Interaktionselement abgebildet. Jede Methode der Klasse wird auf eine Schaltfläche des Fensters und eventuell auf einen Eintrag in einem pop-up-Menü abgebildet. Das Erfassungsfenster wird sowohl für die Eingabe eines neuen Objektes als auch zur Änderung bereits bestehender Objekte verwendet. Dabei sollten in dem Erfassungsfenster folgende Schaltflächen vorhanden sein.

- OK  
Speichern des Objektes und Schließen des Fensters
- Übernehmen  
Speichern des Objektes ohne das Fenster zu schließen. Es können weitere Objekte eingegeben oder bearbeitet werden.
- Abbrechen  
Schließt das Fenster ohne Änderungen zu übernehmen
- Liste  
Öffnet das zur Klasse gehörende Listenfenster, während das Erfassungsfenster geöffnet bleibt.

### 5..20. Das Listenfenster

In dem Listenfenster werden alle Objekte einer Klasse angezeigt. Hierbei müssen nicht alle Attribute der Klasse im Listenfenster beschrieben werden. Hier ist es ausreichend, wenn der Benutzer die wichtigsten Attribute der Klasse einsehen kann. Bei Bedarf kann für ein selektiertes Objekt das Erfassungsfenster geöffnet werden, das alle Attribute anzeigt.

Klassenattribute und Klassenfunktionen beziehen sich auf alle Elemente einer Klasse und werden deshalb ebenfalls im Listenfenster außerhalb der Liste angezeigt. Auch werden wieder die Klassenattribute auf passende Interaktionselemente und Klassenfunktionen auf Schaltflächen abgebildet. Es könnte dabei z. B. die folgenden Schaltflächen eingesetzt werden. Eine sinnvolle Auswahl der Schaltflächen ergibt sich aus der Analyse und muss im Einzelfall festgelegt werden.

- Neu  
Öffnen eines leeren Erfassungsfensters
- Ändern  
Öffnen eines Erfassungsfensters für das selektierte Objekt
- Löschen  
Löscht das selektierte Objekt aus der Liste
- Schließen



Schließt das Listenfenster

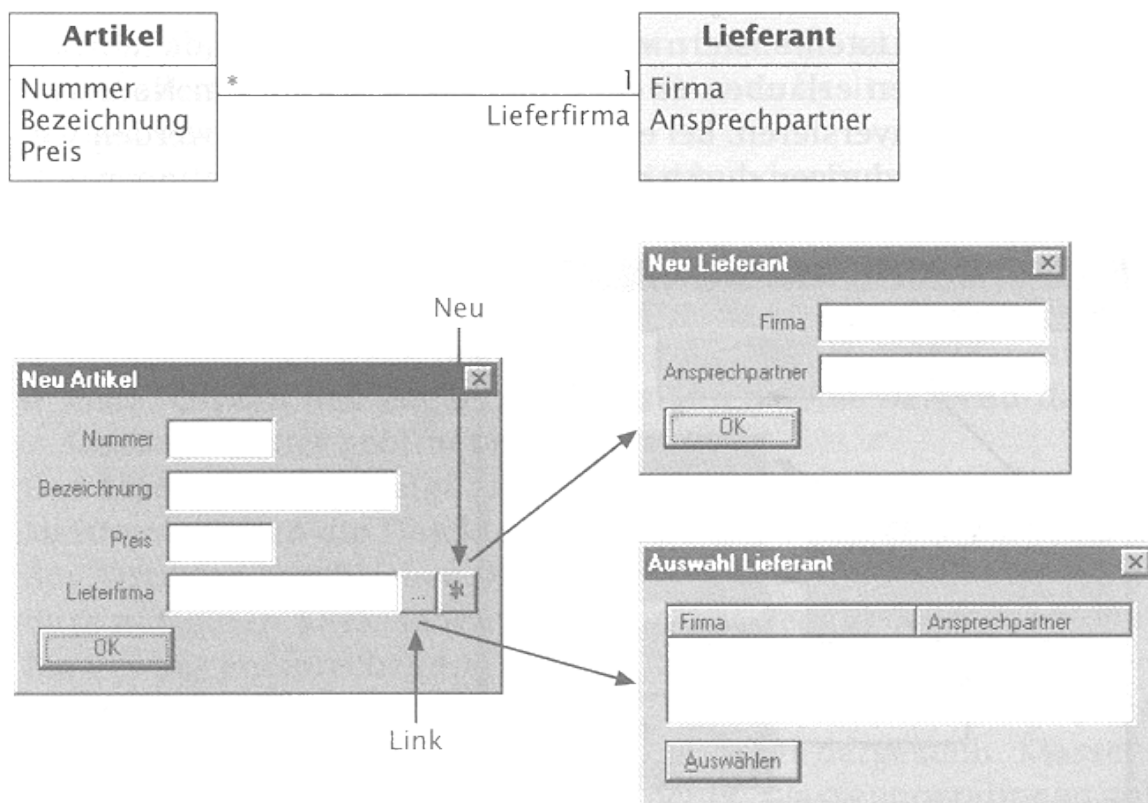
Jedes Fenster sollte über eine entsprechende Menüoption erreichbar sein. Der Benutzer kann jederzeit zwischen den Fenstern wechseln.

### **5..21. Assoziationen**

Assoziationen erlauben es dem Benutzer sich zwischen den unterschiedlichen Objekten einer Anwendung zu bewegen. Im OOA-Modell sind alle Assoziationen bidirektional, sie werden daher auch bidirektional auf den Prototyp abgebildet. Häufig ist es jedoch auch ausreichend, wenn einige Assoziationen nur in eine Richtung realisiert werden. In diesem Fall kann auch die entsprechende Dialogstruktur entsprechend gewählt werden. Das Erstellen und Entfernen einer Verbindung zu einem anderen Objekt wird in das Erfassungsfenster des jeweiligen Objektes integriert. Für jedes Erfassungsfenster muss zunächst dargestellt werden, zu welchen Klassen Verbindungen möglich sind, welche Verbindungen zwischen Objekten bereits bestehen oder eventuell aufgebaut werden können. Verbindungen zu anderen Objekten können sowohl aufgebaut als auch getrennt werden.

### **5..22. 1-Assoziation**

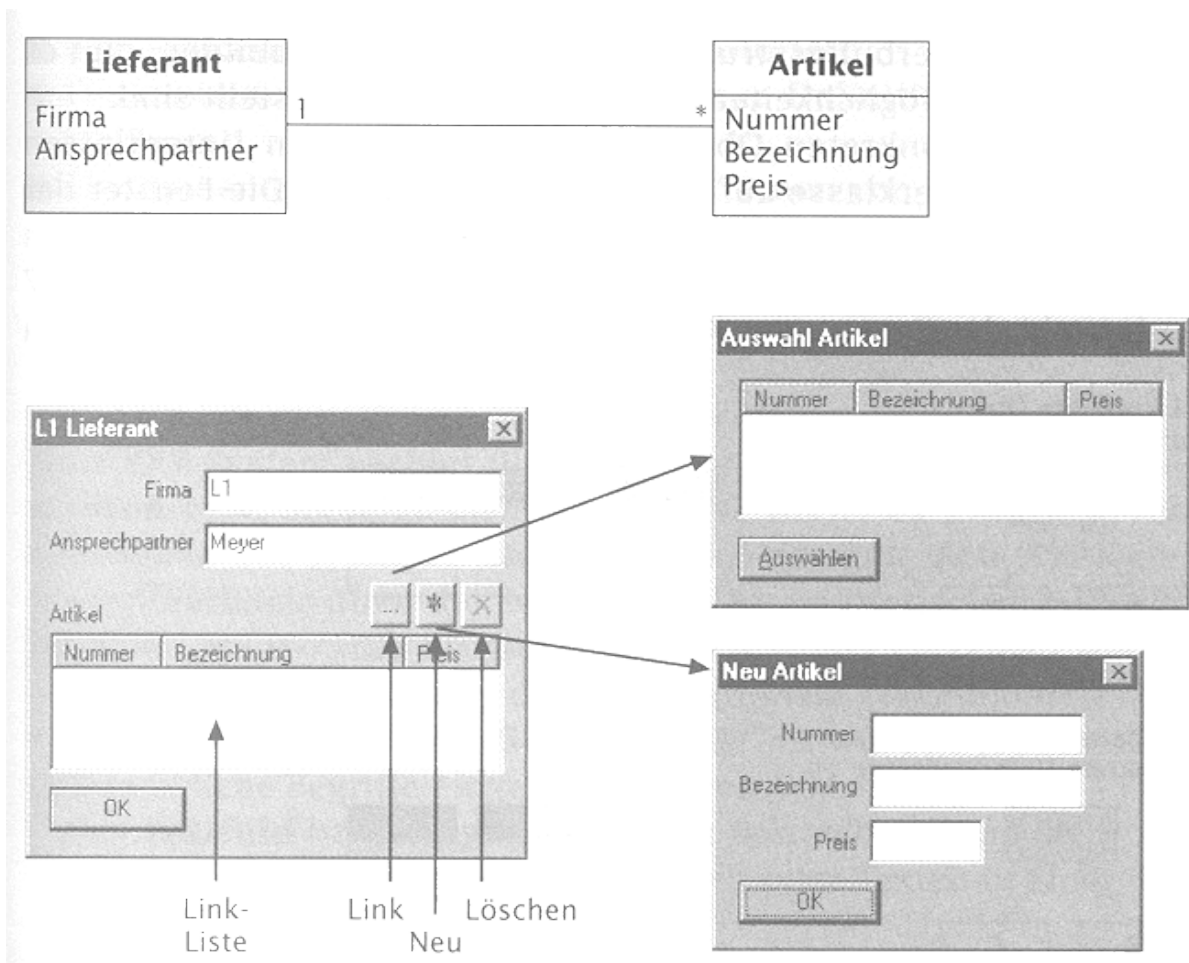
Im Erfassungsfenster in Abbildung 5.30 ist jeder Artikel mit genau einem Lieferanten verbunden. Diese Verbindung wird durch das Textfeld „Lieferfirma“ widergespiegelt. Bei einer Muss-Verbindung zwischen Artikel und Lieferant ist auch das Textfeld zwangsläufig ein Muss-Feld. Das bedeutet, dass das Erfassungsfenster erst geschlossen werden kann, wenn eine Verbindung zu einer Lieferfirma hergestellt wurde. Dafür muss dieses Textfeld zwei Schaltflächen zugeordnet bekommen. Die Neu-Schaltfläche (\*) öffnet ein leeres Erfassungsfenster des Lieferanten, in das ein neuer Lieferant eingetragen werden kann. Die Link-Schaltfläche (...) öffnet das Listenfenster für Lieferanten, aus denen dann ein Lieferant ausgewählt werden kann. Mit der Neueingabe bzw. der Auswahl des Lieferanten und dem Schließen des Lieferanten-Erfassungsfensters bzw. des Lieferanten-Listenfensters erfolgt automatisch die Zuordnung zwischen Artikel und ausgewähltem Lieferant.



**Abbildung 5.30 Darstellung einer 1-Assoziation**

### 5..23. many-Assoziation

Bei der häufiger vorkommenden many-Assoziation ist der Fall etwas schwieriger. Hier gehören zu einem Lieferanten mehrere Artikel. Daher muss das Erfassungsfenster des Lieferanten um eine Link-Liste der zugehörigen Artikel erweitert werden. Ebenso wie bei der 1-Assoziation sind die zusätzlichen Schaltflächen „Link“ (...) und „Neu“ (\*) enthalten. Die Funktionsweise dieser Schaltflächen ist analog zur oben beschriebenen Funktionalität. Hinzu kommt eine Schaltfläche (X) zum Löschen eines Artikels aus der Liste, mit der jedoch nur die Verbindung zwischen dem Lieferant und dem Artikel aufgehoben wird. Der Artikel selber verbleibt als Objekt im System. Bestehen mehrere many-Assoziationen, so können die zugehörigen Link-Listen auch auf Registerkarten untergebracht werden.



**Abbildung 5.31** Darstellung einer many-Assoziation

## 5..24. Vererbungsstrukturen in Dialogen

Ein Vererbungsstruktur kann dargestellt werden, indem an die Darstellung des Fensters der Oberklasse die zusätzlichen Attribute der Unterklasse in einem Bereich unterhalb der Attribute der Oberklasse angefügt werden. Die Darstellung der Oberklasse im Fenster sollte dabei in allen Unterklassen gleich sein.

Klasse A	Klasse B	Klasse C	Klasse D
Attribute der Klasse A	Attribute der Klasse B	Attribute der Klasse A	Attribute der Klasse A
		Attribute der Klasse C	Attribute der Klasse B
			Attribute der Klasse D

### Interaktionselemente

Jedes GUI-System verfügt über eine Menge von Interaktionselementen oder auch Steuerelementen (controls) genannt. Es gibt jedoch von GUI-System zu GUI-System spezifische Unterschiede. In Abbildung 5.3 sind die wichtigsten Interaktionselemente von Windows aufgeführt, deren Funktionsweise im Rahmen dieses Seminars als bekannt vorausgesetzt wird.


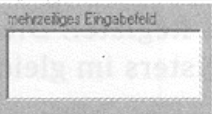
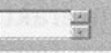
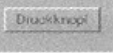
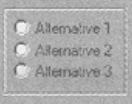

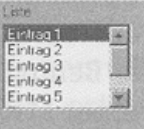
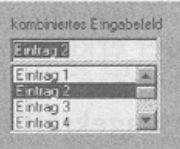

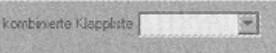
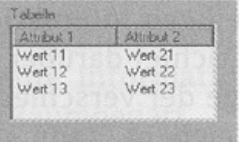

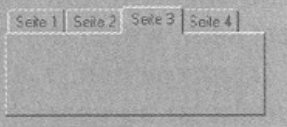
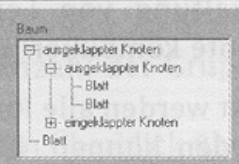
Bezeichnung	engl. Bezeichnung	Beispiel
Textfeld Eingabefeld	<i>text box, edit control</i>	Eingabefeld 
Mehrzeiliges Textfeld	<i>multi-line text box</i>	mehrzeiliges Eingabefeld 
Drehfeld	<i>spin box</i>	Drehhebel 
Schaltfläche, Druckknopf	<i>command button, push button</i>	Druckknopf 
Optionsfeld, Einfach- auswahlknopf	<i>option button, radio button</i>	
Kontrollkästchen, Mehrfach- auswahlknopf	<i>check box</i>	
Listenfeld, Auswahlliste	<i>list box</i>	Liste 
Kombinationsfeld	<i>combo box</i>	kombiniertes Eingabefeld 
Dropdown-Listenfeld, Klappliste	<i>drop-down list box</i>	Klappliste 
Dropdown- Kombinationsfeld	<i>drop-down combo box</i>	kombinierte Klappliste 
Listenelement, Tabelle	<i>list view control</i>	Tabelle 
Regler, Schieberegler	<i>slider</i>	Schieberegler 
Register	<i>tab control, property sheet</i>	
Strukturansicht, Baum	<i>tree view control</i>	Baum 

Abbildung 5.32 Interaktionselemente von Windows

## Gestaltung von Fenstern

Für die Gestaltung von Fenstern gibt es einige Regeln, die zu einer ergonomischeren Darstellung führen. Hierzu gehören insbesondere Hervorhebungen und Gruppierungen. Semantisch zusammengehö-

ge Elemente sollten gruppiert werden, um die Suchzeit des Benutzers im Fenster zu verringern. Der Benutzer orientiert sich zunächst an den Gruppen, dann an deren Inhalten.

### **5..25. Gestaltungsregeln für Fenster**

1. Informationen im oberen Bereich einer Gruppe werden schneller entdeckt als im unteren Bereich.
2. Die Elemente werden in der Gruppe so angeordnet, wie es der Arbeitsablauf des Benutzers erfordert.
3. Für das Suchen und Vergleichen von Elementen innerhalb einer Gruppe ist es günstiger, die Elemente spaltenweise statt zeilenweise anzuordnen.
4. Gruppenüberschriften erhöhen zwar die Übersichtlichkeit, sie vergrößern aber auch die Informationsmenge und den benötigten Raumbedarf im Fenster.
5. Eine Gruppe sollte nicht mehr als vier bis fünf Elemente enthalten, damit das gesuchte Element schneller identifiziert werden kann.
6. Bei fachlicher Notwendigkeit müssen natürlich größere Gruppen gebildet werden.
7. Die Anzahl der Gruppen sollte vier bis fünf nicht übersteigen, um einen besseren Überblick zu gewährleisten.
8. Wenn es nicht auf einen umfassenden Überblick über die Gruppen ankommt, kann die Zahl der Gruppen auf bis zu 15 erhöht werden.
9. Gruppen sollten deutlich voneinander getrennt sein.

### **5..26. Gestaltungsregeln für das Hervorheben**

Manche Bereiche eines Fensters sollten die Aufmerksamkeit des Benutzers auf sich ziehen. Dies kann durch entsprechende Hervorhebungen erreicht werden. Prinzipiell kann dies durch Größe, Farbe/Kontrast, Isolierung/Einzelstellung oder Umrandungen geschehen. Dabei sollten folgende Regel beachtet werden:

1. Maximal 10 – 20 Prozent aller Informationen hervorheben.
2. Farben sparsam einsetzen, maximal fünf Farben.
3. Verschiedene Arten von Hervorhebungen nur in geringem Umfang einsetzen.
4. Hervorhebungen durch blinken vermeiden.

### **5..27. Gestaltungsregeln für die Farbwahl**

Farben können den Benutzer visuell bei der Bearbeitung eines Fensters unterstützen. Ein Fenster sollte aber nicht bunt sein, sondern durch Hervorhebungen mit Farbe dezent die Aufmerksamkeit des Benutzers auf sich lenken. Hierbei sollten folgende Regeln eingehalten werden:

1. Gestaltung der Fenster nicht bunt, sondern farbig.
2. Vor dunklem Hintergrund eignen sich die Farben Weiß, Gelb, Cyan und Grün am besten, vor hellem Hintergrund die Farben Magenta, Rot, Blau, Schwarz.
3. Unterschiedliche Farben nur sparsam einsetzen, eine Farbtonabstufung ist besser.
4. Über die Fenster eines Projektes hinweg nicht mehr als sieben Farben einsetzen. Ausnahmen bilden graduelle Abstufungen des Farbtons.
5. Farben konsistent verwenden.
6. Konventionelle Farbcodierungen sollen beibehalten werden: Rot für Gefahr / Halt; Grün für weiter, sicher; Gelb für Vorsicht; Blau für kalt und Beruhigung.
7. Farbtonunterschiede im Rot- und Purpurbereich sind schwerer zu erkennen als im Gelb- und Blaubereich.

### **5..28. harmonische Gestaltung**

Auch die ästhetische Gestaltung von Fenstern trägt mit dazu bei, den Benutzer seine Arbeit zu erleichtern. Diesbezüglich ist folgenden Aspekten Rechnung zu tragen.

#### **Proportionen**

Einem Betrachter erscheint eine Fläche angenehmer, wenn das Verhältnis von Breite zu Höhe 1:1 bis 1:2 entspricht. Dies kann durch entsprechende Verteilung der Informationen in dem Fenster erreicht werden.

#### **Balance**

Wenn man ein Fenster durch eine vertikale Linie teilt, sollten beide Seiten der Linie die gleiche Informationsdichte aufweisen.

#### **Symmetrie**

Die Symmetrie verstärkt die Balance. Zusätzlich zur Balance wird für die Symmetrie gefordert, dass die horizontal gegenüberliegenden Interaktionselemente gleichartig und gleich groß sein sollen.

The 'Hilfsassistent' dialog box features a balanced layout. On the left, the 'Studienadresse' group box contains fields for 'Straße', 'PLZ', 'Ort', and 'Telefon'. Above it are 'Name' and 'Vorname' fields, and at the bottom is the 'Geburtsdatum' field. On the right, the 'Heimadresse' group box contains fields for 'Straße', 'PLZ', 'Ort', and 'Telefon'. Above it is the 'Matrikel-Nr.' field, and below it is the 'Datum Arbeitszeugnis' field. A checkbox for 'Arbeitszeugnis' is positioned between the two address sections. At the bottom, there are four buttons: 'OK', 'Übernehmen', 'Abbrechen', and 'Liste...'.

The 'Hilfsassistent' dialog box features a symmetrical layout. The 'Studienadresse' group box on the left contains fields for 'Straße', 'PLZ', 'Ort', and 'Telefon'. Above it are 'Name' and 'Vorname' fields, and at the bottom is the 'Geburtsdatum' field. The 'Heimadresse' group box on the right contains fields for 'Straße', 'PLZ', 'Ort', and 'Telefon'. Above it is the 'Matrikel-Nr.' field, and below it is the 'Datum Arbeitszeugnis' field. A checkbox for 'Arbeitszeugnis' is positioned between the two address sections. At the bottom, there are four buttons: 'OK', 'Übernehmen', 'Abbrechen', and 'Liste...'.

**Abbildung 5.33**      **Balanciertes Fenster und symmetrisches Fenster**

## Sequenz

Das Auge des Benutzers soll sequenziell d. h. entsprechend dem Ablauf der Bearbeitungsschritte durch das Fenster geführt werden. Unnötige Sprünge müssen vermieden werden. Die wichtigsten Informationen eines Fensters sollten immer links oben zu finden sein.

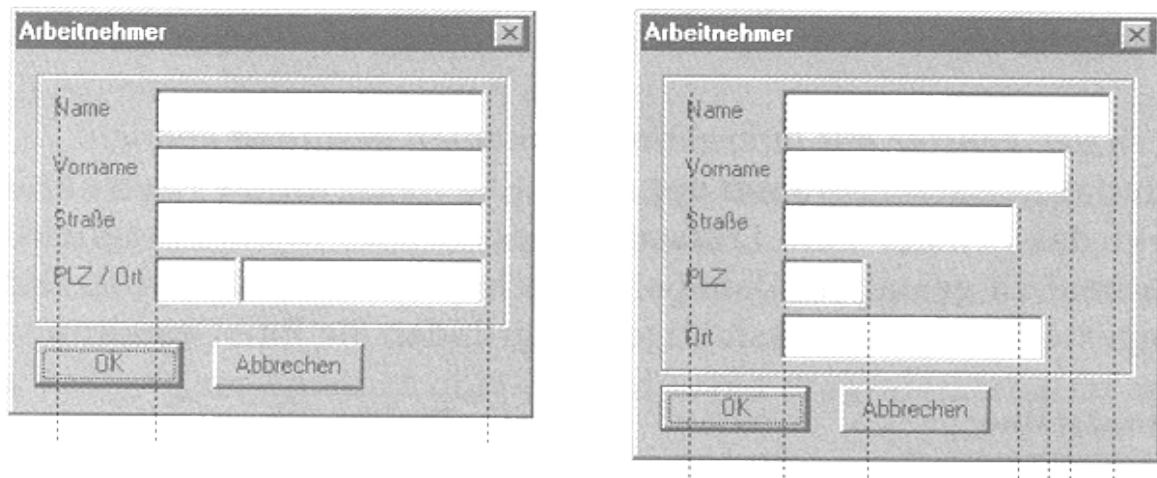
## Einfachheit

Ein Fenster sollte so einfach wie möglich gestaltet werden. Hierzu sollte eine einheitliche Schriftart zum Einsatz kommen und Farben sehr zurückhaltend eingesetzt werden.



## Virtuelle Linien

Neben den sichtbaren Linien eines Fensters wie den Kanten von Interaktionselementen, Rahmen, Kanten der Schaltflächen u. ä. wird das Aussehen eines Fensters auch durch die virtuellen Linien geprägt. Hierunter sind die Verlängerungen der Kanten der Steuerelemente eines Fenster zu verstehen. Je mehr virtuelle Linien in einem Fenster enthalten sind, desto unruhiger ist sein Erscheinungsbild und desto ungünstiger wirkt es sich auf den Benutzer aus. Ziel sollte daher sein, durch entsprechende Fenstergestaltung die Anzahl der virtuellen Linien so gering wie möglich zu halten.



**Abbildung 5.34** Fenster mit wenigen und vielen virtuellen Linien

## 6. Der objektorientierte Entwurf (Design)

Wurde bei der OOA von einer idealen Umgebung ausgegangen, so ist es nun die Aufgabe des Entwurfs, die spezifizierten Anforderungen unter Berücksichtigung der Plattform zu realisieren. Hierbei müssen nun auch die technischen Randbedingungen mit in die Überlegungen einfließen. Allerdings findet der objektorientierte Entwurf (OOD) noch auf einer höheren Abstraktionsebene statt, als die spätere Implementation. Das OOD-Modell wird unter den Gesichtspunkten der Effizienz und der Standardisierung entwickelt. Entwurfs- und Implementierungsphase sind hierbei so stark verzahnt, dass ein direktes Implementieren einer Klasse aus dem Modell heraus möglich ist.

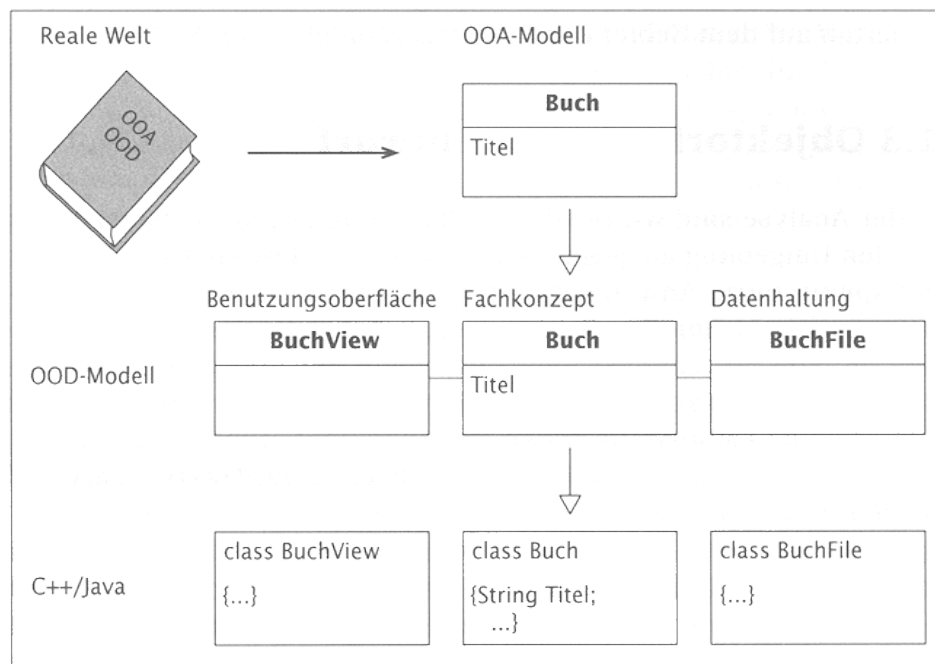
### Entwurfsziel

Betrachtet man heute ältere Systeme, stellt man fest, dass deren Funktionalität zum Teil noch „modern“ ist. Bezüglich der Benutzungsoberfläche und der Datenhaltung sind sie jedoch veraltet. Zur Aktualisierung dieser Systeme auf die neuesten Standards wäre eine Neuprogrammierung notwendig. Daher verfolgt das OOD eine Trennung des Entwurfs in ein Drei-Schichten-Modell. Hierbei wird das Fachkonzept, die Benutzungsschicht und die Datenhaltungsschicht weitestgehend entkoppelt. Das Aussehen der Benutzungsoberfläche hängt weitgehend von dem verwendeten GUI ab, wohingegen die Datenhaltung entscheidend von der verwendeten Datenbank bestimmt wird.

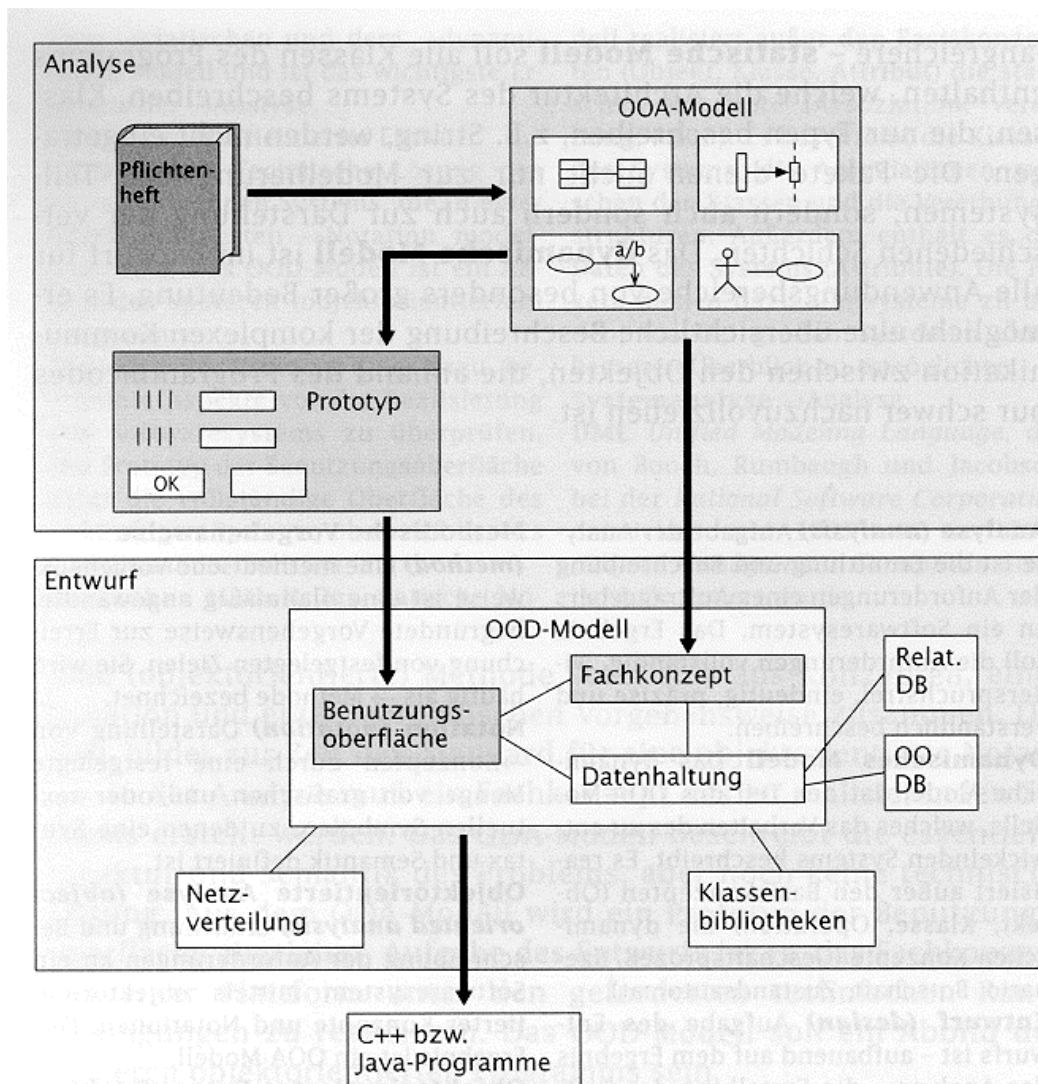
Aus dem Entwurfsziel lässt sich nun direkt diese Drei-Schichten-Architektur ableiten. D. h. es wird getrennt zwischen der Fachkonzeptschicht, der Benutzungsoberflächenschicht und der Datenschicht.

Aus dem Ergebnis der Analyse, dem OOA-Modell, kann nun die erste Version der Fachkonzeptschicht erstellt werden. Das Modell wird hierfür unter dem Gesichtspunkt der Effizienz und der Wiederverwendbarkeit überarbeitet. Der Prototyp der Analysephase bildet die Grundlage für die Schicht der Benutzungsoberfläche. Die Datenhaltungsschicht bildet die Verbindung zur Datenbank.

Im Gegensatz zum OOA-Modell, das keine technischen Lösungen liefert, soll das OOD-Modell ein Abbild des späteren Programms sein. Dabei findet sich jede Klasse, jedes Attribut und jede Operation, die innerhalb des OOD-Modells entwickelt wird später im Programm mit exakt dem gleichen Namen wieder. Jedoch ist das OOD-Modell abstrakter als die Implementierung und macht das Zusammenwirken der einzelnen Komponenten deutlich. Auch bei dem OOD-Modell arbeitet man – wie beim OOA-Modell – mit statischen und dynamischen Modellen. Diese sind aber gegenüber der Analyse wesentlich umfangreicher.



**Abbildung 6.35**    **Drei-Schichten-Architektur**



**Abbildung 6.36 Abgrenzung Analyse und Entwurf**

### Das statische Modell des OOD

Im statischen Modell des OOD sind alle Klassen enthalten, die das System beschreiben. Soweit Klassen nur Datentypen enthalten (z. B. Strings) werden sie nicht extra beschrieben.

Pakete werden nicht nur zur Modellierung von Teilsystemen genutzt, sondern dienen auch zur Darstellung verschiedener Schichten.

### Das dynamische Modell des OOD

Das dynamische Modell des OOD ist beim Entwurf in allen Anwendungsbereichen von Software-Programmen besonders wichtig. Hier spiegelt sich wesentlich klarer als im späteren Programmcode die komplexe Kommunikation zwischen den einzelnen Objekten wider. Dieses Zusammenspiel ist im Programmcode nur sehr schwer nachvollziehbar.

### Konzept und Notation des objektorientierten Entwurfs

Im folgenden werden nun einige Erweiterungen der Konzepte der objektorientierten Analyse behandelt, die für das Konzept des objekto-

rientierten Entwurfs benötigt werden. Das OOD-Modell soll ein Spiegelbild des Programms sein. Alle Namen des objektorientierten Entwurfs müssen sich daher an die Syntax der verwendeten Programmiersprache halten. In diesem Script wird hierfür beispielhaft die Programmiersprache C++ verwendet.

### 6..1. Objekte / Klassen

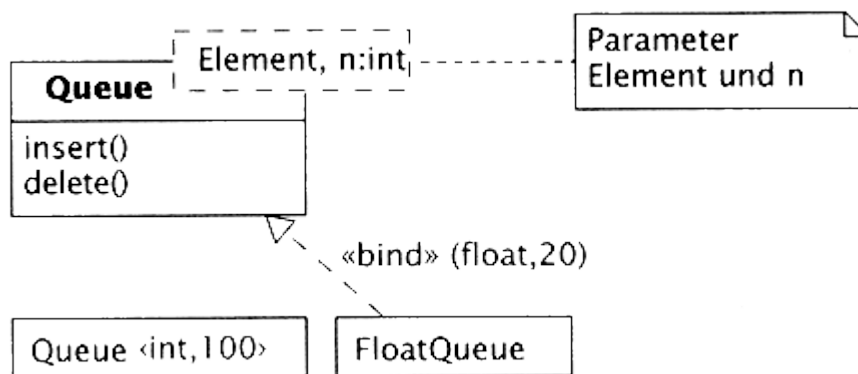
Die Notation der OOA gilt auch für den Entwurf. Allerdings muss der Name einer Klasse im Entwurf nun der Syntax der verwendeten Programmiersprache folgen. Klassennamen beginnen immer mit einem Großbuchstaben. Innerhalb der OOA ist die verwendete Sprache zunächst ohne Bedeutung, im OOD ist es üblich die englische Sprache zu verwenden. Es können Stereotypen verwendet werden, um die Zugehörigkeit einer Klasse zu einer bestimmten Entwurfskomponente zu zeigen. So können z. B. alle Klassen zur Datenverwaltung mit DB (data base) oder alle Klassen zur Benutzungsoberfläche mit UI (user interface) beginnen. Eine wesentliche Erweiterung zum Konzept der Analyse stellen die im folgenden vorgestellten generischen Klassen und Containerklassen dar.

#### Generische Klassen

*Eine generische (parameterized class, template) Klasse ist eine Beschreibung einer Klasse mit einem oder mehreren formalen Parametern, die eine Familie von Klassen definiert.*

Die Parameter einer solchen Klasse setzen sich aus dem Namen des Parameters und seinem Datentyp zusammen, wobei der Datentyp entfällt, wenn der Name bereits ein Typ ist. Es gibt keine leere Parameterliste. Um eine solche Klasse zu nutzen, müssen ihre formalen Parameter an aktuelle Parameter gebunden werden.

Stellen wir uns eine generische Klasse Queue vor, wie sie in Abbildung 6.37 gezeigt wird. Sie besitzt zwei Operationen insert() und delete(). Zunächst ist noch nicht festgelegt, welche und wie viel Objekte diese Klassen verwalten soll. Da der Parameter Element einen Typ beschreibt, sind hier keine weiteren Angaben erforderlich. Der Parameter n vom Typ int gibt die maximale Größe der Queue an. Diese generische Klasse bildet die Vorlage für die normalen Klassen Queue<int, 100>, in der maximal 100 int-Werte gespeichert werden können, und Floatqueue, die maximale 20 Werte vom Typ float aufnimmt. Außerdem zeigt die Abbildung 6.37 wie die Bindung der Klassen an die generische Klasse erfolgt. Zum einen kann die Klasse an die generische Klasse gebunden werden, indem die aktuellen Parameter direkt bei der Klasse mit eingetragen werden (wie bei Queue<int, 100>) oder durch Verwendung des Stereotypen <<bind>>, wie im Fall von FloatQueue.



**Abbildung 6.37 Generische Klasse Queue**

Eine generische Klasse kann nicht Oberklasse einer normalen Klasse sein. Sie kann jedoch selber eine andere Klasse als Oberklasse (Super) haben. In dem Fall sind alle mit binding gebildeten normalen Klassen Unterklassen von Super.

Außerdem kann eine unidirektionale Assoziation von einer generischen Klasse zu einer Unterklasse bestehen. Hierbei „kennt“ die generische Klasse zwar die normale Klasse, nicht aber umgekehrt. Die unidirektionale Assoziation wird weiter unten noch ausführlicher besprochen.

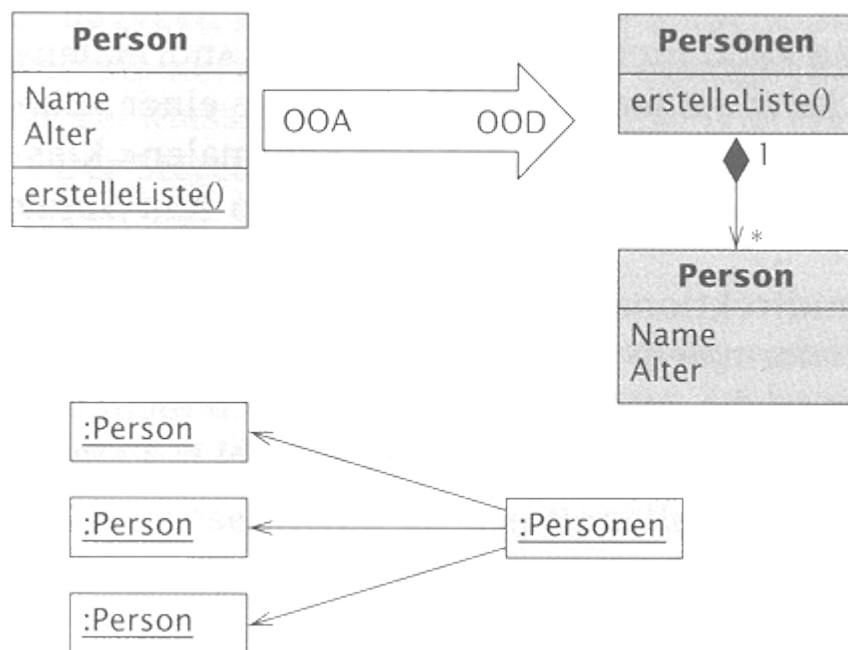
## Container-Klassen

Bei der objektorientierten Analyse sollen keine Klassen gebildet werden, die Mengen von Objekten verwalten, da in der Analysephase bei Klassen von der Fähigkeit zu einer solchen Objektverwaltung ausgegangen wird. Im Entwurf hingegen muss nun für eine solche Verwaltung gesorgt werden. Hierfür lassen sich Container-Klassen einsetzen. Viele Bibliotheken stellen solche Container-Klassen bereits zur Verfügung. Häufig werden generische Klassen eingesetzt.

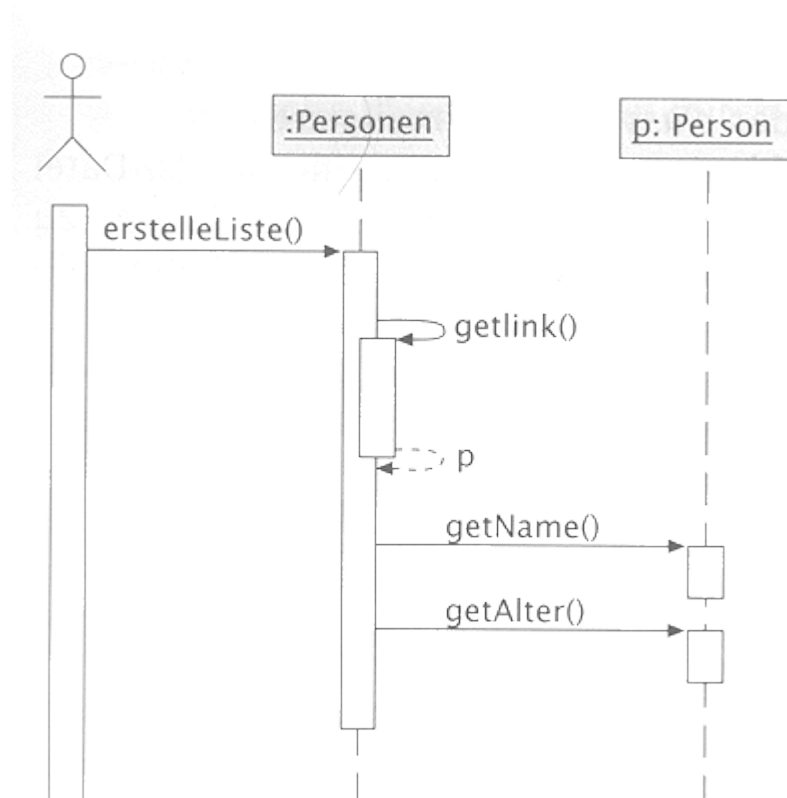
*Eine Container-Klasse ist eine Klasse, die dazu dient, eine Menge von Objekten einer anderen Klasse zu verwalten. Sie stellt Operationen bereit, um auf die verwalteten Objekte zuzugreifen. Ein Objekt einer Container-Klasse wird als Container bezeichnet.*

Typische Container sind beispielsweise Felder bzw. Arrays.

In Abbildung 6.38 werden alle Objekte der Klasse Person im Container Personen verwaltet. Personen kennt also alle Objekte und damit die ObjektID's der Klasse Person. Aus der Klassenoperation erstelleListe() wird eine Objektoperation der Klasse Personen, die sich mit getLink() (Abbildung 6.39) die OID der Objekte der Klasse Person holt und auf die Daten der jeweiligen Objekte zugreift.



**Abbildung 6.38** Container für die Verwaltung von Personen



**Abbildung 6.39** Sequenzdiagramm für die Verwaltung von Personen

## Schnittstellen

*Eine Schnittstelle spezifiziert einen Ausschnitt aus dem Verhalten einer Klasse. Eine Schnittstelle besteht nur aus den Signaturen der Operationen einer Klasse. Sie besitzt also keine Implementierung von Quellcode, keine Attribute, Zustände oder Assoziationen.*

Eine Schnittstelle ist äquivalent zu einer abstrakten Klasse, die ausschließlich abstrakte Operationen besitzt. Die Realisierung einer Schnittstelle durch eine Klasse ist durch einen gestrichelten Vererbungspfeil gekennzeichnet.

## Syntax in C++

### Klasse

Eine Klasse wird in C++ mit dem Schlüsselwort `class` realisiert. Üblicherweise steht die Spezifikation einer Klasse in einer Headerdatei (.h). Die Implementierung erfolgt in einer Quelldatei (.cpp). Eine Ausnahme bilden hier Templates, deren Deklaration und Implementierung in einer Headerdatei stehen müssen.

```
// zaehler.h
class Zaehler
{private:
    unsigned int Zaehlerstand;
public:
    void inkremmentieren();
    void initialisieren();
    int getZaehlerstand() const;
};

//zaehler.cpp
void Zaehler::inkremmentieren() {...}
void Zaehler::initialisieren() {...}
int Zaehler::getZaehlerstand() const {...}
```

### 6.4.1.4.2 Objekt

Objekte werden wie normale Variablen behandelt. Sie können entweder statisch im Datasegment deklariert oder auf dem stack angelegt sein. Des weiteren gibt es die Möglichkeit, sie dynamisch (heap) zu erzeugen. Für dynamisch erzeugte Variablen muss der Speicherplatz explizit wieder freigegeben werden.



```
Zaehler  einZaehler;      //Datasegment oder stack-Variable
Zaehler* pZaehler;        //Zeiger-Variable
pZaehler = new Zaehler;    //heap-Variable wird erzeugt
```

### Abstrakte Klasse

Eine direkte Kennzeichnung abstrakter Klassen erfolgt in C++ nicht. In C++ wird eine Klasse dadurch zur abstrakten Klasse, dass mindestens eine Operation als abstrakt deklariert ist.

```
class abstract
{public:
void operation() = 0;
};
```

### Generische Klasse

Generische Klassen werden durch templates realisiert.

```
template<class ElementType, int max>
class Queue
{protected:
struct ListElement
    {ElementType  value;
      Listelement* next;
    };
public:
void insert(ElementType element);
};

typedef Queue<int, 100> IntergerQueue;
IntergerQueue aQueue;
aQueue.insert(5);
```

### Schnittstelle

Das Konzept der Schnittstelle wird von C++ nicht unterstützt.

## 6..2. Attribute

Auch die Attributnamen müssen an die Konventionen der Programmiersprache angepasst werden. Die UML fordert hier Kleinbuchstaben. In der Systemanalyse waren alle Attribute einer Klasse nach außen nicht sichtbar. Nur die Unterklassen konnten die Attribute „sehen“. Dieser Modus wird als protected bezeichnet. Im Entwurf ist eine Differenzierung der Sichtbarkeit von Attributen notwendig. Es gibt drei Sichtbarkeits-Modi (visibility).

- public: sichtbar für alle Klassen
- protected: sichtbar in der eigenen Klasse und in den Unterklassen

- **private:** nur innerhalb der Klasse sichtbar Die Notation der UML sieht folgende Schreibweisen vor:

	class
public	+ Attributname
protected	# Attributname
private	- Attributname

Bei public wird das Geheimnisprinzip verletzt, weswegen dieser Sichtbarkeits-Modus nur in Ausnahmefällen verwendet werden sollte. Der Sichtbarkeits-Modus protected erlaubt den direkten Zugriff auf die Attribute der Oberklasse, hat aber den Nachteil, dass sich Änderungen der Attribute der Oberklasse direkt auf die Unterklasse auswirken. Der Sichtbarkeits-Modus private schützt die Attribute der Oberklasse vor dem Zugriff der Unterklassen, so dass die Unterklassen nur noch über entsprechende Operationen Zugriff auf die Attribute der Oberklasse haben, was jedoch einen höheren Programmieraufwand zur Folge hat.

Die allgemeine Notation lautet:

Sichtbarkeit Attribut: Typ = Anfangswert {Merkmalsliste}

Bezüglich der verwendeten Typen muss sich nach der jeweiligen Programmiersprache gerichtet werden. Hierbei kann auf die Standardtypen sowie auf die Elementarklassen der Programmiersprachen zurückgegriffen werden.

Die optionale Merkmalsliste nimmt eventuell benötigte Eigenschaften der Attribute auf (z. B. mandatory, frozen).

Die Transformation einer Klasse Kreis aus der OOA in das OOD würde z. B. wie folgt aussehen:



An dieser Stelle sollen nochmals drei Begriffe gegenübergestellt werden, die häufig zu Verwirrung führen:

- **Abstraktion:**

*Abstraktion als ein Prozeß betrachtet, bezeichnet die Vorgehensweise, die wesentlichen Aspekten über etwas zu ermitteln und die unwesentlichen Details zu ignorieren. Auch ein Modell oder ein bestimmter Blickwinkel wird als Abstraktion betrachtet.*

- **Geheimnisprinzip:**

*Das Geheimnisprinzip bedeutet, dass der Zustand eines Objektes und die Implementierung der Operationen außerhalb der Klasse nicht sichtbar sind.*

➤ Verkapselung:

*Die Verkapselung sagt aus, dass zusammengehörende Attribute und Operationen in einer Einheit – der Klasse – zusammengefasst sind. Im Unterschied zum Geheimnisprinzip können Attribute und Operationen durchaus nach außen hin sichtbar sein. Die Programmiersprache C++ erlaubt über o. g. Sichtbarkeits-Modi eine Verkapselung ohne und mit Einhaltung des Geheimnisprinzips.*

## Syntax in C++

### Sichtbarkeit

Die Sichtbarkeit von Attributen ist in C++ genauso definiert wie in der UML. Das jeweilige Schlüsselwort wird dem Abschnitt der zu deklarierenden Attribute oder Operationen vorangestellt, gefolgt von einem Doppelpunkt. Die Sichtbarkeit bleibt erhalten bis eine neue Sichtbarkeit für einen Abschnitt angegeben wird. Standardmäßig verfügen alle Attribute und Operationen einer Klasse über die Sichtbarkeit `private`.

### Initialisierung von Attributen

Im Konstruktor (wird aufgerufen beim Erzeugen eines Objektes)

```
Kreis::Kreis()
{
    istSichtbar = true;
}
```

### Klassenattribute

```
class Kreis
{protected:
    bool istSichtbar;
    static unsigned int Anzahl; // Klassenattribut da static
};

unsigned int Kreis::Anzahl = 0; // Deklaration und Initialisierung
```

## 6..3. Operationen

### Sichtbarkeit

Für die Sichtbarkeit von Operationen gelten die gleichen Regeln wie für die Sichtbarkeit von Attributen.

## Signatur

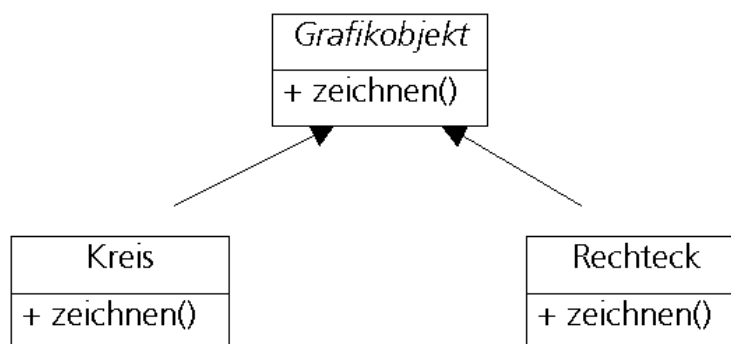
Im Entwurf ist die vollständige Signatur einer Operation anzugeben. Die Signatur einer Operation besteht aus den Namen der Operation, den Namen und Typen aller Parameter und dem Ergebnistyp der Operation. Die Menge aller Signaturen, die von den Operationen einer Klasse definiert werden, nennt man die Schnittstelle einer Klasse.

## Implementierung einer Operation

Da durch die objektorientierte Modellierung der Anwendungen im allgemeinen bei der Implementierung kompaktere Operationen entstehen, erübrigt es sich meistens, die Wirkung einer Operation im Entwurfsmodell separat zu dokumentieren. Sie kann statt dessen sofort in der entsprechenden Programmiersprache implementiert und dort ausreichend kommentiert werden. Bei Bedarf kann hier auch eine Beschreibung mittels Vor- und Nachbedingung erfolgen. Die Vorbedingung beschreibt, welche Bedingungen vor dem Aktivieren der Operation erfüllt sein müssen, die Nachbedingung beschreibt die Änderungen, die die Operation bewirkt.

## abstrakte Operationen

Eine abstrakte Operation besteht nur aus der Signatur der Operation. Sie besitzt keine Implementierung. Mit abstrakten Operationen können gemeinsame Schnittstellen für die Unterklassen definiert werden. In der UML werden abstrakte Operationen kursiv eingetragen. Bei handschriftlicher Notierung wird die Operation mit {abstrakt} gekennzeichnet.



Im Entwurf muss außerdem die Parameterliste der Operation angegeben werden. Diese Liste enthält formale Parameter, die durch Komma getrennt sind. Es gilt folgende Syntax:

Art Name : Typ = Anfangswert oder Art Name : Typ.

Die Art gibt an, ob es sich um einen Eingabe- (in), einen Ausgabe- (out) oder eine transienten (inout) Parameter handelt.

Der Ergebnistyp beschreibt den Typ des Ergebnisparameters. Wenn der Typ fehlt, gibt die Operation keinen Wert zurück.

In einer optionalen Merkmalsliste können spezielle Eigenschaften der Operation angegeben werden.

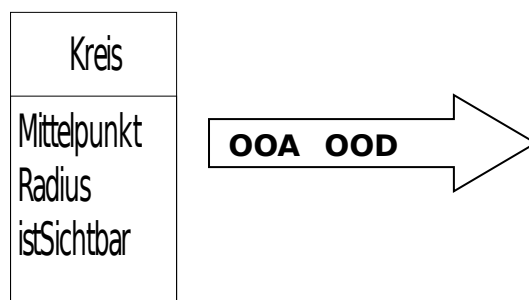
Damit kann die obige Syntax wie folgt vervollständigt werden:

Sichtbarkeit Operation (Parameterliste) : Ergebnistyp {Merkmalsliste}

Für jeden Parameter der Parameterliste gilt:

[in | out | inout] Name: Typ = Anfangswert

Wobei Name für den formalen Parameter steht. Mehrere Parameter werden durch Komma getrennt. Die UML erlaubt es, auf die Angabe der Parameterliste und des Ergebnistyps zu verzichten, wodurch sich kompaktere Klassendiagramme ergeben. Außerdem erlaubt die UML die Notation in einer Programmiersprache zu spezifizieren.



### overloading

Bei der Analyse müssen alle Operationsnamen eindeutig sein. Im Entwurf dürfen hingegen Operationsnamen mehrfach verwendet werden. Hierbei müssen sich jedoch die Parameterlisten der Operationen unterscheiden. Man spricht in diesem Fall vom Überladen (overloading).

## Syntax in C++

### Sichtbarkeit

```
class Kreis
{protected:
    Punkt Mittelpunkt;
    unsigned int Radius;
    bool istSichbar;
    static unsigned int Anzahl; //Klassenattribut
protected:
    void loeschen();
public:
    Kreis(); //Konstruktor
    Kreis(int x, int y); //Konstruktor
    void zeichnen();
    void verschieben(Punkt neu);
    void vergroessern(int faktor);
    void verkleinern(int faktor);
    static unsigned int getAnzahl(); //Klassenoperation
};
```

### Parameterkonzept

In C++ können Eingabeparameter by value oder by reference an Operationen übergeben werden. Ausgabeparameter können entweder als call by reference oder als Ergebnistypen realisiert werden.

### Abstrakte Operationen

Abstrakte Operationen in C++ werden als pure virtual member functions realisiert (nicht zu verwechseln mit virtual functions!).

```
void zeichnen() = 0;
```

### frozen-Operationen

Operationen, bei denen die Attributwerte nicht verändert, sondern nur gelesen werden, werden als const deklariert.

```
bool getIstSichtbar() const;
```

### Klassenoperationen

Analog zu den Klassenattributen werden Klassenoperationen ebenfalls mit static deklariert. Beim Aufruf einer Klassenoperation wird der Klassenname vorangestellt.

```
static unsigned int getAnzahl();           Klassenoperation
i = Kreis::getAnzahl();
```

## Konstrukoren und Destruktoren

Wenn ein Objekt einer Klassen erzeugt wird, wird ein Konstruktor der Klasse aufgerufen. Der Konstruktor dient zur Initialisierung der Attribute und zur Reservierung von Speicherplatz, falls für ein Klassenattribut dieser benötigt wird. Eine Klasse kann mehrere Konstrukturen besitzen. Diese müssen sich aber alle wie beim Überladen in ihrer Parameterliste unterscheiden. Wird einer Klasse nicht explizit einem Konstruktor zugeordnet, wird implizit ein Konstruktor erstellt.

Wird ein Objekt wieder gelöscht, wird der Destruktor der Klasse aufgerufen. In ihm müssen eventuell im Konstruktor vorgenommene Speicherplatzreservierungen wieder aufgehoben d. h. der Speicher wieder freigegeben werden. Auch ein Destruktor wird implizit erstellt, wenn der Klasse kein Destruktor explizit zugeordnet wurde.

```
class Kreis
{protected:
bool istSichtbar;
public:
Kreis() {istSichtbar = false;}
Kreis(int x, int y)
    {
    istSichtbar = false;
    Mittelpunkt.setX(x);
    Mittelpunkt.setY(y);
    }
};

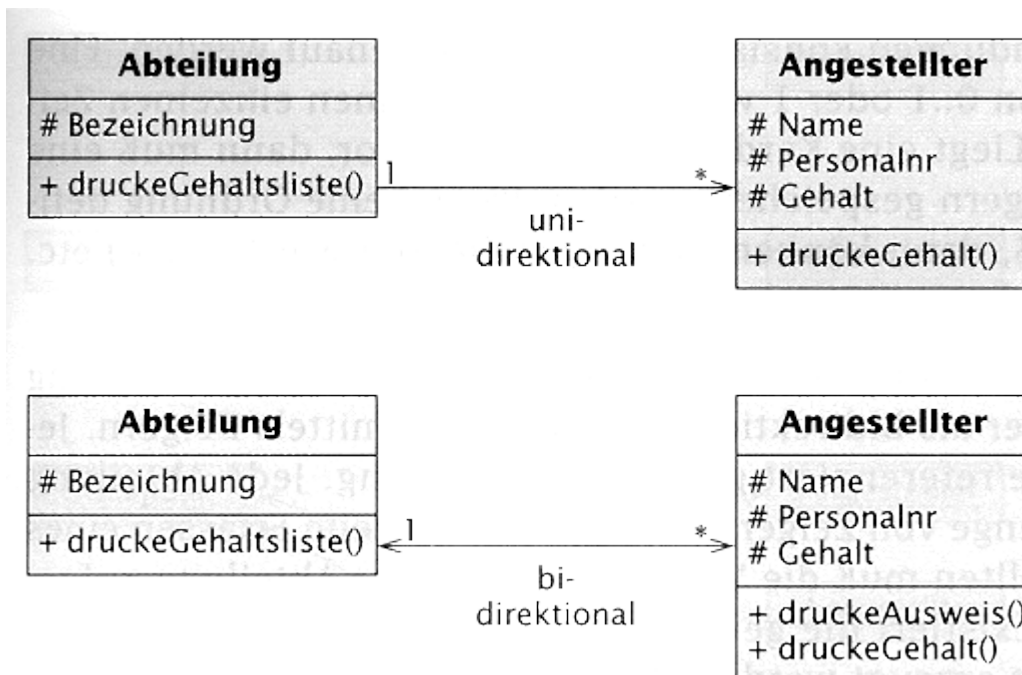
Kreis    einKreis;
```

## 6..4. Assoziation

Während in der Analyse alle Assoziationen als bidirektional angenommen werden, wird im Entwurf nun festgelegt, ob sie uni- oder bidirektional implementiert werden müssen. Dies wird als die Navigation der Assoziation bezeichnet. In der muss im oberen Modell nur von einer Abteilung auf ihre Angestellten zugegriffen werden. Die Assoziation muss also nur in eine Richtung sprich unidirektional implementiert werden. Im unteren Modell sollen auch die Angestellten auf ihre zugehörige Abteilung zugreifen können, um z. B. die Daten für das Ausdrucken eines Ausweises zu ermitteln. In dem Fall ist eine bidirektionale Implementierung der Assoziation erforderlich.

In der UML werden diese beiden Assoziationstypen durch eine Verbindungslinie gekennzeichnet. Bei einer unidirektionalen Assoziation befindet sich an dem Ende ein Pfeil, in dessen Richtung ein Zugriff

auf die Daten möglich sein soll. Zur Darstellung einer bidirektionalen Assoziation erhält die Linie entweder in jede Richtung einen Pfeil oder gar keine Pfeile.



**Abbildung 6.40 Unidirektionale und bidirektionale Assoziation**

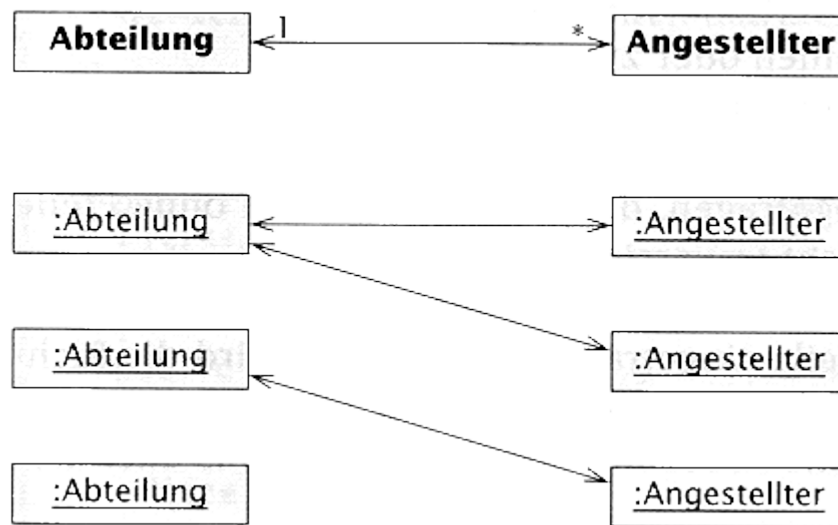
Die Kennzeichnung aller im Programm realisierten Navigationsrichtungen ist im Entwurf erforderlich, um eine spätere Wartung des Programms zu erleichtern.

Die Implementierung einer bidirektionalen Assoziation ist weitaus komplexer als die einer unidirektionalen Assoziation. So muss zum Beispiel beim Löschen eines Objektes bei einer bidirektionalen Assoziation darauf geachtet werden, dass auch der Zeiger darauf im assoziierten Objekt entfernt wird.

Eine Assoziation kann mittels Zeiger zwischen den Objekten realisiert werden. Dabei muss auf einen konsistenten Auf- und Abbau aller Verbindungen geachtet werden. Eine Kardinalität von 0..1 oder 1 lässt sich durch einen einzigen Zeiger verwirklichen. Größere Kardinalitäten müssen über eine Menge von Zeigern realisiert werden. Hierfür können z. B. Container-Klassen verwendet werden.

In dem obigen Beispiel der bidirektionalen Assoziation referenziert jeder Angestellter genau eine Abteilung. Jede Abteilung hingegen referenziert eine Menge von Angestellten. Wird ein neuer Angestellter erfasst, muss eine entsprechende Verbindung vom Angestellten zu „seiner“ Abteilung aufgebaut werden. Für den Fall, dass die gewünschte Abteilung noch nicht existiert, muss diese zunächst erstellt werden. In der Abteilung muss im Gegenzug ein Zeiger auf den neuen Angestellten gespeichert werden. Scheidet ein Angestellter aus dem Unternehmen aus, muss nicht nur das entsprechende Objekt entfernt, sondern auch der zugehörige Zeiger in dem Abteilungsobjekt gelöscht werden.

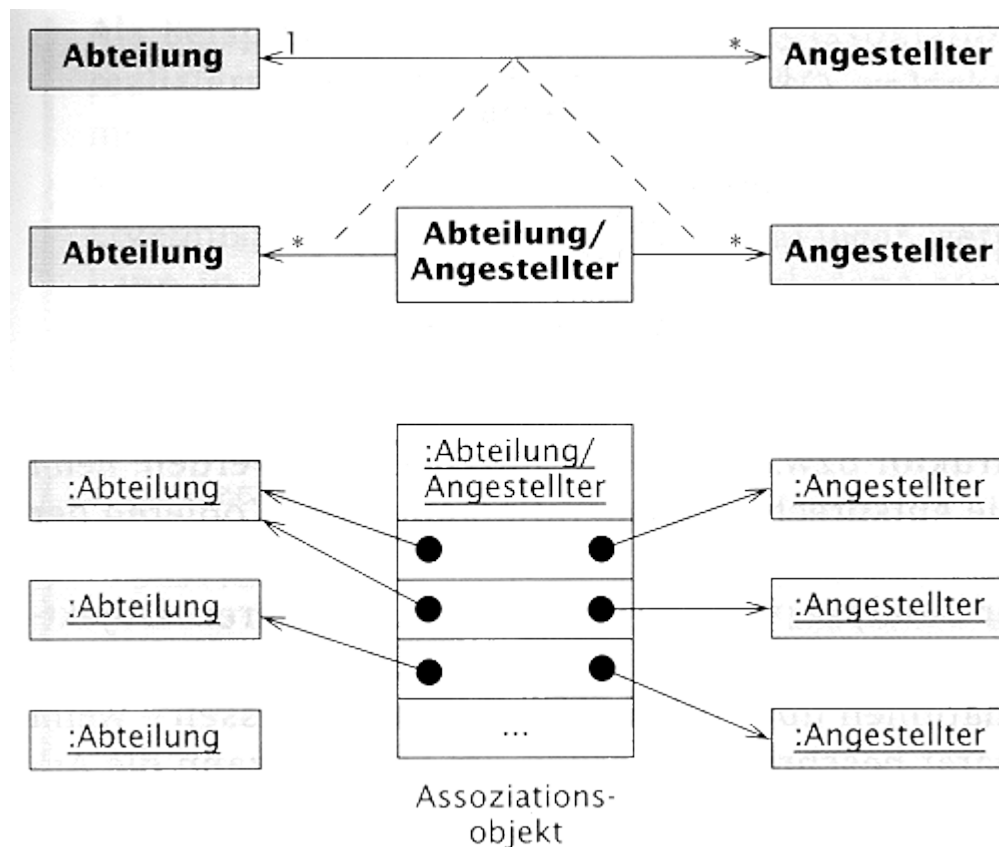




**Abbildung 6.41 Realisierung einer bidirektionalen Assoziation mittels Zeigern**

Eine andere Möglichkeit der Realisierung von bidirektionalen Assoziationen besteht in der Verwendung mittels einer eigenen Klasse, die jedoch nicht im Klassendiagramm dargestellt wird. Eine solche Klasse wird als Assoziationsobjekt bezeichnet. Nur dieses Assoziationsobjekt kennt nun welches Objekt mit welchem Objekt verbunden ist. Die assoziierten Objekte selber kennen sich nicht mehr direkt. Eine solche Vorgehensweise ist insbesondere hilfreich, wenn Assoziationen nachträglich hinzugefügt und die Klassen nicht verändert werden sollen. Dies ist z. B. bei vordefinierten Bibliotheksklassen der Fall, weil hier die Veränderung der bestehenden Klassen nicht möglich ist.

In Abbildung 6.42 wird eine solche bidirektionale Assoziation zwischen **Angestellter** und **Abteilung** realisiert. Dazu wird die Klasse **Abteilung/Angestellter** hinzugefügt, von der nur ein einziges Objekt existiert. Dieses Objekt enthält eine Liste aus Zeigereinträgen, wobei jeder Zeiger aus einem Eintrag auf einen Angestellten und eine Abteilung besteht. Ein Objekt der Klasse **Abteilung/Angestellter** kann unabhängig von einem Eintrag eines Angestellten existieren. Beim Erfassen eines neuen Angestellten muss dagegen ein neuer Eintrag erfolgen. Dabei muss eventuell ein neues Abteilungsobjekt erzeugt werden. Beim Ausscheiden eines Angestellten müssen die entsprechenden Einträge im Assoziationsobjekt ebenfalls entfernt werden. Wird eine Abteilung aufgelöst, müssen alle Angestellten dieser Abteilung einer anderen Abteilung zugewiesen werden.



**Abbildung 6.42 Realisierung einer bidirektionalen Assoziation mittels Assoziationsobjekt**

Aggregationen werden prinzipiell genauso realisiert wie „normale“ Assoziationen. Hierbei muss das Ganze stets seine Teile kennen, d. h. es muss eine Navigation vom Ganzen zu seinen Teilen möglich sein.

Auch bei der Komposition ist dies zu fordern. Darüber hinaus muss beachtet werden, dass die Operationen, die das Ganze betreffen, auch Auswirkungen auf seine Teile haben. Ganzes und Teil einer Komposition müssen als eine Einheit verstanden werden.

Die Realisierung einer Komposition kann auf zwei Weisen erfolgen. Zum einen kann sie mittels Zeiger implementiert werden, zum anderen ist eine Lösung durch physisches Enthaltensein (by value) möglich. Der Lösungsweg by value hat den Vorteil, dass die Teile automatisch mit dem Ganzen erzeugt werden. Auch das Kopieren bezieht sich automatisch auf seine Teile. Bei einer by reference Lösung muss das Erzeugen und Löschen der Teile im Konstruktor bzw. Destruktor erfolgen. Beim Kopieren muss auch die Kopie der Teile erfolgen.

```

class Ganzes
{
    TeilA einTeilA;
    TeilB* einTeilB;

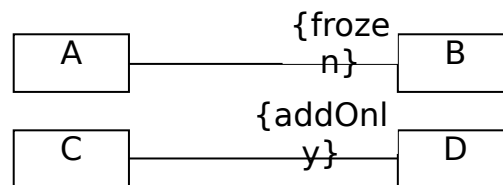
public:
    Ganzes() {einTeilB = new TeilB;}      // Erzeugen des Teils B
    ~Ganzes() {delete einTeilB;} // Löschen des Teils B
};

```

Die UML ermöglicht es, für Assoziationen Merkmale zu definieren, die auf jeder Seite der Assoziation angegeben werden können.

- {frozen} mit frozen wird definiert, dass eine Objektverbindung weder hinzugefügt noch gelöscht werden kann, nachdem ein Objekt der Klasse B erzeugt und initialisiert wurde.
- {addOnly} addOnly gibt an, dass für ein Objekt der Klasse D zwar weitere Verbindungen eingetragen werden dürfen, aber keine Verbindung wieder gelöscht werden darf.

Nachfolgend eine Implementierung einer bidirektionalen Assoziation



in C++.

Kardinalität maximal 1:

```

class Angestellter
{protected:
    Abteilung*          arbeitetIn:
public:
    void link (Abteilung* abt);    //Erstellung einer Verbindung
    void unlink (Abteilung* abt); //Löschen einer Verbindung
    void getlink(Abteilung* &abt); //Lesen einer Verbindung
};

void Angestellter::link (Abteilung* abt) {arbeitetIn = abt;}
void Angestellter::unlink (Abteilung* abt) {arbeitetIn = Null;}
void Angestellter::getlink(Abteilung* &abt) {abt = arbeitetIn}

```

Kardinalität größer 1:

Eine Menge von Objekten lässt sich günstiger mit einer generischen Klasse verwalten.

```
template <class VObject>
```

```
class VectorBag
```

```
{      //verwaltet Referenzen auf Objekte von VObject
```

```
    ...
```

```
    virtual void addElement(Vobjekt* Obj);
```

```
    void delElement(int pos);
```

```
    Vobject getElement(int pos);
```

```
};
```

```
class Abteilung
```

```
{protected:
```

```
    VectorBag<Angestellter> Mitarbeiter;
```

```
public:
```

```
    void link (Angestellter * ang);
```

```
    void unlink(Angestellter* ang);
```

```
    int getlink(Angestellter* &ang, int pos); // = 0, wenn ok
```

```
};
```

```
void Angestellter::link (Angestellter * ang) {Mitarbeiter.addElement(ang);}
```

```
void Angestellter::unlink(Angestellter* ang) { Mitarbeiter.delElement(ang);}
```

```
int Angestellter::getlink(Angestellter* &ang, int pos) { ang = Mitarbeiter.getEle-  
ment(pos);}
```

**6..5. Polymorphismus**

**6..6. Vererbung**

**6..7. Paket**

**6..8. Szenario**

**6..9. Zustandsautomat**

## **7. Implementierung**

**Programmiergrundlagen anhand von Visual Basic**

### **Lehrziele**

- 1. Variablentypen**
- 2. Typ-Umwandlungsfunktionen**
- 3. Zugriffsrechte**
- 4. Prozeduren und Funktionen**
- 5. Argumentübergabe**
- 6. Anweisungen**

## 7.1 Variablentypen

### Boolean-Datentyp

Variablen vom Datentyp **Boolean** werden als 16-Bit-Zahlen (2 Bytes) gespeichert, die nur die Werte **True** oder **False** annehmen können. Variablen vom Datentyp **Boolean** werden als **Wahr** oder **Falsch** ausgegeben, wenn **Print** verwendet wird, bzw. als **#TRUE#** oder **#FALSE#**, wenn **Write#** verwendet wird. Mit den Schlüsselwörtern **True** und **False** weist man einer Variablen vom Typ **Boolean** einen von zwei Zuständen zu.

Beim Umwandeln numerischer Datentypen in Werte des Typs **Boolean** wird 0 zu **False**, und alle anderen Werte werden zu **True**. Beim Umwandeln von Werten des Datentyps **Boolean** in numerische Datentypen wird **False** zu 0 und **True** zu -1.

### Byte - Datentyp

Variablen vom Datentyp **Byte** werden als einzelne 8-Bit-Zahlen (1 Byte) ohne Vorzeichen gespeichert und haben einen Wert im Bereich von 0 bis 255.

Der Datentyp **Byte** wird häufig zur Speicherung binärer Daten verwendet.

### Currency-Datentyp

Variablen vom Datentyp **Currency** werden als 64-Bit-Zahlen (8 Bytes) in einem ganzzahligen Format gespeichert und durch 10.000 dividiert, was eine Festkommazahl mit 15 Vorkomma- und 4 Nachkommastellen ergibt. Diese Darstellung ergibt einen Wertebereich von -922.337.203.685.477,5808 bis 922.337.203.685.477,5807. Das Typenkennzeichen für **Currency** ist das Zeichen (@). Der Datentyp **Currency** eignet sich besonders für Berechnungen mit Geldbeträgen und für Festkommaberechnungen, die eine hohe Genauigkeit erfordern.

### Date-Datentyp

Variablen vom Datentyp **Date** werden als 64-Bit-Gleitkommazahlen (8 Bytes) (nach IEEE-Standard) gespeichert und können ein Datum im Bereich vom 01. Januar 100 bis zum 31. Dezember 9999 und eine Uhrzeit im Bereich von 0:00:00 bis 23:59:59 darstellen. Jeder gültige Wert eines Datums- oder Zeitliterals kann einer Variablen vom Datentyp **Date** zugewiesen werden. Ein Datumsliteral muß durch das Zeichen (#) eingeschlossen sein, zum Beispiel: **#January 1, 1993#** oder **#1 Jan 93#**.

### Decimal-Datentyp

Variablen des Datentyps **Decimal** werden als 96-Bit-Ganzzahlen (12 Bytes) mit Vorzeichen mit einer variablen Potenz zur Basis 10 gespeichert. Die Potenz zur Basis 10 wird als Skalierungsfaktor verwendet und bestimmt die Anzahl der Nachkommastellen, die in einem Bereich von 0 bis 28 liegen kann. Beim Skalierungsfaktor von 0 (keine Nachkommastellen) liegt der größtmögliche Wert bei +/-79.228.162.514.264.337.593.543.950.335. Bei 28 Nachkommastellen liegt der größte Wert bei +/-7,9228162514264337593543950335 und der kleinste Wert, der ungleich Null ist, bei +/-0,00000000000000000000000000000001.

### Double-Datentyp

Variablen vom Datentyp **Double** (Gleitkommazahl mit doppelter Genauigkeit) werden als 64-Bit-Gleitkommazahlen (8 Bytes) nach IEEE im Bereich von -1,79769313486231E308 bis -4,94065645841247E-324 für negative Werte und von 4,94065645841247E-324 bis 1,79769313486232E308 für positive Werte gespeichert.

### Integer-Datentyp

Variablen vom Datentyp **Integer** werden als 16-Bit-Zahlen (2 Bytes) in einem Bereich von -32.768 bis 32.767 gespeichert. Mit Variablen vom Datentyp **Integer** können auch Aufzählungswerte dargestellt werden. Ein Aufzählungswert besteht aus einer endlichen Menge eindeutiger ganzer Zahlen, von denen jede im verwendeten Kontext eine spezielle Bedeutung hat. Aufzählungswerte erlauben eine einfache Auswahl aus einer bestimmten Anzahl von Möglichkeiten, z.B. schwarz = 0, weiss = 1 usw. Um Konstanten für jeden Aufzählungswert zu definieren, wird die **Const**-Anweisung verwendet.

### Long-Datentyp

Variablen vom Datentyp **Long** (lange Ganzzahl) werden als 32-Bit-Zahlen (4 Bytes) mit Vorzeichen im Bereich von -2.147.483.648 bis 2.147.483.647 gespeichert.

### Object-Datentyp

Variablen vom Datentyp **Object** werden als 32-Bit-Adressen (4 Bytes) gespeichert, die auf Objekte in einer Anwendung verweisen. Einer Variablen, die als **Object** deklariert wurde, kann anschließend mit der **Set**-Anweisung



ein Verweis auf jedes von der Anwendung erzeugte Objekt zugewiesen werden.

### **Single-Datentyp**

Variablen vom Datentyp **Single** (Gleitkommazahl mit einfacher Genauigkeit) werden als 32-Bit-Gleitkommazahlen (4 Bytes) nach IEEE im Bereich von -3,402823E38 bis -1,401298E-45 für negative Werte und von 1,401298E-45 bis 3,402823E38 für positive Werte gespeichert.

### **String-Datentyp**

Es gibt zwei Arten von Zeichenfolgen: Zeichenfolgen variabler Länge und Zeichenfolgen fester Länge.

- Zeichenfolgen variabler Länge können bis zu 2 Milliarden (oder  $2^{31}$ ) Zeichen enthalten.
- Zeichenfolgen fester Länge können 1 bis etwa 64 KB ( $2^{16}$ ) Zeichen enthalten.

Die Codes für Zeichen vom Datentyp **String** liegen im Bereich von 0 bis einschließlich 255. Die ersten 128 Zeichen (0 bis 127) entsprechen den Buchstaben und Symbolen auf einer US-amerikanischen Standardtastatur. Diese ersten 128 Zeichen stimmen mit den im ASCII-Zeichensatz definierten Zeichen überein. Die zweiten 128 Zeichen (128 bis 255) sind Sonderzeichen, z.B. Buchstaben aus internationalen Alphabeten, Akzentzeichen, Währungssymbole und Symbole für mathematische Brüche.

### **Variant-Datentyp**

Der Datentyp **Variant** ist der Datentyp für alle Variablen, die nicht explizit (durch eine Anweisung wie **Dim**, **Private**, **Public** oder **Static**) als anderer Datentyp deklariert werden.

**Variant** ist ein besonderer Datentyp, der beliebige Daten mit Ausnahme von String-Daten fester Länge und benutzerdefinierte Typen enthalten kann. Ein **Variant** kann auch die speziellen Werte **Empty**, **Error**, **Nothing** und Null enthalten. Mit der Funktion **VarType** oder **TypeName** kann festgelegt werden, wie die Daten in einer Variablen vom Datentyp **Variant** interpretiert werden.

### **Benutzerdefinierter Datentyp**

Benutzerdefinierte Datentypen können ein oder mehrere Elemente eines beliebigen Datentyps, eines Datenfeldes oder eines bereits bestehenden benutzerdefinierten Datentyps enthalten und werden mit einer Type-Anweisung definiert. Beispiel:

```

Type Person
    Name1 As String      ' String-Variable für Namen.
    Geburtstag As Date   ' Date-Variable für Geburtstag.
    Geschlecht As Integer ' Integer-Variable für
                        ' Geschlecht (0 für weiblich, 1 für männlich)
End Type

```

## 7. 2 Typ-Umwandlungsfunktionen

Jede Funktion legt für einen bestimmten Datentyp zwingend einen Ausdruck fest.

### Syntax

```

CBool(Ausdruck)   CByte(Ausdruck)   CCur(Ausdruck)
CDate(Ausdruck)   CDb1(Ausdruck)    CDec(Ausdruck)
CInt(Ausdruck)    CLng(Ausdruck)    CSng(Ausdruck)
CVar(Ausdruck)    CStr(Ausdruck)

```

Das erforderliche Argument *Ausdruck* ist ein Zeichenfolgeausdruck oder ein numerischer Ausdruck.

### Rückgabetypen

Der Funktionsname legt den Rückgabetyt wie folgt fest:

Funktion	Rückgabetyt	Bereich des Arguments <i>Ausdruck</i>
<b>Cbool</b>	Boolean	Eine gültige Zeichenfolge oder ein gültiger numerischer Ausdruck.
<b>CByte</b>	Byte	0 bis 255.
<b>Ccur</b>	Currency	-922.337.203.685.477,5808 bis 922.337.203.685.477,5807.
<b>CDate</b>	Date	Ein beliebiger gültiger Datumsausdruck.
<b>CDbl</b>	Double	-1,79769313486231E308 bis -4,94065645841247E-324 für negative Werte; 4,94065645841247E-324 bis 1,79769313486232E308 für positive Werte.
<b>Cdec</b>	Decimal	+/-79.228.162.514.264.337.593.543.950.335 für skalierte Ganzzahlen, d.h. Zahlen ohne Dezimalstellen. Für Zahlen mit 28 Dezimalstellen gilt der Bereich +/-7,9228162514264337593543950335. Die kleinste mögliche Zahl ungleich Null ist 0,0000000000000000000000000001.

<b>CInt</b>	Integer	-32.768 bis 32.767; Nachkommastellen werden gerundet.
<b>CLng</b>	Long	-2.147.483.648 bis 2.147.483.647; Nachkommastellen werden gerundet.
<b>CSng</b>	Single	-3,402823E38 bis -1,401298E-45 für negative Werte; 1,401298E-45 bis 3,402823E38 für positive Werte.
<b>Cvar</b>	Variant	Numerische Werte im Bereich des Typs <b>Double</b> . Nichtnumerische Werte im Bereich des Typs <b>String</b> .
<b>CStr</b>	String	Rückgabe für <b>CStr</b> hängt vom Argument <i>Ausdruck</i> ab.

## Beispiele :

### CStr-Funktion

In diesem Beispiel wird die **CStr**-Funktion verwendet, um einen numerischen Wert in den Typ **String** umzuwandeln.

```
Dim TestDouble, TestString
TestDouble = 437.324           ' TestDouble hat den Typ Double.
TestString = CStr(TestDouble)  ' TestString enthält "437,324".
```

### CInt-Funktion

In diesem Beispiel wird die **CInt**-Funktion zum Umwandeln eines Wertes in den Typ **Integer** verwendet.

```
Dim TestDouble, TestInt
TestDouble = 2345.5678        ' TestDouble hat den Typ Double.
TestInt = CInt(TestDouble)    ' TestInt enthält 2346.
```

### Cvar-Funktion

Dieses Beispiel verwendet die **CVar**-Funktion zum Umwandeln eines Ausdrucks in den Typ **Variant**.

```
Dim TestInt, TestVar
TestInt = 4534                ' TestInt hat den Typ Integer.
TestVar = CVar(TestInt & "000") ' TestVar enthält die Zeichenfolge
                                ' 4534000.
```

### **Java**

In Java findet die Typ-Umwandlungsfunktion wie folgt statt :

```
Double zahl = 1.3454345 ; // Deklariert und Initialisiert die Variable zahl
von Typ Double
int ganzzahl = (int) zahl; // Deklariert und Initialisiert die Variable ganz-
zahl vom Typ          Integer. (int) schneidet vom Double die Nachkom-
mastellen ab.          Das Ergebnis ist ein Integer
```

### **oder**

```
Date theDate = new Date(); // Es wird ein Objekt der Klasse Date erzeugt
System.out.println( toString(theDate));
// mit dem Aufruf toString() wird das Datum in einen String umgewandelt.
```

## **7.3 Zugriffsrechte**

### **Dim-Anweisung**

Die **Dim**-Anweisung wird auf Modul- oder Prozedurebene verwendet, um den Datentyp einer Variablen zu deklarieren. Dies ist in Visual Basic nicht zwingend erforderlich, aber bei einem guten Programmierstil, der eine effiziente Fehlersuche unterstützt, dringend zu empfehlen. Die Deklaration von Variablen kann durch das Hinzufügen der Anweisung

```
Option Explicit
```

in der ersten Zeile eines Moduls erzwungen werden. Variable, die mit **Dim** auf Modulebene deklariert wurden, stehen allen Prozeduren innerhalb des Moduls zur Verfügung. Auf Prozedurebene deklarierte Variablen stehen nur innerhalb der umgebenden Prozedur zur Verfügung.

Die folgende Anweisung deklariert beispielsweise eine Variable als Integer.

**Dim AnzahlAngestellte As Integer**

Verwenden Sie die **Dim**-Anweisung auch, um den Objekttyp einer Variablen zu deklarieren. Die folgende Anweisung deklariert eine Variable für eine neue Instanz einer Tabelle.

**Dim X As New Worksheet**

Wenn das Schlüsselwort **New** beim Deklarieren einer Objektvariable nicht verwendet wird, muß die Variable, die auf ein Objekt verweist, mit der **Set**-Anweisung einem existierenden Objekt zugewiesen werden, bevor sie verwendet werden kann. Solange ein Objekt zugewiesen ist, hat die deklarierte Objektvariable den Spezialwert **Nothing**, d.h., sie verweist auf keine bestimmte Instanz eines Objekts.

## In Java

Sichtbarkeit	Public	protected	Default	private
In der Klasse	Ja	Ja	Ja	Ja
Klasse im selben Paket	Ja	Ja	Ja	Nein
Außerhalb der Pakets	Ja	Nein	Nein	Nein
In einer Subklasse	Ja	Ja	Ja	Nein
Subklassen außerhalb des Pakets	Ja	Ja	Nein	Nein

**Static-Anweisung**

Wenn der Code eines Moduls ausgeführt wird, behalten mit der **Static**-Anweisung deklarierte Variable ihren Wert, bis das Modul zurückgesetzt oder neu gestartet wird. In nichtstatischen Prozeduren können die **Static**-Anweisung verwendet werden, um explizit Variablen zu deklarieren, die nur innerhalb der Prozedur verfügbar sind, deren Lebensdauer aber der des Moduls entspricht, in dem die Prozedur definiert wurde.

Mit einer **Static**-Anweisung innerhalb einer Prozedur können Sie den Datentyp einer Variablen deklarieren, die ihren Wert zwischen Prozeduraufrufen behält. Die folgende Anweisung deklariert zum Beispiel ein Datenfeld fester Größe für ganze Zahlen:

**Static AngestelltenAnzahl(200) As Integer**

Die folgende Anweisung deklariert eine Variable für eine neue Instanz einer Tabelle:

**Static X As New Worksheet**

Das Schlüsselwort **New** erzeugt eine *Instanz* eines Objekts, d.h. es wird zusätzlich zur Adresse ein entsprechender Speicherbereich für die Elemente

des Objekts reserviert. Wenn das Schlüsselwort **New** nicht bei der Deklaration einer Objektvariablen verwendet wird, existiert momentan keine Instanz des Objekts. Einer Variablen mit einem Verweis auf ein Objekt muß dann mit der **Set**-Anweisung ein existierendes Objekt zugewiesen werden, bevor sie verwendet werden kann. Anderenfalls besitzt sie den Spezialwert **Nothing**, der anzeigt, daß die Variable auf keine bestimmte Instanz eines Objekts verweist.

### Private-Anweisung

Variablen vom Typ **Private** stehen nur in dem Modul zur Verfügung, in dem sie deklariert wurden. Mit der **Private**-Anweisung deklariert man, ähnlich wie mit der **Dim**-Anweisung, den Datentyp einer Variablen. Die folgende Anweisung deklariert beispielsweise eine Variable als **Integer**:

```
Private AnzahlAngestellte As Integer
```

Verwenden Sie eine **Private**-Anweisung auch, um den Objekttyp einer Variablen zu deklarieren. Die folgende Anweisung deklariert eine Variable für eine neue Instanz einer Tabelle.

```
Private X As New Worksheet
```

### Public-Anweisung

Auf Variable, die mit der **Public**-Anweisung deklariert wurden, kann von allen Prozeduren in allen Modulen zugegriffen werden. Wenn allerdings die Option **Private** Module aktiviert wurde, sind die Variablen nur innerhalb des zugehörigen Projekts öffentlich.

Mit der **Public**-Anweisung deklariert man ebenfalls den Datentyp einer Variablen. Die folgende Anweisung deklariert beispielsweise eine Variable als **Integer**:

```
Public AnzahlAngestellte As Integer
```

Analog zu den **Dim**- und **Private**-Anweisungen kann die **Public**-Anweisung auch für die Deklaration des Objekttyps einer Variablen verwendet werden. Beispiel:

```
Public X As New Worksheet
```

Bemerkung: Die **Public**-Anweisung kann nicht in Klassenmodulen zur Deklaration einer Zeichenfolgenvariable fester Länge verwendet werden.

## 7.4 Prozeduren und Funktionen

### Function-Prozeduren

sind standardmäßig öffentlich, wenn sie nicht explizit mit **Public**, **Private** oder **Friend** deklariert werden. Wird **Static** nicht angegeben, so bleiben die Werte lokaler Variabler zwischen den Aufrufen nicht erhalten. Das Schlüsselwort **Friend** kann nur in Klassenmodulen verwendet werden. Allerdings können **Friend**-Prozeduren von Prozeduren in jedem Modul eines Projekts aufgerufen werden. Eine **Friend**-Prozedur wird nicht in der Typenbibliothek ihrer übergeordneten Klasse angezeigt und kann nicht spät gebunden werden.

Vorsicht: Manche **Function**-Prozeduren sind rekursiv (d.h., sie rufen sich selbst auf, um eine bestimmte Operation durchzuführen). Rekursionen können jedoch Stapelüberläufe verursachen. Vermeiden Sie daher das Schlüsselwort **Static** in rekursiven **Function**-Prozeduren.

Der gesamte ausführbare Code muß sich in Prozeduren befinden. Innerhalb einer **Function**-, **Sub**- oder **Property**-Prozedur kann keine weitere **Function**-Prozedur definiert werden.

Die **Exit Function**-Anweisung führt zum unmittelbaren Verlassen einer **Function**-Prozedur. Die Programmausführung wird mit der Anweisung fortgesetzt, die auf die Anweisung folgt, welche die **Function**-Prozedur aufgerufen hat. **Exit Function**-Anweisungen können an

Im folgenden Beispiel wird einer Funktion mit dem Namen BinäreSuche ein Rückgabewert zugewiesen. Der zugewiesene Wert ist in diesem Fall **False**, da ein bestimmter Wert nicht gefunden wurde.

```
Function BinäreSuche(. . .) As Boolean
. . .
    ' Wert nicht gefunden. Rückgabewert ist False.
    If untGrnze > obrGrnze Then
        BinäreSuche = False
        Exit Function
    End If
. . .
End Function
```

### **Sub**-Prozeduren

sind standardmäßig öffentlich, wenn sie nicht explizit mit **Public** oder **Private** festgelegt werden. Wird **Static** nicht angegeben, so bleiben die Werte lokaler Variablen zwischen den Aufrufen nicht erhalten.

**Vorsicht** **Sub**-Prozeduren können rekursiv sein (d.h., sie rufen sich selbst auf, um eine bestimmte Operation durchzuführen). Rekursionen können jedoch zu Stapelüberläufen führen. Vermeiden Sie daher das Schlüsselwort **Static** in rekursiven **Sub**-Prozeduren.

Der gesamte ausführbare Code muß sich in Prozeduren befinden. Innerhalb einer **Sub**-, **Function**- oder **Property**-Prozedur kann keine weitere **Sub**-Prozedur definiert werden.

Das Schlüsselwort **Exit Sub** führt zum unmittelbaren Verlassen einer **Sub**-Prozedur. Die Programmausführung wird mit der Anweisung fortgesetzt, die auf die Anweisung folgt, die die **Sub**-Prozedur aufgerufen hat. **Exit Sub**-Anweisungen können beliebig oft an beliebigen Stellen innerhalb einer **Sub**-Prozedur verwendet werden.

Wie bei einer **Function**-Prozedur handelt es sich bei der **Sub**-Prozedur um eine eigenständige Prozedur, die Argumente erhalten, eine Reihe von Anweisungen ausführen und die Werte der übergebenen Argumente ändern kann. Im Gegensatz zu einer **Function**-Prozedur, die einen Wert zurückgibt, kann eine **Sub**-Prozedur aber nicht in einem Ausdruck verwendet werden.

Der Aufruf einer **Sub**-Prozedur erfolgt durch Angabe des Prozedurnamens gefolgt von der Argumentliste.

### Sub-Anweisung

In diesem Beispiel wird die **Sub**-Anweisung verwendet, um Namen, Argumente und Prozedurrumpf einer **Sub**-Prozedur zu definieren.

```
' Definition der Sub-Prozedur.
' Sub-Prozedur mit zwei Argumenten.
Sub FlächeBerechnen(Länge, Breite)
    Dim Fläche As Double      ' Lokale Variable deklarieren.
    If Länge = 0 Or Breite = 0 Then
        ' Wenn ein Argument = 0,
        Exit Sub              ' dann Sub unmittelbar
                               ' verlassen.
    End If
    Fläche = Länge * Breite    ' Rechteckfläche berechnen.
    Debug.Print Fläche         ' Fläche im Testfenster
                               ' ausgeben.
End Sub
```

### Java

In Java nennen sich Funktionen Methoden.



Ein Beispiel :

```
public static void main(String[] argv)
{
    System.out.println("Guten Tag");
}
```

Der Anfang und das Ende der Methode wird durch die geschweiften Klammern gekennzeichnet. Diese Methode bekommt beim Aufruf ein String Array mit Namen argv übergeben und die Methode wird beendet ohne dass ein Wert zurückgegeben wird ( void ).

Beispiel 2 :

```
int berechnen(int a, int b)
{
    int c;
    c = a + b;
    return c;
}
```

Diese Methode bekommt 2 Integer Werte übergeben und speichert diese in die Variablen a und b. In die Variable c wird die Addition aus a + b gespeichert und mit return zurückgegeben.

## 7.5 Argumentübergabe

### ByRef

```
Sub sortiere (ByRef zahl as Integer)
    If zahl > 1000 Then
        zahl = 1
    Exit Function
End If
End Function
```

Das Argument wird als Referenz übergeben. **ByRef** ist der Standard in Visual Basic, seine Angabe ist daher optional. Eine Möglichkeit, die Adresse eines Arguments an eine Prozedur anstelle des Werts zu übergeben. Dadurch kann die Prozedur auf die tatsächliche Variable zugreifen. Das be-

deutet, daß der aktuelle Wert der Variablen von der Prozedur geändert werden kann, an die er übergeben wird. Wenn nichts anderes angegeben wird, werden Argumente als Referenz übergeben.

### **ByVal**

```
Sub sortiere (ByVal zahl as Integer)
    If zahl > 1000 Then
        zahl = 1
        Exit Function
    End If
End Function
```

Eine Möglichkeit, den Wert eines Arguments an eine Prozedur anstelle der Adresse zu übergeben. Dadurch kann die Prozedur auf ein Kopie der Variablen zugreifen. Das bedeutet, daß der aktuelle Wert der Variablen nicht von der Prozedur verändert werden kann, an die er übergeben wird.

```
Sub sortiere (ByVal zahl as Integer)
    If zahl > 1000 Then
        zahl = 1
        Exit Function
    End If
End Function
```

### **Optional**

Schlüsselwort, das angibt, daß ein Argument nicht erforderlich ist. Alle im Anschluß an **Optional** in *ArgListe* angegebenen Argumente müssen auch optional sein und mit dem Schlüsselwort **Optional** deklariert werden. **Optional** kann nicht verwendet werden, wenn **ParamArray** verwendet wird.

## **7.6 Anweisungen**

### **Programmschleifen**

#### **Do...Loop-Anweisung**

Wiederholt einen Block mit Anweisungen, solange eine Bedingung den Wert **True** hat oder bis eine Bedingung den Wert **True** erhält.

#### **Syntax**

```

Do [{While | Until} Bedingung]
[Anweisungen]
[Exit Do]
[Anweisungen]

```

## Loop

Sie können auch die folgende, ebenfalls zulässige Syntax verwenden:

```

Do
[Anweisungen]
[Exit Do]
[Anweisungen]
Loop [{While | Until} Bedingung]

```

Die Syntax für die **Do Loop**-Anweisung besteht aus folgenden Teilen:

## Beispiel :

```

Dim Check, Counter
Test = True: Zähler = 0      ' Variablen initialisieren.
Do                          ' Äußere Schleife.
    Do While Zähler < 20    ' Innere Schleife.
        Zähler = Zähler + 1 ' Zähler hochzählen.
        If Zähler = 10 Then ' Wenn Bedingung = True,
            Test = False    ' Attributwert auf False setzen.
            Exit Do         ' Innere Schleife verlassen.
        End If
    Loop
Loop Until Test = False     ' Äußere Schleife sofort verlassen.

```

## Java

In Java kann man eine Endlosschleife so realisieren :

```

while(true)
{
    System.out.println("Hallo");
}

```

Auch hier wird der Anfang und das Ende mit den geschweiften Klammern gekennzeichnet.

## For/next Anweisung

Wiederholt eine Gruppe von Anweisungen so oft wie angegeben.

## Syntax

```

For Zähler = Anfang To Ende [Step Schritt]
[Anweisungen]
[Exit For]
[Anweisungen]
Next [Zähler]

```

Die Syntax für die **For...Next**-Anweisung besteht aus folgenden Teilen:

<i>Zähler</i>	Erforderlich. Numerische Variable, die als Schleifenzähler dient. Eine boolesche Variable oder ein Element eines Datenfeldes ist nicht zulässig.
<i>Anfang</i>	Erforderlich. Startwert von <i>Zähler</i> .
<i>Ende</i>	Erforderlich. Endwert von <i>Zähler</i> .
<i>Schritt</i>	Optional. Betrag, um den <i>Zähler</i> bei jedem Schleifendurchlauf verändert wird. Falls kein Wert angegeben wird, ist die Voreinstellung für <i>Schritt</i> eins.
<i>Anweisungen</i>	Optional. Eine oder mehrere Anweisungen zwischen <b>For</b> und <b>Next</b> , die so oft wie angegeben ausgeführt werden.

## Beispiel :

```

Dim Wörter, Zeichen, Text1
For Wörter = 10 To 1 Step -1      ' 10 Wiederholungen.
    For Zeichen = 0 To 9          ' 10 Wiederholungen.
        Text1 = Text1 & Zeichen  ' Zahl an Zeichenfolge anfügen.
    Next Zeichen                  ' Zähler hochzählen
    Text1 = Text1 & " "           ' Leerzeichen anfügen.
Next Wörter

```

## Java

Eine For schleife sieht in Java so aus.

```

for(int a = 0 ; a < 100 ; a++)
{
    System.out.println("Hallo");
}

```

Diese Schleife läuft 100 mal bzw. bis **a** den Wert 99 erreicht hat. Es gibt in Java kein **Next**

## Programmverzweigungen

### if then else oder elseif

Führt eine Gruppe von Anweisungen aus, wenn bestimmte Bedingungen erfüllt sind, die vom Wert eines Ausdrucks abhängen.

## Syntax

**If** *Bedingung* **Then** [*Anweisungen*] [**Else** *elseAnweisungen*]

Alternativ können Sie die Block-Syntax verwenden:

**If** *Bedingung* **Then**  
[*Anweisungen*]

[**ElseIf** *Bedingung-n* **Then**  
[*elseifAnweisungen*] ...

[**Else**  
[*elseAnweisungen*]]

**End If**

Die Syntax für die **If...Then...Else**-Anweisung besteht aus folgenden Teilen:

<i>Bedingung</i>	<p>Erforderlich. Ein oder mehrere Ausdrücke der beiden folgenden Arten:</p> <p>Ein numerischer Ausdruck oder ein Zeichenfolgenausdruck, der <b>True</b> oder <b>False</b> ergibt. Wenn <i>Bedingung</i> den Wert Null hat, wird <i>Bedingung</i> als <b>False</b> interpretiert.</p> <p>Ein Ausdruck der Form <b>TypeOf Objektname Is Objekttyp</b>. <i>Objektname</i> ist ein beliebiger Objektverweis, und <i>Objekttyp</i> ist ein beliebiger gültiger Objekttyp. Der Ausdruck ergibt <b>True</b>, wenn <i>Objektname</i> dem Objekttyp entspricht, der durch <i>Objekttyp</i> angegeben wird. Andernfalls ist das Ergebnis <b>False</b>.</p>
<i>Anweisungen</i>	Optional in Form eines Blocks, erforderlich in der einzeiligen Variante, die keinen <b>Else</b> -Abschnitt beinhaltet. Eine oder mehrere durch Doppelpunkte getrennte Anweisungen, die ausgeführt werden, wenn <i>Bedingung</i> den Wert <b>True</b> hat.
<i>Anweisung-n</i>	Optional. Dieselbe Bedeutung wie <i>Bedingung</i> .
<i>Elseif Anweisungen</i>	Optional. Eine oder mehrere Anweisungen, die ausgeführt werden, wenn die zugehörige Bedingung ( <i>Bedingung-n</i> ) <b>True</b> ergibt.
<i>Else Anweisungen</i>	Optional. Eine oder mehrere Anweisungen, die ausgeführt werden, wenn keine der Bedingungen ( <i>Bedingung</i> -Ausdruck oder <i>Bedingung-n</i> -Ausdruck) <b>True</b> er-

gibt.

### Beispiel:

In diesem Beispiel wird die Blockform und die einzeilige Form der If...Then...Else-Anweisung gezeigt und zudem die Verwendung von If TypeOf...Then...Else demonstriert.

```
Dim Zahl, Stellen, Text1
Zahl = 53      ' Variable initialisieren.
If Zahl < 10 Then
    Stellen = 1
ElseIf Zahl < 100 Then
    ' Bedingung ist erfüllt, also wird die nächste Anweisung ausgeführt.
    Stellen = 2
Else
    Stellen = 3
End If
' Zuweisen eines Wertes mit der einzeiligen Syntaxvariante.
If Stellen = 1 Then Text1 = "Eine" Else Text1 = "Mehrere"
```

Mit der If TypeOf-Konstruktion wird festgelegt, welcher Steuerelementtyp an eine Prozedur übergeben wird.

```
Sub Steuerelementverarbeitung(StElement1 As Control)
    If TypeOf StElement1 Is CommandButton Then
        Debug.Print "Sie haben folgenden Steuerelementtyp übergeben: " _
            & TypeName(StElement1)
    ElseIf TypeOf StElement1 Is CheckBox Then
        Debug.Print "Sie haben folgenden Steuerelementtyp übergeben: " _
            & TypeName(StElement1)
    ElseIf TypeOf StElement1 Is TextBox Then
        Debug.Print "Sie haben folgenden Steuerelementtyp übergeben: " _
            & TypeName(StElement1)
    End If
End Sub
```

### Java

In Java sind die Vergleichsoperatoren u.a. das == oder != für ungleich.

### Beispiel :

```
if(weihnachten == ostern)
{
```

```

System.out.println("Weihnachten und Ostern fallen auf einen Tag");
}
else
{ System.out.println("Schade doch nicht"); } // Optional
oder
else if(weihnachten != muttertag)
{ System.out.println("Muttertag ist nicht an Weihnachten"); } //Optional

```

## Select Case-Anweisung

Führt eine von mehreren Gruppen von Anweisungen aus, abhängig vom Wert eines Ausdrucks.

### Syntax

**Select Case** *Testausdruck*

[**Case** *Ausdrucksliste-n*

[*Anweisungen-n*]]

[**Case Else**

[*elseAnw*]]

.....

**End Select**

Die Syntax der **Select Case**-Anweisung besteht aus folgenden Teilen:

Teil	Beschreibung
<i>Testausdruck</i>	Erforderlich. Ein beliebiger numerischer Ausdruck oder Zeichenfolgenausdruck.
<i>Ausdrucksliste-n</i>	Erforderlich, wenn der <b>Case</b> -Abschnitt verwendet wird. Eine durch Kommas getrennte Liste in einer oder mehreren der folgenden Formen: <i>Ausdruck</i> , <i>Ausdruck To Ausdruck</i> , <i>Is Vergleichsoperator Ausdruck</i> . Das Schlüsselwort <b>To</b> gibt einen Bereich von Werten an. Bei diesem Schlüsselwort muß der kleinere Wert immer links von <b>To</b> stehen. Verwenden Sie das Schlüsselwort <b>Is</b> in Kombination mit Vergleichsoperatoren (außer <b>Is</b> und <b>Like</b> ), um einen Bereich von Werten anzugeben. Wenn Sie das Schlüsselwort <b>Is</b> nicht angeben, wird es automatisch eingefügt.
<i>Anweisungen-n</i>	Optional. Eine oder mehrere Anweisungen, die ausgeführt werden, wenn <i>Testausdruck</i> mit irgendeinem Teil in <i>Ausdrucksliste-n</i> übereinstimmt.
<i>ElseAnw</i>	Optional. Eine oder mehrere Anweisungen, die aus-

geführt werden, wenn *Testausdruck* mit keinem der Ausdrücke im **Case**-Abschnitt übereinstimmt.

## Bemerkungen

Wenn *Testausdruck* mit irgendeinem der **Case**-Ausdrücke in der *Ausdrucksliste* übereinstimmt, werden die *Anweisungen* dieses **Case**-Abschnitts bis zum nächsten **Case**-Abschnitt (oder beim letzten **Case**-Abschnitt bis zu **End Select**) ausgeführt. Anschließend setzt das Programm die Ausführung mit der Anweisung im Anschluß an **End Select** fort. Wenn *Testausdruck* mit einer *Ausdrucksliste*-Anweisung in mehreren **Case**-Abschnitt übereinstimmt, werden nur die Anweisungen nach der ersten Übereinstimmung ausgeführt.

Die *elseAnw*-Anweisungen im **Case Else**-Abschnitt werden ausgeführt, wenn keine Übereinstimmung zwischen *Testausdruck* und einer *Ausdrucksliste* in einer der anderen **Case**-Abschnitte gefunden wird. Die **Case Else**-Anweisung ist optional, sollte aber in keinem **Select Case**-Block fehlen, damit unvorhergesehene Werte von *Testausdruck* verarbeitet werden können. Wenn keine **Case Else**-Anweisung angegeben ist und keine **Case**-Ausdrucksliste mit *Testausdruck* übereinstimmt, setzt das Programm die Ausführung mit der Anweisung im Anschluß an **End Select** fort.

Sie können in jedem **Case**-Abschnitt mehrere Ausdrücke oder Bereiche verwenden, wie zum Beispiel in der folgenden Zeile:

Case 1 To 4, 7 To 9, 11, 13, Is > Maximalwert

## Beispiel:

In diesem Beispiel wird die **Select Case**-Anweisung verwendet, um den Wert einer Variablen auszuwerten. Der zweite **Case**-Abschnitt enthält den Wert der ausgewerteten Variablen, und nur die zugehörige Anweisung wird ausgeführt.

```
Dim Zahl
Zahl = 8          ' Variable initialisieren.
Select Case Zahl  ' Zahl auswerten.
Case 1 To 5       ' Zahl von 1 bis 5.
    Debug.Print "Zahl von 1 bis 5"
                  ' Das ist der einzige Case-Abschnitt, der True ergibt.
Case 6, 7, 8      ' Zahl von 6 bis 8.
    Debug.Print "Zahl von 6 bis 8"
Case Is 9 To 10   ' Zahl ist 9 oder 10.
    Debug.Print "Größer als 8"
Case Else         ' Andere Werte.
    Debug.Print "Außerhalb von 1 bis 10"
End Select
```



Java

Ein Beispiel für eine Select Case Anweisung :

```
switch(zahlen)
{
    case 1:
        text = „1“;
        break;
    case 2:
        text = "2";
        break;
    case 3:
        text = "3"
        break;
    default :
        text = " keine Eingabe";
}
```

### With-Anweisung

Führt eine Reihe von Anweisungen für ein einzelnes Objekt oder einen benutzerdefinierten Typ aus.

#### Syntax

**With** *Objekt*  
[*Anweisungen*]

**End With**

Die Syntax der **With**-Anweisung besteht aus folgenden Teilen:

Teil	Beschreibung
<i>Objekt</i>	Erforderlich. Name eines Objekts oder eines benutzerdefinierten Typs.
<i>Anweisungen</i>	Optional. Eine oder mehrere Anweisungen, die für <i>Objekt</i> ausgeführt werden sollen.

### Bemerkungen

Mit der **With**-Anweisung können Sie eine Reihe von Anweisungen für ein bestimmtes Objekt ausführen, ohne den Namen des Objekts mehrmals angeben zu müssen. Wenn Sie zum Beispiel mehrere Eigenschaften eines be-

stimmten Objekts verändern möchten, sollten Sie die Zuweisungsanweisungen für die Eigenschaft in einer **With**-Kontrollstruktur unterbringen. Sie brauchen dann den Namen des Objekts nicht bei jeder einzelnen Zuweisung, sondern nur einmal zu Beginn der Kontrollstruktur anzugeben. Das folgende Beispiel veranschaulicht die Verwendung der **With**-Anweisung, um mehreren Eigenschaften desselben Objekts Werte zuzuweisen.

### Beispiele :

```
With Bezeichnungsfeld1
```

```
    .Height = 2000
```

```
    .Width = 2000
```

```
    .Caption = "Schönen Tag noch"
```

```
End With
```

```
With Objekt1
```

```
    .Höhe = 100
```

```
    .Titel = "Hallo Welt"
```

```
    With .Schrift
```

```
        .Farbe = Rot
```

```
        .Fett = True
```

```
    End With
```

```
End With
```

```
' Entspricht Objekt1.Höhe = 100.
```

```
' Entspricht Objekt1.Titel = "Hallo Welt".
```

```
' Entspricht Objekt1.Schrift.Farbe = Rot.
```

```
' Entspricht Objekt1.Schrift.Fett = True.
```

### Java

In Java gibt es keine With anweisung ! Java arbeitet mit Objekten und Attributen. Beispiel :

```
mensch deutscher = new mensch();
```

```
deutsche.sprache = Deutsch;
```

```
deutscher.hautfarbe = „blass, weil wegen keine Sonne ☺“;
```

Das Objekt deutscher hat jetzt die Attribute : sprache und hautfarbe

Raum für Ihre Notizen

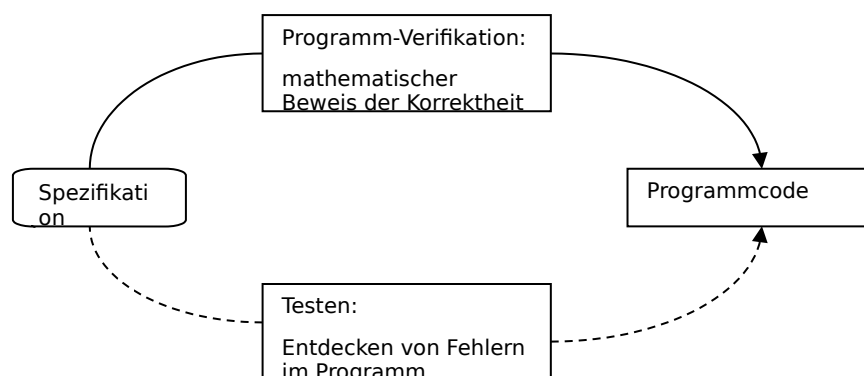
## 8. Softwaretestverfahren

### Einführung

Im Bereich der Softwareentwicklung wird Methoden der Fehlererkennung und der Fehlerbehebung eine wachsende Bedeutung beigemessen. Dies liegt auch an zwei Tendenzen, die in der Zusammensetzung der Kosten für Software-Entwicklungsprojekte sichtbar geworden sind. Zum einen wächst im Bereich der automatisierten Datenverarbeitung der prozentuale Anteil der Softwareentwicklungskosten an den Gesamtkosten. Im Vergleich dazu nahm in der Vergangenheit der Anteil der Hardwarekosten aufgrund des rasanten technischen Fortschritts im Bereich der Computertechnik beständig ab. Zum anderen stieg auch der prozentuale Anteil der Kosten für die Softwarewartung beständig an. Der Anteil der Testkosten an den Entwicklungskosten beträgt mittlerweile bis zu 50 % und der Anteil der Kosten für die Softwarewartung kann sich auf 40 % bis zu 60 % der Gesamtkosten belaufen.

Die Wartungskosten fallen nach der Installation eines Softwareproduktes in der Betriebsphase an. Bedingt sind diese meist durch Anpassung des Produkts an im Lauf des Betriebs geänderte Produkthanforderungen. Dagegen entstehen die Testkosten während der Softwareentwicklung. Ihre Ausweitung ist einerseits durch die zunehmende Komplexität von Softwareprodukten bedingt, andererseits wurde durch die unzähligen und manchmal geradezu katastrophalen Auswirkungen von Fehlern das Augenmerk von Entwicklern immer stärker auf eine möglichst weitgehende Fehlerfreiheit von Programmen gelenkt.

Ein ganzer Forschungszweig der Informatik beschäftigt sich mit den Problemen der Fehlerfreiheit und dem mathematischen Korrektheitsbeweis von Programmen.



**Abbildung 8.43 Testen und Korrektheitsbeweis**

Testen und Korrektheitsbeweis gehen von einer Bezugsbasis aus, der (Programm-) Spezifikation. Im Entwicklungskonzept ist die Spezifikation durch die Entwurfsdokumente gegeben. Natürlich stellen Methoden des

Korrektheitsbeweises hohe Anforderungen an die Genauigkeit der Spezifikation.

Das Testen von Software ist eine mit der Entwicklung eines Softwaresystems einhergehende Prozedur und kein am Ende des Entwicklungsprozesses angesiedelter einmaliger Vorgang.

## **Fehlerbegriff und Fehlerursachen**

*Ein Softwarefehler ist eine Abweichung des fertigen Produktes von der Spezifikation*

Fehler können in allen Phasen der Entwicklung auftreten. Sie können unterschieden werden in:

- Anforderungsdefinitionsfehler
- Problemanalysefehler
- Entwurfsfehler
- Implementierungsfehler

Die beiden letzten Punkte lassen sich weiter unterteilen:

- Datenreferenz-Fehler
- Datendeklarationsfehler
- Berechnungsfehler
- Vergleichsfehler
- Steuerflußfehler
- Schnittstellenfehler
- Ein- und Ausgabefehler
- sonstige Fehler

In der Entwicklungsphase sind mehrere Tätigkeiten wie das Erfassen und Verstehen der Anforderung, das Konzipieren eines Lösungswegs und das Ausformulieren des gefundenen Weges erforderlich. Dementsprechend gibt es auch unterschiedliche Ursachen für die oben genannten Fehlerarten, z.B.:

- unvollständiges Erfassen oder Nichtverstehen der Anforderung (z. B. falsche Fallunterscheidungen)
- Konzipieren eines nicht ausgereiften Lösungsweges (z. B. falsche Berechnungsalgorithmen)
- falsches Ausformulieren eines Lösungsweges (z. B. Schreibfehler)

Die Auswirkungen von Softwarefehlern können sehr unterschiedlich sein. Manchmal sind sie harmlos wie z. B. das Ausstellen von Rechnungen über Nullbeträge. Kritischer wird es im medizinischen und technischen Bereich, wo das Auftreten von Fehlern in Softwareprodukten bereits Todesopfer gefordert hat oder zu großen finanziellen Schäden führte.

## **Testphasen und Entwicklungsprozeß**

### **8..1. Testbegriff**

*Testen ist der Prozeß, ein Programm mit der Absicht auszuführen, Fehler zu finden. Ein erfolgreicher Testfall ist dadurch gekennzeichnet, dass er einen bisher unbekannten Fehler entdeckt.*

### **8..2. Testprinzipien**

Für das erfolgreiche Testen von Software sollten folgende Prinzipien beachtet werden.

- Zur vollständigen Definition eines Testfalls gehört auch die Festlegung des erwarteten Testergebnisses. Je nach Testobjekt kann das Ergebnis ein bestimmter Wert, eine Menge von Werten usw. sein.
- Ein Testobjekt sollte nicht von seinem Entwickler, sondern von einer anderen Person getestet werden. Beim Entwickler eines Testobjektes ist im allgemeinen nicht die für das Testen notwendige, destruktive Einstellung vorauszusetzen.
- Die Ergebnisse eines Tests sollten gründlich überprüft werden. Experimente haben gezeigt, dass Tester offensichtliche Fehler im Ausgabeteil nicht entdeckten.
- Testfälle sind nicht nur für gültige und erwartete Eingabedaten vorzusehen, sondern auch für ungültige und unerwartete Eingabedaten. Letzteres ist notwendig um festzustellen, ob das Testobjekt bei ungültiger Eingabe auch entsprechend reagiert.
- Das vorherige Prinzip kann man wie folgt ergänzen: Zu untersuchen, ob eine Programm nicht das tut, was es tun sollte erfüllt nur die Hälfte der Anforderung. Ebenso wichtig ist es, zu untersuchen, ob ein Programm etwas tut, was es nicht tun sollte.
- Testfälle stellen eine erhebliche Investition dar – daher sollte man keine Wegwerftestfälle verwenden. Immer wenn das Programm erneut getestet werden soll, müssen die alten Testfälle zur Verfügung stehen.
- Ein Testverfahren sollte man nicht unter der stillschweigenden Annahme planen, dass keine Fehler gefunden werden. Hat man beim Test

erst einmal Fehler gefunden, so gilt erfahrungsgemäß: Die Wahrscheinlichkeit für die Existenz weiterer Fehler in einem Programmabschnitt ist proportional zu der Zahl der bereits in diesem Abschnitt gefundenen Fehler.

### **8..3. Testphasen im Entwicklungsprozeß**

Softwarefehler können in allen Phasen des Entwicklungsprozesses verursacht werden. Entsprechend sind auch Testmaßnahmen in allen Entwicklungsphasen vorzusehen. Ein wichtiger Anteil der Softwarefehler resultieren aus mangelhafter Informationsvermittlung zwischen Entwicklungsphasen. Es bietet sich daher an, das Testen wie folgt zu organisieren:

1. Überprüfung der Phasenergebnisse durch Vergleich des Ergebnisses einer Phase mit dem Ergebnis der Vorphase. Auf diese Weise können bereits frühzeitig Fehler entdeckt werden, deren Behebung zu einem späteren Zeitpunkt mit wesentlich mehr Aufwand verbunden wäre.
2. Strukturierung des Testprozesses in Analogie zum Entwicklungsprozeß. Jeder Entwicklungsphase wird also ein separater Testabschnitt zugeordnet. In den Testabschnitten sind Testmethoden anzuwenden, die geeignet sind, abschnittsspezifische Fehler zu entdecken.

#### **Testarten**

### **8..4. Modultest**

Im Modultest wird die Funktion eines Moduls in Bezug auf die im Modulentwurf erarbeitete Modulspezifikation getestet. Man überprüft vor allem die interne Logik der Operationen bzw. Prozeduren daraufhin, ob sie die vorgesehenen aus der Spezifikation hervorgehenden Anforderungen korrekt erfüllen.

Vorteilhaft sind beim Modultest die Möglichkeit, Module parallel zu testen und die leichtere Lokalisierung von Fehlern. Insbesondere bei größeren Systemen sollte man daher auf Modultests nicht verzichten.

Was das methodische Vorgehen betrifft, ist beim Modultest zu unterscheiden zwischen:

1. Methoden zum Testen eines einzelnen Moduls.
2. Vorgehensweisen bei der Auswahl von Testdaten.
3. Strategien bei der Verzahnung von Implementierung und Test mehrerer Module.

Der Test eines einzelnen Moduls kann in der Weise erfolgen, dass ein einzelnes Modul isoliert, jedoch unter Verwendung von Stubs oder Treibern getestet wird oder dass

ein Modul mit mehreren bereits getesteten Modulen zusammengebunden und unter Verwendung von Stubs oder Treibern getestet wird.

Als Methoden zum Test eines einzelnen Moduls bieten sich an:

der Black-Box-Test,

der White-Box-Test.

Auf diese Methoden wird in Kap. 8.6 näher eingegangen. Besonderes Gewicht wird dabei auf den Black-Box-Test gelegt. Bei dieser Art des Testens sind folgende Vorgehensweise bei der Ermittlung von Testdaten bzw. Testfällen angebracht:

Bildung von Äquivalenzklassen,

Grenzwertanalyse und

Intuitive Testfallermittlung

Für den Modultest gelten, wie auch für den Test ganzer Programme die oben aufgeführten Prinzipien, u. a. :

Implementierer und Tester eines Moduls sollten verschieden Personen sein,

in einem Modul entdeckte Fehler sollten vom Implementierer behoben werden,

die verwendeten Testfälle sollten dokumentiert werden.

## **8..5. Integrationstest**

Gegenstand des Integrationstests ist der Test des Moduls, das mit anderen Modulen gemäß der angewandten Implementierungs- und Teststrategie zusammengebunden wird. Getestet werden damit:

die Operationen bzw. Prozeduren von Modulen in Bezug auf das mit den Modulspezifikationen vorgesehene Verhalten und insbesondere auch

das korrekte Zusammenwirken der Module über die im Entwurf spezifizierten Schnittstellen.

## **8..6. Systemtest**

Der Systemtest verfolgt den Zweck, das vollständig integrierte System mit der Leistungsbeschreibung, d. h. im gegebenen Fall der Anforderungsdefinition, gegenüberzustellen. Der Test besteht also aus einer Bewertung der vom System erbrachten (Ist-) Leistungen , die als Bezugsbasis dienen.

Die mit Hilfe der Anforderungsdefinition und unter Berücksichtigung der Benutzerdokumentation zu erstellenden Testfälle sollen das Abprüfen folgender Kriterien ermöglichen.



1. Vollständigkeit der Systemleistungen: Es ist im Detail zu prüfen, ob sämtliche geforderten Systemleistungen auch programmtechnisch realisiert werden. Dies kann häufig auch durch einen Vergleich von Code und Benutzerdokumentation und somit ohne Rechnereinsatz geschehen. Beispielsweise müsste man bei einem Tourenplanungssystem untersuchen, ob die realisierte Auftragsverwaltung es zulässt, Aufträge eines Kunden zu erfassen, dessen Daten noch nicht in der Kundenstammdatei gespeichert wurde.
2. Datenvolumen: da das System durch ein umfangreiches Datenvolumen belastet wird, prüft man, ob es das geforderten Datenvolumen handhaben kann. Bei einem Tourenplanungssystem würde man z. B. prüfen, ob ein Programmvorgang mit den angeforderten Anzahl von Aufträgen problemlos durchgeführt werden kann.
3. Laufzeitverfahren: zu prüfen ist, ob das System bei unterschiedlichen Datenvolumina das geforderte Laufzeitverhalten zeigt. Dieses Kriterium wird auch mit effizient oder Performance umschrieben.
4. Benutzerfreundlichkeit: je nach Art des Systems und der Benutzerumgebung variieren die Eigenschaften, die es als benutzerfreundlich charakterisieren. Geeignete Testkriterien müssen daher individuell entwickelt werden. Zumindest sollte jedoch untersucht werden, ob die Benutzer schnittstelleneinheitlich konzipiert und die Fehlermeldungen für den jeweiligen Benutzer verständlich abgefragt wurden.
5. Datensicherheit/ -schutz: zu prüfen ist die Einhaltung der entsprechenden rechtlichen Bestimmungen und Sicherheitsvorgaben. Dies kann dadurch geschehen, dass man versucht durch Verwendung geeigneter Testfälle die Datensicherheits- und Datenschutzmechanismen zu unterlaufen.
6. Dokumentation: besonders wichtig ist die Untersuchung der Benutzerdokumentation hinsichtlich Genauigkeit und Klarheit da diese – wie oben erwähnt – als Grundlage für den Systemtest dienen.

Der Systemtest sollte unter Beachtung genannter Testprinzipien von einem Testteam durchgeführt werden. Mitglieder des Testteams sollten sein:

Professionelle Systemexperten,

Repräsentative Benutzer,

Ein Fachmann für Fragen der Benutzerfreundlichkeit und

Einige Mitglieder des Entwicklungsteams

Ein nach obigen Kriterien ausgeführter Systemtest ist sehr aufwendig. In der Praxis wird man daher versuchen, den Testaufwand durch Reduktion des Testumfangs zu begrenzen. Ist man zu einer solchen Begrenzung gezwungen empfiehlt es sich, folgende Leitlinien zu beachten:

es ist wichtiger, die Systemfunktionen zu testen, als Komponenten ( Module )

es ist wichtiger, alte Funktionen zu testen als neue, da man an alte bzw. bekannte Funktionen gewöhnt ist und neue Funktionen meist als Option verfügbare Verbesserungen darstellen.

es ist wichtiger, typische Situationen zu testen als Grenzwerte.

In der Literatur werden auch Meinungen geäußert, die diesen Leitlinien teils entgegenstehen. So verspricht man sich beispielsweise von Testfällen mit Grenzwerten einen höheren Testerfolg.

### **8..7. Abnahmetest**

Gegenstand des Abnahmetests ist die Überprüfung des entwickelten Produkts im Hinblick auf die Benutzeranforderungen. Der Abnahmetest ist durch den Benutzer oder zumindest unter seiner Mitwirkung vorzunehmen. Grundlagen des Abnahmetests ist die mit dem Benutzer vereinbarte oder sogar vertraglich festgelegte Leistungsbezeichnung des Systems.(Pflichtenheft)

Beim Abnahmetest festgestellte Leistungsdefizite des Systems stellen Mängeln dar, die durch Nachbesserungen zu beheben sind, davon zu unterscheiden sind Anpassungen eines Systems an geänderte Benutzeranforderungen. Sie fallen unter den Begriff der Systemwartung.

### **8..8. Installationstest**

Der Installationstest bezieht sich nicht wie die übrigen Testabschnitte auf eine bestimmte Entwicklungsphase. Vielmehr wird lediglich die korrekte Installation des gestellten und abgenommenen Systems vor seinem Gebrauch durch den Benutzer überprüft. Solch ein Test ist insbesondere dann erforderlich, wenn das System auf einen Entwicklungsrechner erstellt wurde, der von dem Computersystem des Benutzers bzw. der Benutzermaschine verschieden ist.

#### **Manuelles Testen**

### **8..9. Begriff des manuellen Testens**

Unter dem Begriff des manuellen Testens werden Methoden des nicht Computer unterstützten Testens verstanden; man spricht daher auch von „Onemantesting“. Die Erfahrung hat gezeigt, dass manuelle Test-Techniken beim Auffinden von Fehlern so effektiv sind, dass eins oder mehrere Themen beim Test eines Programmes als Objekts eingesetzt werden sollten.

Eine Anwendung manueller Testmethoden empfiehlt sich in dem Entwicklungsabschnitt, der zwischen dem Ende der Implementierung und dem Be-

ginn des computergeschützten Testens liegt. Für eine solche zeitliche Organisation des Testens sprechen zwei Gründe:

Je eher Fehler gefunden werden, desto geringer sind die Kosten für die Fehlerkorrektur und desto höher ist die Wahrscheinlichkeit, die Fehler korrekt zu beheben.

Beim Beheben von Fehlern sehen sich Programmierer dem psychologischen Druck ausgesetzt, „diesen verdammtten Bug“ (Fehler) so schnell wie möglich zu beseitigen“. Unter diesem Druck besteht die Gefahr, bei der Korrektur eines im Computertest gefundenen Fehlers mehr (neue) Fehler zu machen, als bei der Korrektur eines frühzeitig gefundenen Fehlers.

## **8..10.Methoden des manuellen Testens**

Man unterscheidet folgende manuelle Testmethoden

Code- Inspektion,

Walkthroughs und

Schreibtischtest

Code- Inspektion und Walkthroughs sind die beiden ursprünglichen Methoden des „Onemantesting“. Die Ideen zu diesen Methoden gehen auf Weinberg zurück. Das Grundprinzip beider Methoden besteht im Lesen und visuellen Überprüfen eines Programms durch ein Testteam. Eine abschließende Konferenz des Testteams hat zum Ziel, Fehler zu finden, nicht aber zu beheben.

Das Testteam sollte aus drei bis fünf Mitgliedern bestehen, wobei ein Mitglied zum Kreis der Autoren des Programms gehören sollte. Da ein Programm somit im wesentlichen von Nichtautoren getestet wird, sind Code-Inspektion und Walkthroughs wesentlich effizienter als der Schreibtischtest. Dieser stellt eine Einmanninspektion oder einen Einmann-Walkthrough durch den Autor selbst dar.

Wie Erfahrungen gezeigt haben, ist manuelles Testen im Team sehr effizient. So werden erfahrungsgemäß 30-70% der logischen Zugriffs- und Codierungsfehler gefunden. Im Durchschnitt werden, wie Experimente gezeigt haben, ca. 40% sämtlicher Fehler in Programmen mittels Inspektionen und Walkthroughs entdeckt. Bei IBM ergab die Anwendung von Codeinspektionen einen noch höheren Wirkungsgrad mit einem Fehlerentdeckungsgrad von 80%. Das bedeutet, dass 80% aller Fehler, die bis zum Testabschluss gefunden wurden, mittels Codeinspektion ermittelt wurden.

Allerdings eignen sich manuelle Testmethoden zum Auffinden bestimmter Fehlerarten weniger gut als computergeschützte Tests. Dazu gehören z. B. „High-Level“-Entwurfsfehler, die beispielsweise bei der Analyse von Anforderungen entstanden sind.

## Code - Inspektion

Bei einer Code - Inspektion liest ein Team gemeinsam einen vorgegebenen Programmtext mit der Absicht durch, Fehler zu entdecken. Das Inspektionsteam sollte drei bis vier Mitglieder mit folgenden Rollen bzw. Herkunft umfassen:

1. Ein Moderator, der zwar kompetenter Programmierer ist, aber nicht zum Kreis der Programmautoren gehört.
2. der Programmautor bzw. ein Mitglied der Gruppe der Programmautoren.
3. der Programmdesigner, falls er nicht zugleich zur Gruppe der Programmautoren gehört,
4. Ein Testspezialist

Der Autor ist verantwortlich für die Verteilung der Unterlagen die zeitliche und räumliche Organisation der Inspektionssitzung in die Protokollierung der gefundenen Fehler. Außerdem fungiert er als Sitzungsleiter.

Zu der Sitzung entwickeln die Mitglieder, die Unterlagen ( Programmliste, Entwurfspezifikation) rechtzeitig erhalten und studiert haben, im wesentlichen zwei Aktivitäten:

- Nachvollzug der Programmlogik, die von einem der Programmautoren Anweisung für Anweisung erklärt wird. Einige Fehler werden vom Vortragenden während des Vortrags selbst entdeckt. Andere ergeben sich aus der Weiterverfolgung von Fragen, die beim Vortrag aufkommen.
- Das Programm wird anhand einer Checkliste analysiert, in der bekannte Fehler nach Gruppen zusammengestellt sind.

Während der Sitzung hat der Moderator dafür zu sorgen, dass sich die Teilnehmer auf die Entdeckung von Fehlern konzentrieren und nicht auf ihre Korrektur. Diese haben Programmautoren später vorzunehmen.

Nach der Sitzung sind zwei weitere Aktivitäten durchzuführen

1. Übergabe einer Liste der gefundenen Fehler an die Programmautoren und evtl. erneute Überprüfung des korrigierten Programms. Dies kann dann der Fall sein, wenn relativ viele oder komplexe Fehler gefunden wurden
1. Analyse der Fehlerliste und Ergänzung bzw. Verfeinerung der Fehler - Checkliste, um die Wirksamkeit künftiger Inspektionen zu verbessern

Eine Code – Inspektion sollte unter folgenden Bedingungen durchgeführt werden:

Geeignete Wahl von Durchführungszeit und Räumlichkeit, um Störungen zu vermeiden.

Begrenzung der Sitzungsdauer auf 90-120 Minuten, um der in Zeitablauf nachlassenden Konzentration der Teilnehmer gerecht zu werden.

Verteilung der Code- Inspektion auf mehrere Sitzungen, da durchschnittlich pro Stunde lediglich ca. 150 Statements inspiziert werden können.

Wahl der Gesprächsform so, dass der Programmautor die Entdeckung von Fehlern nicht als Angriff seiner Person betrachtet.

Der Programmierer sollte möglichst eine „ego-less“ Haltung einnehmend das Inspektionsziel in den Vordergrund stellen.

Vertrauliche Behandlung der Inspektionsergebnisse.

Neben dem Haupteffekt Fehler zu entdecken, bewirkt die Code- Inspektion noch vorteilhafte Seiteneffekte:

Der Programmautor erkennt eigene Fehler und lernt daraus.

Der Programmautor erhält eine Rückkopplung bezüglich Programmierstil und Programmiertechnik,

Analyse Lerneffekte treten auch bei den übrigen Team-Mitgliedern auf.

Eine komprimierte Fehlerprüfliste für die Code- Inspektion ist nachfolgend wiedergegeben.

Datenreferenz	Steuerfluß
1. Wurden Variable angesprochen.,die nicht initialisiert wurden?	1. Wurden bei Mehrfachentscheidungen alle Sprungmöglichkeiten berücksichtigt?
2. Befinden sich die Indizes innerhalb Grenzen?	2. Wird jede Schleife beendet?
3. Wurden nichtganzzahlige Indizes verwendet?	3. Wird das Programm beendet?
4. Hat der Zeiger einen noch gültigen Wert (dangling reference)?	4. Wird eine Schleife aufgrund der Eingangsbedingung nicht ausgeführt?
5. Sind die Attribute bei einer Redefinition korrekt?	5. Sind möglich Umgehungen von Schleifen korrekt?
6. Stimmen Satz. und Strukturattribute überein?	6. Gibt es bei einer Iteration "off by one" Fehler?
7. Passen die Adressen der Bitketten? Wurden Bitkettenattribute übergeben?	7. Gehören die DO/END-Anweisungen zusammen?
8. Sind die Attribute für den „based“ Speicherbereich korrekt?	8. Gibt es unvollständige Entscheidungen?
9. Stimmen die Datenstrukturen in verschiedenen Prozeduren überein?	<b>Schnittstellen</b>
10. Wurden die Indexgrenzen einer Kette	1. Stimmt die Anzahl der Eingabeparamete-

überschritten?	ter mit der Anzahl der Argumente überein?
11. Gibt es "off by one" Fehler?	2. Stimmen die Parameter- und Argumentattribute überein?
<b>Datendeklaration</b>	3. Stimmen die Einheiten der Parameter und der Argumente überein?
1. Sind all Variablen erklärt?	4. Stimmt die Anzahl der an dem gerufenen Modul übergebenen Argumente mit der Anzahl der Parameter überein?
2. Sind die Standardattribute verstanden worden?	5. Stimmen die Attribute der an dem gerufenen Modul übergebenen Argumente mit den Attributen der Parameter überein?

3. Sind Felder und Ketten richtig initialisiert?	6. Entsprechen die Einheiten der dem gerufenen Modul übergebenen Argument den Einheiten der Parameter?
4. Sind die korrekten Längen, Typen und Speicherklassen zugewiesen?	7. Sind Anzahl, Attribute und Reihenfolge der Argumente für „eingebaute“ Funktionen korrekt?
5. Paßt die Initialisierung zu der Speicherklasse?	8. Gibt es Referenzen auf Parameter, die nicht mit dem aktuellen Entry-point assoziiert sind?
6. Gibt es Variablen mit ähnlichen Namen?	9. Werden Argumente verändert, die nur als Input dienen sollten?
<b>Berechnungen</b>	10. Ist die Definition globaler Variabler über alle Module konsistent?
1. Werden Rechnungen mit nichtarithmetisch Variablen durchgeführt?	11. Werden Konstante als Argumente übergeben?
2. Gibt es Rechnungen mit verschiedenen Datentypen?	<b>Ein-/Ausgabe</b>
3. Gibt es Rechnungen mit Variablen verschiedener Länge?	1. Sind die Dateiattribute korrekt?
4. Ist die Größe der Zielvariablen kleiner als die des zugewiesenen Werts?	2. Ist das OPEN-Statement korrekt?
5. Gibt es einen Ober- oder Unterlauf im Zwischenergebnis?	3. Passen die Formatspezifikationen zu den E/A-Anweisungen?
6. Division durch Null?	4. Passen die Puffergröße zu den Satzgrößen?
7. Treten Ungenauigkeiten bei der Dualdarstellung auf?	5. Wird die Datei vor Benutzung geöffnet?
8. Liegt der Variablenwert innerhalb eines Sinnvollen Bereichs?	6. Werden Dateiendebedingungen behandelt?
9. Ist die Priorität der Operatoren richtig verstanden worden?	7. Werden E/A-Fehler behandelt?
10. Ist die Division mit ganzzahligen Werten korrekt?	8. Gibt es Textfehler in der Ausgabeinformation?
<b>Vergleich</b>	<b>Andere Prüfungen</b>
1. Wird ein Vergleich zwischen inkonsistenten Variablen durchgeführt?	1. Gibt es nicht angesprochene Variablen in der Crossreference-Liste?
2. Wurden Daten verschiedenen Typs miteinander verglichen?	2. Bringt die Attributliste das, was man erwartet?
3. Sind die Vergleichsoperatoren korrekt?	3. Gibt es warnende oder informative Meldungen?

4. Sind die Boole'schen Ausdrücke korrekt?	4. Werden die Eingabedaten auf Gültigkeit überprüft?
5. Sind Vergleichs- und Boole'sche Ausdrücke vermischt?	5. Fehlen Funktionen?
6. Gibt es Vergleiche mit Brüchen zur Basis 2?	
7. Ist die Priorität der Operatoren richtig verstanden worden?	
8. Ist die Compilerdarstellung der Boole'sche-Ausdrücke richtig verstanden worden?	

## Walkthroughs

Der Codewalkthrough ist ein der Code-Inspektion ähnliches Verfahren. Unterschiede gibt es jedoch vor allem bei der Vorgehensweise der Fehlerentdeckung während der ein- bis zweistündigen Sitzungsdauer des Testteams.

Das Team besteht aus drei bis fünf Mitglieder und setzt sich wie folgt zusammen:

1. Ein Moderator mit ähnlichen Aufgaben wie der Code-Inspektion
2. Ein Sekretär, der alle entdeckten Fehler protokolliert.
3. Ein „Tester“ mit speziellen, intern genannten Aufgaben.
4. Evtl. weitere Mitglieder

Die Teammitglieder sollten sich aus folgendem Personenkreis rekrutieren: Programmautor ( auf jeden Fall Teammitglied), erfahrener Programmierer, Mitarbeiter der das System später warten soll, Mitarbeiter aus anderen Projekten.

Ebenso wie bei der Code-Inspektion erhalten die Teammitglieder die Sitzungsunterlagen einige Tage vor der Sitzung und bereiten sich durch Studium der Unterlagen auf die Sitzung vor. Das Vorgehen während der Sitzung ist von dem in der Inspektion aber verschieden. Es stellt sich wie folgt dar:

Das Programm wird nicht gemeinsam gelesen und es werden auch keine Prüflisten abgearbeitet; vielmehr bereitet der „Tester“ einige Testfälle auf dem Papier vor. Die Teilnehmer spielen die Testfälle durch, indem sie mit den Testdaten das Programm im Geiste ausführen. Zur Unterstützung dieses Vorgehens wird der Programmzustand z. B. auf einer Wandtafel festgehalten.

Die Auswahl der Testfälle sollte eher gering sein und die Testfälle sollte aus repräsentativen Eingaben und erwarteten Ausgaben bestehen. Das Durchspielen der Testfälle soll als Anreiz dienen, den Programmautor über die Programmlogik und die zugrundeliegende Annahmen zu befragen. Wenn man über diese Thema diskutiert , werden normalerweise mehr Fehler entdeckt, als beim Durchspielen der Testfällen.

Im Hinblick auf Seiteneffekte und Durchführungsbedingungen gelten die bei der Code-Inspektion gemachten in analoger Weise.

## Schreibtischtest

Der Schreibtischtest kann in Form einer Einmaninspektion oder eines Einmanwalkthrough durchgeführt werden. Dabei liest der Tester sein eigenes Programm und überprüft es mit Hilfe einer Fehlerliste, oder er sinniert mit Testdaten einen Testfall.

Der Schreibtischtest ist insbesondere dann unproduktiv, wenn der Tester zugleich auch der Programmautor ist. Tauscht man Programme jedoch zwischen Programmautoren aus, so lässt sich eine höhere Effektivität erzielen. Die dennoch bestehenden Effektivitätsnachteile gegenüber Codeinspektion und Walkthrough resultieren auch aus dem Fehlen der im Team bezüglich des Entdeckens von Fehlern gegebenen Konkurrenzsituation.

## Computergestütztes Testen

Ordnet man die hier im Vordergrund stehenden konventionellen Methoden des Testen in die umfassendere Gruppe der Methoden zum Überprüfen von Programmen ein, so kann man folgende Untergliederung der Überprüfungsmethoden vornehmen.

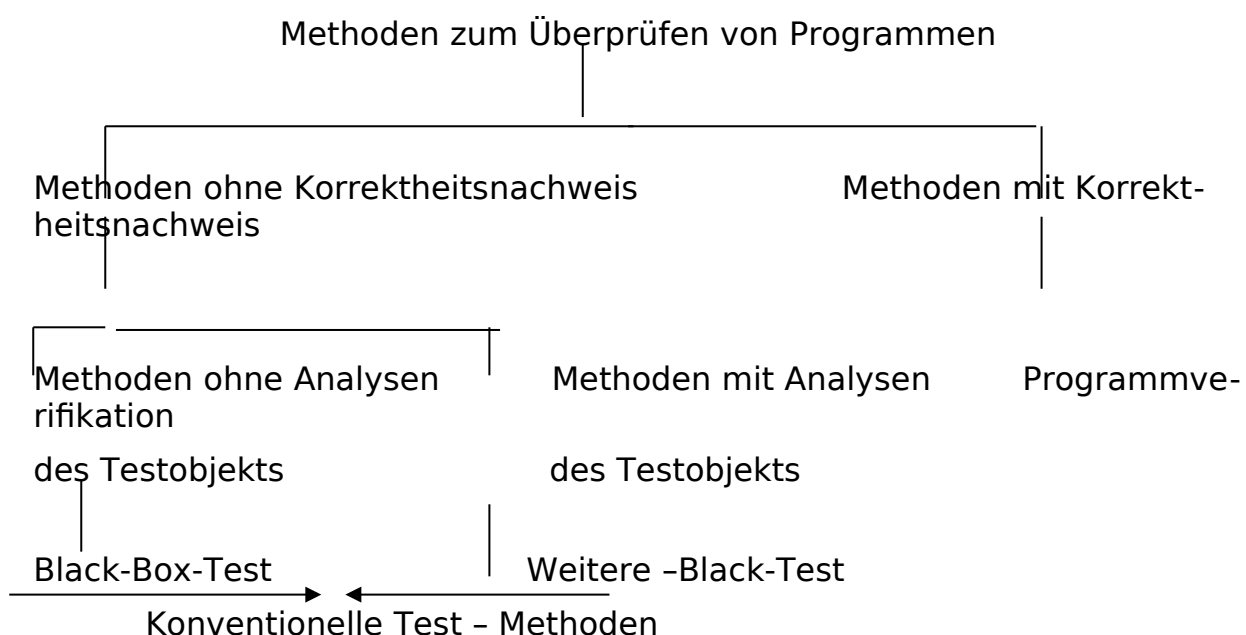


Abb. 7.6: Großübersicht über Methoden zum Überprüfen von Programmen



### 8..11.White - Box - Test

Beim White-Box-Test, auch Strukturtest genannt, wird die innere Struktur eines Testobjektes (Modul, Programm) in Kenntnis und unter Verwendung der inneren Logik überprüft. Die Überprüfung kann statisch oder dynamisch vorgenommen werden. Insgesamt ergeben sich die in Abb. 7.7 dargestellten Formen des White-Box-Tests.

Die statische Analyse besteht in einer Art Review des Quellcodes und ist somit nicht mit einer Programmausführung verbunden. Ziel der statischen Analyse ist das Finden von Fehlern bei der Vornahme von Überprüfungen wie folgt:

1. Nachweis der Verwendung von Programmelementen ( beispielsweise mit Hilfe von Cross-Reference -Listen)

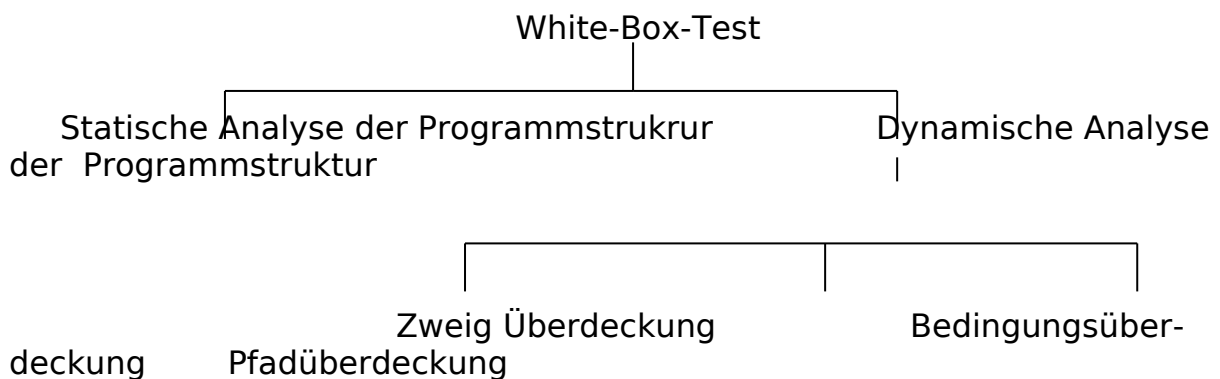


Abb. 7.7: Formen des White-Box-Tests

2. Nachweis der Verwendung aller vereinbarten Datenobjekte
3. Nachweis der Erreichbarkeit aller Anweisungen (kann beispielsweise mit Hilfe von Vorgänger-Nachfolger-Tabelle gesehen werden)
4. Überprüfung der Initialisierung von Datenobjekten ( vor ihrer Verwendung oder ohne ausschließende Verwendung)
5. Prüfung auf Vorhandensein von Endlosschleifen.
6. Prüfung auf typengerechte Verwendung von Datenobjekten.
7. Prüfung auf Konsistenz von Schnittstellen

Der tatsächliche Umfang der( manuellen) statischen Analyse hängt vom Leistungsumfang des verwendeten Compilers ab. Bei neueren Sprachen bzw. Compilern werden eine ganze Reihe der genannten Prüfungen automatisch durchgeführt. Von Interesse ist die statische Analyse also besonders bei Verwendung älterer Sprachen bzw. Compiler.

Bei der dynamischen Analyse überprüft man die innere Struktur des Testobjektes durch Ausführung des Instruments Testobjekt mit Testfällen. Die Testfälle werden aus der Struktur des Programms abgeleitet. Aus der Analyse der Programmstruktur ergibt sich auch, wie die Instrumentierung vorzunehmen ist. Instrumentierung bedeutet Einbau von Zählern an geeigneten Programmstellen. Beim Ausführen eines Testfalls werden die jeweils durchlaufenen Zähler initialisiert oder höher gesetzt. Mit dieser Instrumentierung kann man folgenden Zwecke anstreben:

1. Prüfung auf Zweigüberdeckung: es wird festgestellt, ob bei Ausführung der Testfälle jeder Zweig des Testobjekts mindestens einmal durchlaufen wurde.
2. Prüfung auf Pfadüberdeckung: Es wird festgelegt, ob bei Ausführen der Testfälle jeder mögliche Pfad auch tatsächlich durchlaufen wurde.
3. Prüfung auf Bedingungsüberdeckung: es wird festgelegt, ob bei Ausführung der Testfälle sämtlichen Auswahlmöglichkeiten alle möglichen Zustände angenommen haben.

Die Überprüfung auf Pfadüberdeckung ist dann problematisch, wenn das Testobjekt Wiederholungen enthält. In Verbindung mit unverschlossenen Auswahl-Konstrukten kann dann selbst bei kleinen Programmen die Auswahl der Pfade sehr groß werden. Bedingungs- und Zweigüberdeckung werfen dagegen weniger Probleme auf. In der Literatur wird eine Überdeckung sämtlicher Zweige auch als Minimalkriterium für den Test genannt.

#### Beispiel White-Box-Test:

Betrachtet sei ein ganz einfacheres Testobjekt, das aus einer zweiseitigen Auswahl besteht:

```
IF (Zeichen = „A“) OR ( Zeichen =“I“)  
OR (Zeichen = „O“) OR (Zeichen =“U“)  
THEN WRITE („VOKAL“)  
ELSE WRITE (“KONSONANT”)
```

Bei der Bedingungsüberdeckung werden folgende Testfälle benötigt: “A” “E” “I” “O” “U” und z. B. „B“.

Dagegen genügen bei der Zweigüberdeckung folgende Testfälle, z. B.: „A“ und „B“

## **8..12.Black - Box - Test**

Beim Black-Box-Test, auch datengetriebenes Testen oder Ein-/Ausgabe-Testen genannt, wird das Testobjekt ( Programm, Modul) als schwarzer Kasten betrachtet. Das Innere des Kastens , insbesondere die innere logi-

sche Struktur, ist dem Tester nicht bekannt und für ihn auch nicht von Interesse.

Bekannt und für den Test auch von Interesse sind dagegen :

die Schnittstelle des Testobjekts (z. B. Modul-Schnittstelle),

die Funktion des Testobjekts (z. B. Exportierte und interne Prozeduren eines Moduls).

Schnittstellen- und Funktionsbeschreibung gehen aus der Spezifikation des Testobjekts hervor (z. B. Modulbeschreibung)

In Kenntnis der Spezifikation des Testobjekts versucht der Tester durch die Eingabe und Ausführung von Testfällen Umstände zu entdecken, bei denen sich das Testobjekt nicht gemäß der Spezifikation verhält. Dies geschieht bei einem Testfall konkret durch den Vergleich der Ausgabe mit der erwarteten Ausgabe.

Die Testfälle bzw. Testdaten lassen sich beim Black-Box-Test nicht aus der inneren Struktur des Testobjekts ableiten. Sie sind vielmehr lediglich aus der Spezifikation abzuleiten. Die Zusammenstellung geeigneter Testfälle ist nicht unproblematisch. Sollen alle Fehler eines Testobjekts entdeckt werden, müsste ein vollständiger Eingabetest durchgeführt werden. Es müssten also alle möglichen Eingangsbedingungen zur Definition von Testfällen herangezogen werden und zusätzlich müssten alle möglichen Reaktionen des Testobjekts angegeben werden.

Man kann schon vermuten, dass bei einem vollständigen Eingabetest die Auswahl der Testfälle einen astronomischen Umfang annehmen kann. Um beispielsweise einen Compiler zu testen, müsste man alle möglichen korrekten, aber auch inkorrekten Programme eingeben. Bei Testobjekten mit Gedächtnis, wie in einem Flugbuchungsprogramm, sind die Reaktionen des Testobjekts von den vorangegangenen Transaktionen abhängig. Daraus folgt, dass bei einem vollständigen Test auch alle möglichen Sequenzen von Transaktion durchprobiert werden müssten.

Obige Ausführungen lassen erkennen, dass bei nichttrivialen Testobjekten ein vollständiger Eingabetest nicht durchführbar ist. Beim Black-Box-Test kommt es also darauf an

die Auswahl der Testfälle durch eine sinnvolle Auswahl zu begrenzen

die Testfälle so auszuwählen, dass eine möglichst große Wirkung (Entdeckung möglichst vieler Fehler) erzielt wird

Eine Beschränkung der Testfälle in diesem Sinn setzt voraus, dass man in der Lage sein müsste, bestimmte Aussagen über das Verhältnis zwischen Testfällen zu machen. Gemeint ist damit das gleiche Verhalten des Testobjekts für mehrere Testfälle. Kann man derartige Aussagen treffen, so ist damit ein Ansatzpunkt für die Begrenzung des Testumfangs gegeben. Methoden zur Auswahl von Testfällen werden im folgenden Kapitel behandelt.

Zusammenfassend lässt sich die Vorgehensweise beim Black-Box-Test schematisch darstellen, wie in der nachfolgenden Abb. gezeigt. Basis für die Testfallermittlung ist die Spezifikation des Testobjekts. Die Haupt-

schwierigkeit besteht darin, geeignete, d.h. mit großer Wahrscheinlichkeit Fehler entdeckende, Testfälle abzuleiten.

### Testdaten-Auswahl

Die Methoden zur Ableitung von Testfällen beim Black-Box-Test lassen sich in zwei Gruppen einteilen:

Methoden ohne Berücksichtigung von Eingabekombinationen,

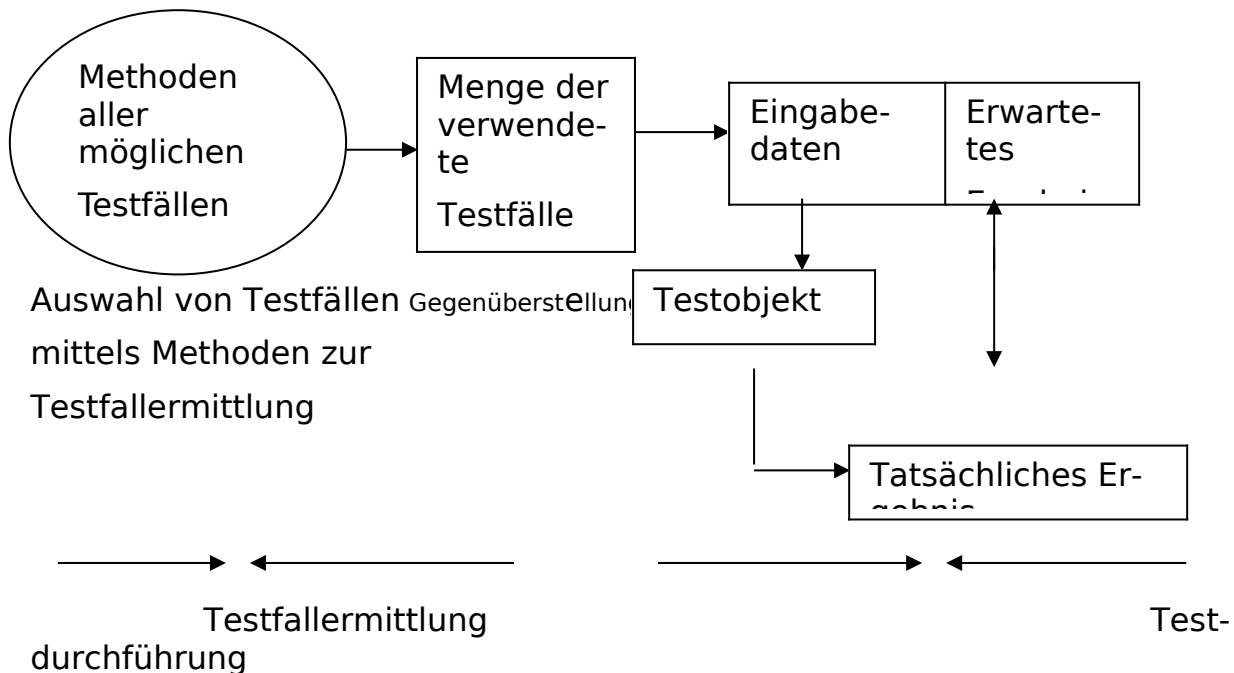


Abb. 7.7: schematische Darstellung der Vorgehensweise beim Black-Box-Test

Methoden mit Berücksichtigung von Eingabekombinationen

Bei der letzten genannten Gruppe von Methoden wählt man Kombinationen von Eingabebedingungen. Auch dies ist keine leichte Aufgabe, da die Auswahl der Kombinationsmöglichkeiten sehr hoch sein kann. Eine Übersicht über die Methoden beider Gruppen zeigt die

Methoden zur Testfallbestimmung beim Black-Box-Test

Testfallbestimmung ohne Eingabekombinationen  
mit Eingabekombination

Testfallbestimmung

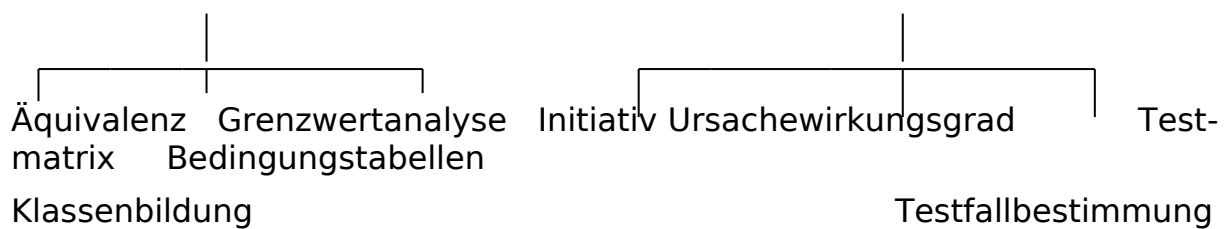


Abb. 7.8 Methoden zur Testbestimmung beim Black-Box-Test

## Testfallbestimmung ohne Eingabekombinationen

### 8..12..1 Äquivalenzklassenbildung

Ausgehend von der Spezifikation des Testobjekts unterteilt man bei dieser Methode alle möglichen Eingabewerte in Äquivalenzklassen. Eine Äquivalenzklasse bezeichnet hierbei eine Untermenge von Eingabewerten, für die gilt:

Für einen Vertreter der Klasse verhält sich das Testobjekt in der gleichen Weise, wie bei allen anderen Vertretern der Klasse.

Läuft also ein Programm für einen beliebigen Wert einer solchen Klasse korrekt, läuft es für alle entsprechend.

Äquivalenzklassen kann man wie folgt unterteilen:

- Äquivalenz für Eingabewert und Äquivalenzklassen für Ausgabewert
- Äquivalenzklassen für gültige Testfälle (d. h. mit zulässigen Ein- bzw. Ausgabewerten) und Äquivalenzklassen für ungültige Testfälle (d. h. mit unzulässigen Ein- bzw. Ausgabewerten)

Bei Testlegung von Äquivalenzklassen sollte man folgende Regeln beachten:

1. Stellt ein Eingabetyp einen Wertbereich dar (z. B. MONATE = 1..12) dann sind eine gültige Äquivalenzklasse ( $1 \leq \text{MONATE} \leq 12$ ) und zwei Äquivalenz ( $< 1$  und  $\text{Monate} < 12$ ) zu wählen.
2. Besteht ein Eingabetyp aus einem Aufzählungstyp (z. B. AMPEL-FÄRBE = Rot, Gelb, Grün) und kann man annehmen, dass jeder Fall unterschiedlich behandelt wird, so ist für jeden einzelnen Fall eine Äquivalenzklasse zu bilden, sowie eine Äquivalenzklasse für ungültige Testfälle (z. B. dunkelgelb).
3. Beschreibt ein Eingabetyp eine Voraussetzung (z. B. eine Real-Zahl muss einen Dezimalpunkt enthalten), dann ist eine Äquivalenzklasse für gültige (Dezimalpunkt vorhanden) und für ungültige Testfälle (Dezimalpunkt fehlt) zu bilden.

Sind alle Äquivalenzklassen bezüglich eines Testobjekts gebildet worden, so sucht man sich für die Testdurchführung aus jeder Klasse irgendeinen Repräsentanten aus.

Beispiel: Fakultätsberechnung

Nachfolgend ist die Spezifikation eines Moduls zur Berechnung von  $n!$ ,  $0 < n < 13$  angegeben [Bal, s. 41569] :

FUNKTION  $\text{nfak}(n; \text{INTEGER}) : \text{INTEGER};$

FURPOSE Berechnung der Fakultät:  $n! = (n-1) \times \dots \times 3 \times 2 \times 1$

CONDITION  $n \leq 0 \text{ AND } n < 13$

STATES

1 WHEN  $n = 0$  OK  $n = 1$

2 WHEN  $n = 2$

$n!$  WHEN  $n$

-1 WHEN  $n < 0$

POSITION  $\text{nfak} \leq 1$

END  $\text{nfak}$ .

Bei diesem Beispiel gibt es drei Äquivalenzklassen:

- (1) Eine Äquivalenzklasse für  $n \leq 0$  und  $n < 13$ .
- (2) Eine Äquivalenzklasse für  $n < 0$ .
- (3) Eine Äquivalenzklasse für  $n \leq 13$ .

Entsprechend wurde man für den Test Moduls je einen Testfall aus jeder der Klasse wählen, also sich:

- Testfall für Klasse (1):  $n = 12$ , Sollergebnis:  $\text{nfakt} = 479\,001\,600$ .
- Testfall für Klasse (2):  $n = -11$ , Sollergebnis:  $\text{nfak} = -1$ .
- Testfall für Klasse (3):  $n = 25$ , Sollergebnis: Overflow.

Die Bildung von Äquivalenzklassen ist dem Verwenden zufällig gebildeter Testfälle überlegen; dennoch können wichtige Arten von Testfällen übersehen werden. Dies sind insbesondere solche Testfälle, wie sie z. b. mit der Grenzwertanalyse ermittelt werden.

## **8.12.2 Grenzwertanalyse**

Die Erfahrung hat gezeigt, dass Testfälle, die die Grenzwerte von Äquivalenzklassen abdecken oder in der unmittelbaren Umgebung dieser Grenzwerte liegen, besonders effektiv sind. Eine Grenzwertanalyse ist jedoch nur dann durchführbar, wenn die zu prüfende Äquivalenzklasse aus Elementen besteht, die sich auf natürliche Weise ordnen lassen. So lassen sich MONATE natürlich ordnen, nicht aber AMPELFARBEN. Testfälle werden demnach wie folgt gebildet:

- Aus der jeweiligen Äquivalenzklasse wird nicht ein beliebiges Element verwendet. Genommen werden vielmehr nur solche Elemente, die sich in unmittelbarer Nähe der Klassengrenzen befinden.
- Eine Annäherung an die Grenzen wird beidseitig durchgeführt, d. h. sowohl von der Seite der gültigen Testfällen, als auch von der Seite der ungültigen Testfällen.

Für das Beispiel der Fakultätsberechnung lassen sich folgende Grenzwerttests bilden:

- Testfall für einen gültigen Grenzwert:  $n = 0$ , Sollergebnis:  $nfak = 1$
- Test für einen ungültigen Grenzwert:  $n = -1$ , Sollergebnis:  $nfak = 0$

### **8..12..3 Intuitive Testfallermittlung**

Testfälle werden hier nicht nach einer vorgesehenen Methode ermittelt, sondern intuitiv auf der Basis von Erfahrungen formuliert. Eine Leitidee der intuitiven Testfallermittlung besteht darin, eine Liste möglicher Fehler oder Fehlersituationen aufzustellen und daraus Testfälle abzuleiten. Häufig wird eine solche Liste Spezialfälle enthalten, die möglicherweise bei der Spezifikation übersehen wurden.

Beispiele für derart ermittelte Testfälle sind:

- (1) Ein Testfall mit dem Wert Null als Ein- oder Ausgabewert, die Ein- und Ausgabedaten mit diesem Wert zeigen oft eine Fehlersituation an.
- (2) Testfälle mit Neuer- oder Sonderzeichen bei der Verarbeitung von Zeichenketten, denn solche Zeichen sind meist für bestimmte Zwecke reserviert.
- (3) Testfälle, die bei einer Tabellenverarbeitung die Fälle „keinen Eintrag“ oder „ein Eintrag“ vorsehen, denn diese stellen i. d. Regel Sonderfälle dar.

### **Testfallbestimmung mit Eingabekombinationen**

Ein Nachteil der behandelten Methoden zur Testfallbestimmung besteht in der Nichtberücksichtigung von Eingabekombinationen. Die im folgenden angesprochen Methoden beheben diesen Nachteil in gewissen Umfang.

### **8..12..4 Ursache-Wirkung-Graph**

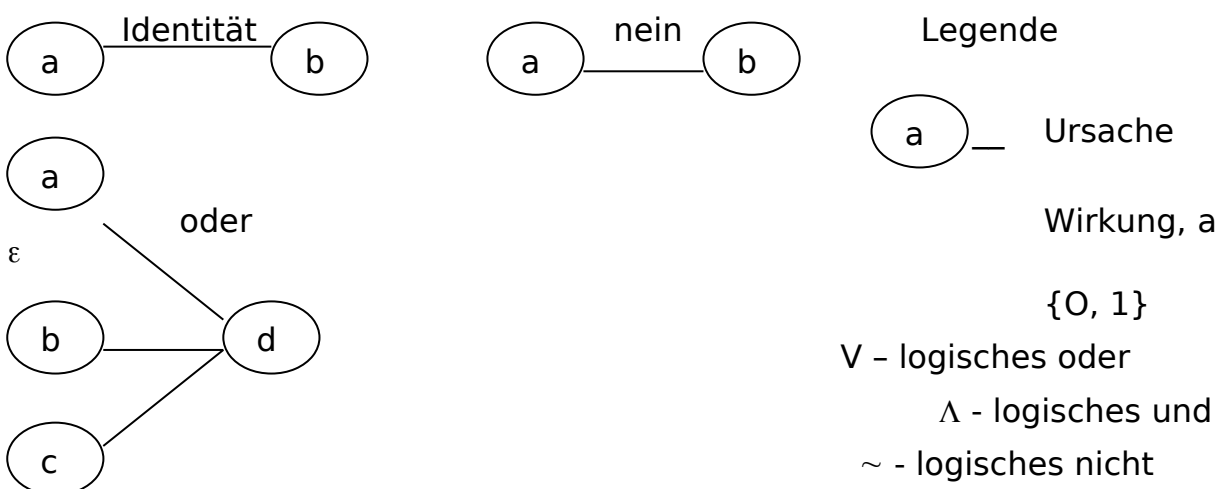
Der Ursache-Wirkung-Graph (UWG) ist eine Methode zur Definition von Testfällen, die sich aus Kombinationen von Eingabebedingungen ergeben.

Darüber hinaus ist der UWG geeignet, Hinweise auf Unvollständigkeiten und Zweideutigkeiten der Spezifikation abzuleiten.

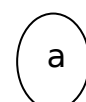
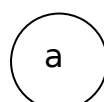
Methodisch gesehen stellt der UWG eine formale Sprache dar, in die eine in natürlicher Sprache gegebene Spezifikation des Testobjekts übersetzt wird. Die Erstellung eines UEG für ein Testobjekt und die Ableitung von Testfällen mit Hilfe des UWG vollzieht sich in folgenden Schritten:

- (1) Zerlegen der Spezifikation des Testobjekts anhand habbarer Teile, damit der UWG nicht zu komplex wird.
- (2) Festlegen der Ursachen und Mitwirkungen einer (Teil) Spezifikation. Eine Ursache ist eine bestimmte Eingabebedingung oder eine Äquivalenzklasse von Eingabebedingungen. Eine Wirkung ist eine Ausgabebedingung oder eine Systemtransformation. Jede Ursache und jede Wirkung wird durch eine eindeutige Zahl gekennzeichnet.
- (3) Analysieren des semantischen Inhalts der Spezifikation und Übersetzen des Analyseergebnisses in einen Booleschen Graphen, der die Ursachen und Wirkungen miteinander verbindet. Dieser Graph stellt den Ursachen-Wirkungs-Graphen dar.
- (4) Angaben von syntaktischen oder kontextabhängigen Beschränkungen mit Hilfe geeigneter Symbole im UWG.
- (5) Erzeugen einer Entscheidungstabelle aus dem UWG durch Verfolgen der einzelnen Zustandsbedingungen im UWG. Jede Spalte der Entscheidungstabelle repräsentiert einen Testfall.
- (6) Umwandeln der Spalten der Entscheidungstabelle in Testfälle.

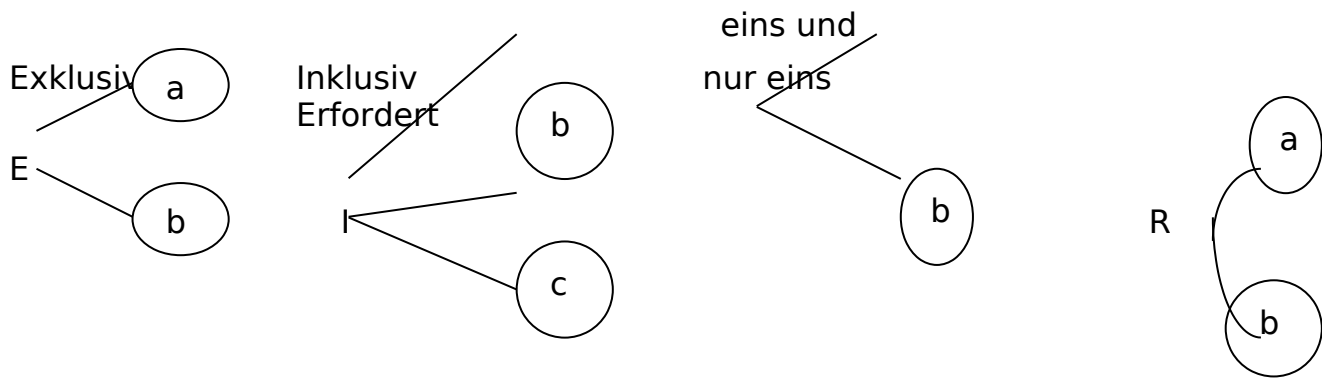
Zur Darstellung von Ursachen-Wirkungs-Graphen werden die in Abb. 7.9 gezeigten Grundsymbole und einschränkenden Symbole verwendet.



a) Grundsymbole







### b) Einschränkende Symbole

Abb.7.9 : Grundsymbole und Einschränkende Symbole zur Darstellung von Ursachen\_Wirkung\_Graphen

#### Beispiele:

##### Als Zeichen

Für die Erzeugung eines UWG soll folgende Spezifikation benutzt werden:

Das Zeichen in Spalte 1 muss A oder B sein. Das Zeichen in Spalte 2 muss eine Ziffer sein. Unter diesen Umständen wird die Ergänzung einer Datei durchgeführt. Ist das erste Zeichen nicht korrekt so wird die Meldung X12 ausgegeben. Ist das zweite Zeichen keine Ziffer, so wird die Meldung X13 ausgegeben.

Die Ursachen sind bei diesen Beispiel:

- (1) Zeichen in Spalte 1 ist ein „A“
- (2) Zeichen in Spalte 1 ist ein „B“
- (3) Zeichen in Spalte 2 ist eine Ziffer.

Die Wirkungen sind:

- (1) Ergänzen der Datei wird durchgeführt.
- (2) Die Meldung X12 wird ausgegeben.
- (3) Die Meldung X13 wird ausgegeben.

Beim Erstellen des UWG ist zu beachten, dass sie bei den Ursachen 1 und 2 nicht gleichzeitig sein können. Sie können aber beide nicht gesetzt sein. Im UWG sind sie daher mit der Einschränkung F zu verbinden.

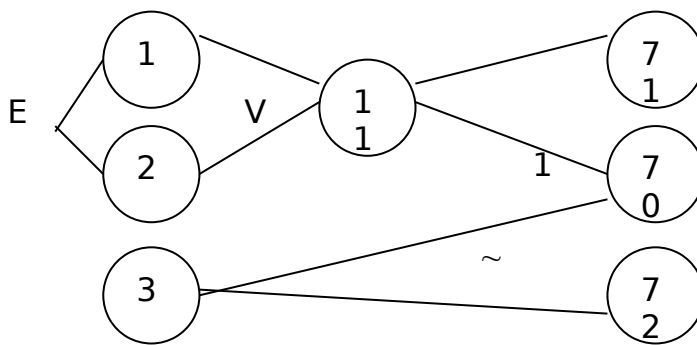


Abb. 7. 10 Beispiel für einen Ursache-Wirkung-Graph.

Die schwierige Umsetzung eines UWG in eine Entscheidungstabelle, aus der die Testfälle abgeleitet werden können, wird hier nicht behandelt. Sie kann automatisch mit Hilfe geeigneter Analysenprogramme durchgeführt werden.

#### 8..12..5 Testmatrix

Unter Testmatrix versteht man eine matrixartige Zuordnung von zu testenden Funktionen des Testobjekts und Testfällen. Eine Matrixdimension wird durch die zustehenden Funktionen gebildet und in der anderen Dimension sind die Testfälle dargestellt. Diese Darstellungsart lässt die Abdeckung der einzelnen Funktionen durch Testfälle bzw. Testfälle, die Kombination von Funktionen repräsentieren, gut erkennen.

#### 8..12..6 Bedingungstabelle

Bedingungstabellen sind ähnlich wie Entscheidungstabellen aufgebaut. Sie dienen dem übersichtlichen Darstellen von Bedingungen bzw. Kombinationen des Testobjekts. Dabei ist vor allem an solche Funktionen gedacht, die für die Lösung des jeweiligen Problems und die korrekte Ausführung des Programms bedeutsam sind.

Glossar

*Raum für Ihre Notizen*

## 9. Index

Im Index werden Hinweise auf Hauptüberschrift zu einem Begriff in Fettbuchstaben dargestellt. Hinweise auf im Script enthaltene Übungen sind kursiv dargestellt.

### A

Abgrenzungskriterium.....4-15  
 Abnahmephase.....2-9  
 Analyse.....2-7, **4-13**

### B

Benutzungsoberfläche.....4-17  
     Gestaltung.....5-53  
 Betriebsbedingungen.....4-16

### D

Design.....2-8, **6-67**  
 Dokumentation.....2-8

### E

Einführungsphase.....2-9  
 Entwurf.....2-8  
     objektorientierter.....**6-67**  
 Ergänzungen.....3-10

### I

Implementierung.....2-8

### L

Lastenheft.....2-7, **3-10**  
     Beispiel.....3-11  
     Übung.....3-12

### M

Musskriterium.....4-15

### N

Notation des OOA-Modells.....**4-23**  
     Basiskonzept.....4-23  
     dynamisches Konzept.....4-38  
     statisches Konzept.....4-32

### O

objektorientierte Analyse.....**4-13**  
 OOA.....**4-13**  
 OOA-Modell.....**4-21**  
     dynamisches Modell.....4-22  
     Erstellung.....4-22  
     statisches Modell.....4-22  
 OOD-Modell.....**6-67**  
     dynamisches Modell.....6-69  
     statisches Modell.....6-69

### P

Pflegephase.....2-9  
 Pflichtenheft.....2-7, **4-14**  
     Beispiel.....4-18  
     Übung.....4-21  
 Produktdaten.....3-10, 4-16  
 Produkteinsatz.....3-10, 4-15  
 Produktfunktionen.....3-10

Produktleistung.....3-10, 4-17  
 Produktschnittstelle.....4-16  
 Produktumgebung.....4-16  
 Prototyp.....**4-22**

---

**Q**

Qualitätsanforderungen.....3-10  
 Qualitätsbestimmung.....4-17

---

**T**

Testfälle.....4-17

Testphase.....2-8  
 Testszenarien.....4-17

---

**W**

Wartungsphase.....2-9  
 Wunschkriterium.....4-15

---

**Z**

Zielbestimmung.....3-10  
 Zielgruppe.....4-15