Grundlagen Objekt-orientierter Modellierung

Analyse und Design mit UML

O.Univ.-Prof. Dipl.-Ing. Dr. Wolfgang Pree

www.SoftwareResearch.net

OO Analyse- und Design-Tools

In OO gesetzte Erwartungen

- Verbesserte Modularisierung
- Verbesserte Wiederverwendbarkeit
  - Potential von wiederverwendbaren OO SW-Architekturen
    (= generischen komplexen Komponenten)
    wurde bisher keineswegs ausgeschöpft
- Durchgängiges Denkmodell
  - Analyse → Design → Implementierung

Wichtige Grundlage: Unterstützung der OO Modellierung

Was ist von OOAD-Tools zu erwarten? (I)

SOFTWARE RESEARCH LAB

Great designs come from great designers, not from great tools.

Tools help bad designers create ghastly designs much more quickly.

Grady Booch

(1994)

Was ist von OOAD-Tools zu erwarten? (II)

SOFTWARE RESEARCH LAB

OO Analyse- und Design- (OOAD) Werkzeuge können fogende Aufgaben übernehmen:

- Erstellen und Editieren von Diagrammen basierend auf diversen OO Notationen
- Konsistenz- und Constraint-Checks
  - hat ein Objekt die aufgerufene Methode?
  - werden die Invarianten (nur eine Instanz, etc.) eingehalten?
  - . . .
- Prüfungen auf Vollständigkeit
  - werden alle Methoden/Klassen benutzt?
  - . . .

Konventionelle (SA/SD) versus OO Werkzeuge (I)

Unterschiede ergeben sich in zweierlei Hinsicht:

(1) Software-Architektur

- **Konventionelle Werkzeuge** basieren auf einer **Trennung von Daten (ER-Modell) und Funktionen**

- **OO Werkzeuge** bieten **bessere Ausdrucksmittel**, um Objekte als in sich geschlossene Einheiten aus Daten und Funktionen zu sehen und darzustellen

Konventionelle (SA/SD) versus OO Werkzeuge (II)

SOFTWARE RESEARCH LAB

(2) Semantische Möglichkeiten

Konventionelles ER-Modell:

has_a (1:1)

is_a (1:1)

owns, contains, is_contained_in (1:m)

consists_of (m:m)

Konventionelle (SA/SD) versus OO Werkzeuge (III)

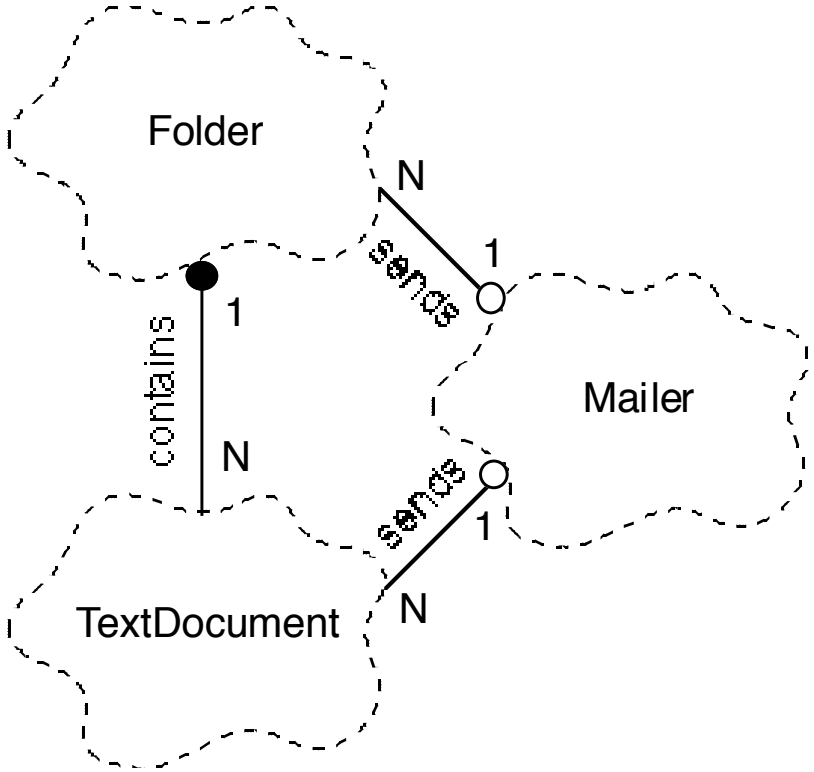
Bei OO Modellierung werden **umfangreichere** und vom ER-Modell abweichende **Ausdrucksmöglichkeiten** geboten:

- Klassen-/Objektbeziehungen:
  - inheritance
  - association (friend, virtual, static)
  - has-a (by value, by reference)
  - uses-a (by value, by reference)
  - is_abstract, is_metaclass, is_parameterized
  - . . .
- Zugriffsrechte

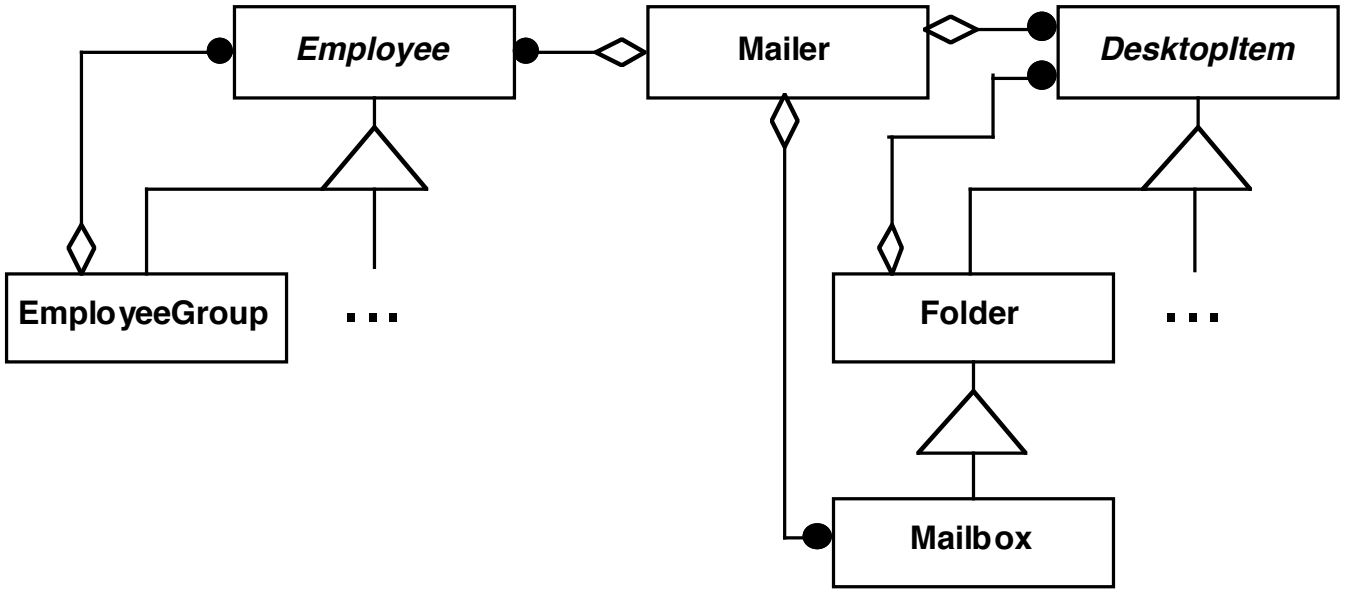
Methodenlandschaft zu Beginn der 90er Jahre

- OOD / Rational Rose
  Grady Booch
- Object Modeling Technique (OMT)
  James Rumbaugh et al.
- OO Software Engineering
  Ivar Jacobson et al.
- OO Analysis (OOA)
  Peter Coad und Ed. Yourdon
- Responsibility-Driven Design (RDD)
  Rebecca Wirfs-Brock et al.
- OO System Analysis (OOSA)
  Sally Shlaer and Steve Mellor
- . . .

Beispiel für Booch-Notation



Beispiel für OMT-Notation



Gemeinsamkeiten von OOAD-Methoden (I)

Sie zielen darauf ab, die **reale Welt ohne künstliche Transformationen auf Software-Systeme** abzubilden:

- Anwendung gleicher Ideen und Konzepte in allen Phasen der Software-Entwicklung
- Grenze zwischen Analyse und Design verschwimmt stärker

Dazu werden sehr vage Richtlinien angegeben.

Gemeinsamkeiten von OOAD-Methoden (II)

OOAD Methoden erlauben die Modellierung folgender Aspekte eines Systems:

- **statische Aspekte**

  im Vordergrund steht das Klassen-/Objektmodell,

  auf höhere Abstraktion werden Subsysteme entworfen

- **dynamische Aspekte**

  Zustandsübergangsdiagramme, Interaktionsdiagramme

Unterschiede zwischen OOAD-Methoden

Die Unterschiede zwischen den Methoden liegen insbesondere in der **Notation**.
Dennoch sind die **Notationen** in den einzelnen OOAD-Methoden **weitgehend sprachunabhängig**.

=> Vereinheitlichung ist naheliegend (→ UML)

All of the OO methodologies have much in common and should be contrasted more with non-OO methodologies than with each other.

James Rumbaugh

(1991)

UML-Einflüße

Die **Unified Modeling Language (UML)** enhält diverse Aspekte und Notationen aus verschiedenen Methoden:

- Booch
  - Harel (State Charts)
  - Fusion (Methodennummerierung)
- Rumbaugh (Notation)
- Jacobson (Use Cases)
- Wirfs-Brock (Responsibilities)
- Shlaer-Mellor (Object Life Cycles)
- Meyer (Pre- und Post-Conditions)

UML-Standard

- Der erste Draft (Version 0.8) wurde im Oktober 1995 publiziert.
- Diverse Anpassungen und die Einbeziehung von Ivar Jacobson führten zu Version 0.9 im Oktober 1996.
- Version 1.0 wurde der Object Management Group (OMG) im Juli 1997 als Grundlage zur Standardisierung vorgelegt.

  Im September 1997 wurden noch diverse Details in Version 1.1 modifiziert.
- 1998 und 1999 ist die Standardisierung wesentlicher Elemente vorangetrieben worden.
- Im April 1999 wurde Version 1.3 veröffentlicht.

The Unified Modeling Language(I)

- Was ist die UML?
  - Sprache
    - Kommunikation
    - Austausch von Ideen
  - Modellierungssprache
    - Vokabeln und Regeln für die Darstellung von Aspekten eines Systems

The Unified Modeling Language(II)

- Was ist die UML nicht?
  - Keine Methode
    - Spezifiziert wie Modelle gemacht werden, aber nicht welche und wann.
    - Aufgabe des Entwicklungsprozesses

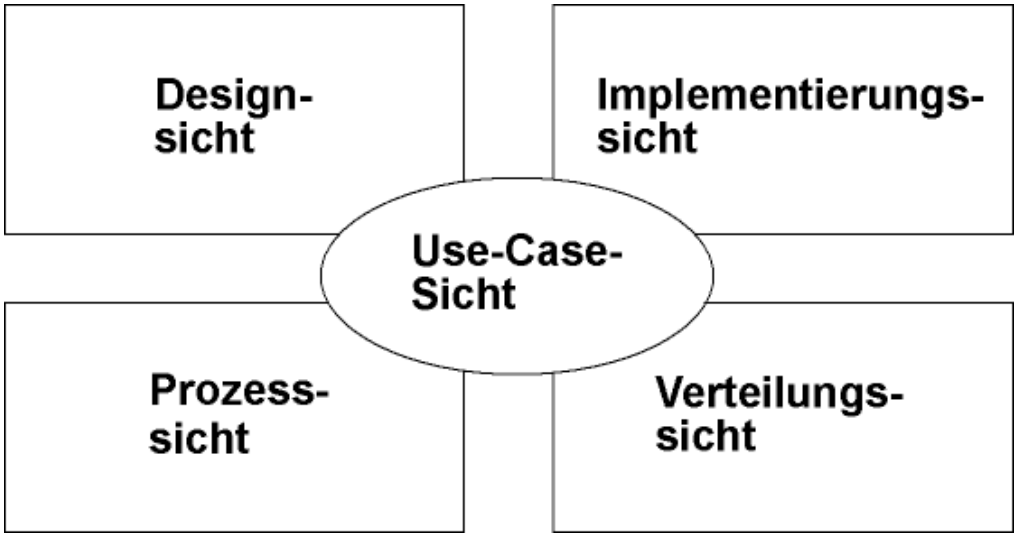
Prozess + Modellierungssprache = Methode

The Unified Modeling Language(III)

- Wofür braucht man die UML?
  - Visualisierung von Modellen
  - Spezifikation von Modellen
  - Konstruktion des Systems
    - Forward and Reverse Engineering
  - Dokumentation des Systems

The Unified Modeling Language(IV)

- Modelle
  - Projektionen des Systems auf eine Sichtweise
  - Für das Verständnis des Systems

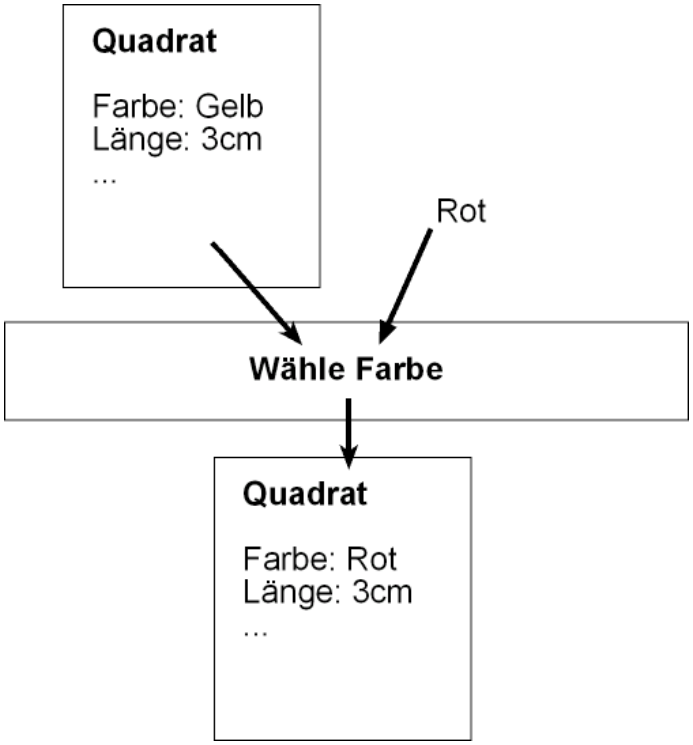


OO Konzepte
Darstellung in UML

- Objekte, Klassen, Nachrichten/Methoden
- Vererbung, Polymorphismus, Dynamische Bindung
- Abstrakte Klassen, abstrakte Kopplung

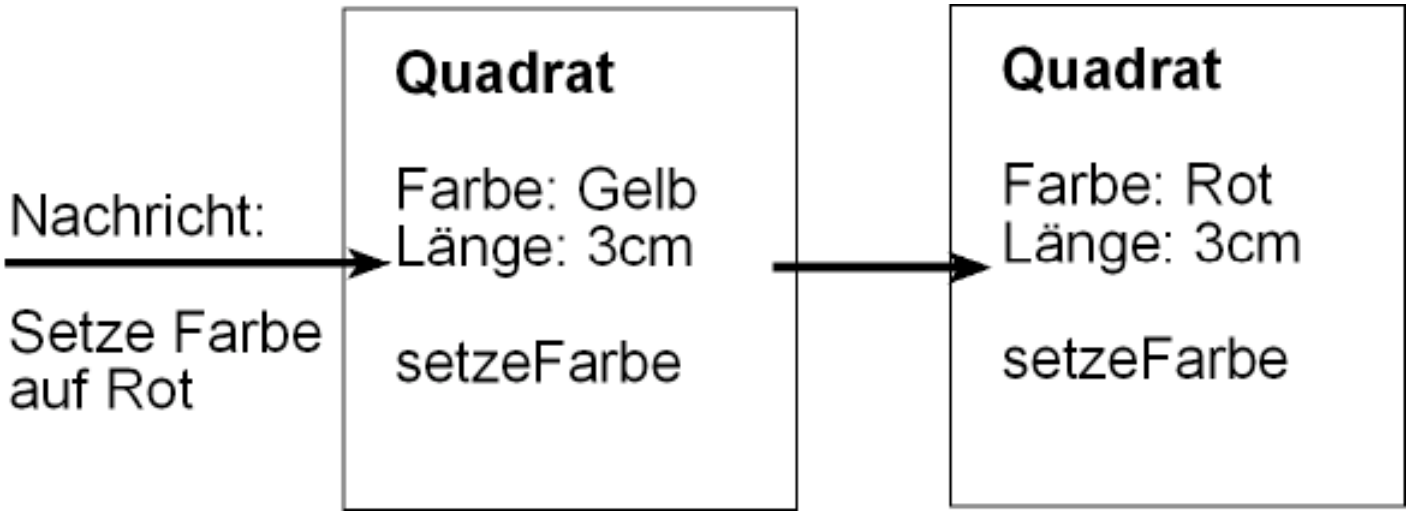
OO vs Prozedural

- Prozedural
  - Trennung von Daten und Prozeduren



OO vs Prozedural

- Objekt-orientiert:
  - Daten und Prozeduren bilden eine logische Einheit → ein Objekt



Objekte(I)

Ein Objekt ist eine Repräsentation einer

konkreten Einheit, wie

einer Person, einem Auto, etc.

oder einer logischen Einheit, wie

einem chemischen Prozess, einer mathematischen Formel, etc.

Objekte(II)

- Die wesentlichen Merkmale eines Objektes sind:

Seine Identität

Sein Zustand

Sein Verhalten

Objekte(III)

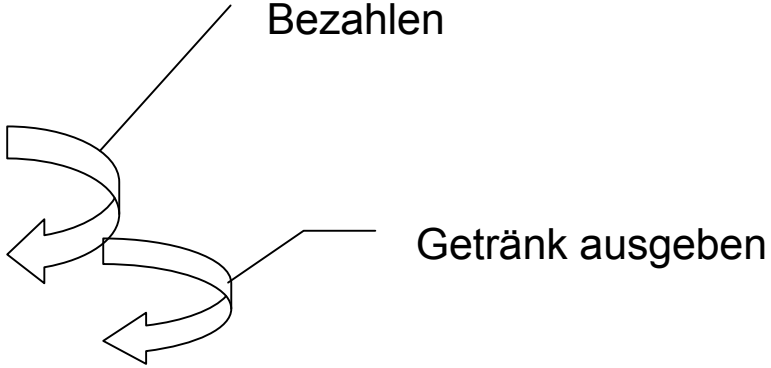
- Zustand

  Der Zustand eines Objektes besteht aus seinen statischen Eigenschaften und deren dynamischen Werten.

  - Werte können primitiv sein: int, double, boolean
  - Werte können Referenzena auf andere Objekte oder selbst andere Objekte sein

Objekte(IV)

- Beispiel
  - Getränkeautomat
    - Zustand: Bereit
    - Zustand: Bezahlt
    - Zustand: Bereit



  - Eigenschaften – Werte
    - Bezahlt:boolean
    - Dosen:Menge von Dosen

Objekte(V)

- Das Verhalten von Objekten wird spezifiziert durch **Methoden (= Operationen).**
- Im Prinzip sind Methoden konzeptionell wie Prozeduren/Funktionen:

  Methode = Name + Parameter + Rückgabeparameter

Objekte(VI)

- Beispiel
  - Quadrat:
    - Name der Operation: setzeFarbe
    - Parameter: Name der Farbe (z.B.: Rot)
    - Rückgabeparameter: keine
- Das Aufrufen einer Operation (z.B.:setzeFarbe) eines Objektes wird als S**chicken einer Nachricht an das Objekt** bezeichnet.

Objekte(VII)

- Identität

  Identität ist die Eigenschaft eines Objektes, die es von allen anderen unterscheidet.

- Zwei Objekte können verschieden sein, auch wenn ihre Eigenschaften, Werte und Operationen übereinstimmen.

Objekt-Orientierung

- Klassifikation
  - Gruppierung von Objekten
- Polymorphismus
  - Statische und dynamische Typen
  - Dynamischen Binden
- Vererbung
  - Hierarchien von Typen

Klassifikation(I)

- Klasse

  Eine Klasse ist eine Menge von Objekten, die gleiche Struktur und gleiches Verhalten haben.

- Klasse

  Eine Klasse ist eine Schablone, aus der Objekte generiert werden können.

Klassifikation(II)

- Klasse: Person
  - Eigenschaften: Name:String, Alter:int....
  - Operationen: schlafe, esse, ...
- Objekt vom Typ Person: Steffen
  - Eigenschaften: Name:Steffen, Alter:24

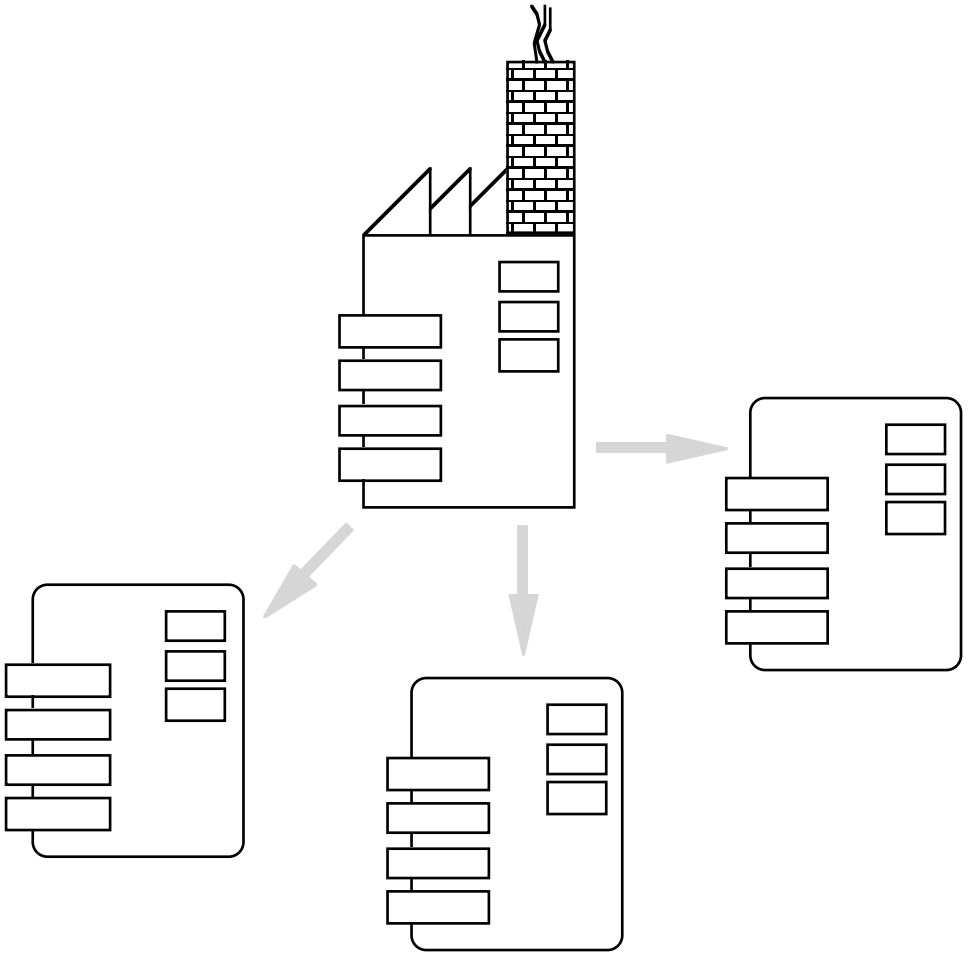
Klasse als Schablone/ Typ (I)

- Vergleich mit der Sprache C:

```
struct{
    int day, month, year;
} date;
date d1, d2;
```

=> alles zugreifbar, keine Methoden

Klasse als Schablone/ Typ (II)



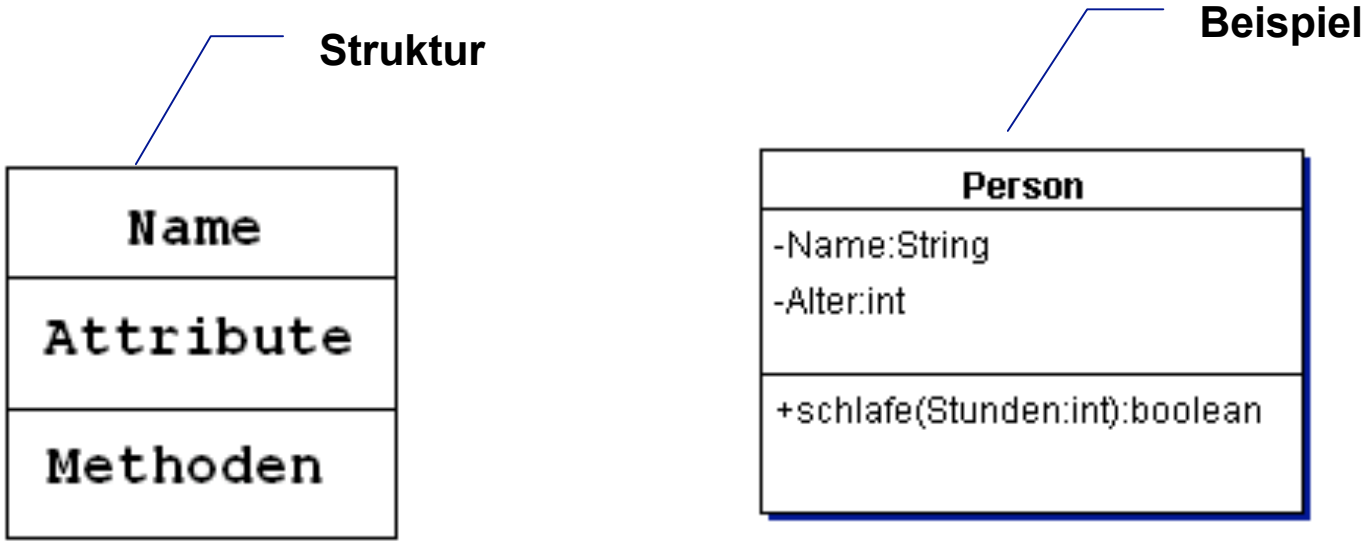
Klasse als Schablone/ Typ (III)

Eine Klasse gibt an, von welchem Typ eine Objekt ist, d.h. welche Nachrichten es versteht und welche Eigenschaften es hat.

- Eine Klasse besteht aus:
  - einem eindeutigen Name
  - Attributen(=Eigenschaften) und deren Typen
  - Methoden/Operationen

Klassen in UML(I)

- UML Notation einer Klasse:



Klassen in UML (II)

Notation für Attribute:

A	nur der Attributname
: C	nur der Name der Attributklasse
A : C	Attributname und Klasse
A : C = E	w. o.; zusätzliche Definition eines Attributwertes

timeWhenStarted	→ A
: Date	→ : C
timeWhenStarted : Date	→ A : C
timeWhenStarted : Date = 1.1.1999	→ A : C = E
timeWhenStarted = 1.1.1999	→ A = E

Klassen in UML (III)

Notation für Methoden/Operationen:

m()	nur der Methodenname
m(arguments): R	Methodenname, Argumente, Rückgabeparametertyp

Beispiele:

printInvoice()	→ m()
printInvoice(itemNo: int): bool	→m(arguments): R

Klassen in UML(IV)

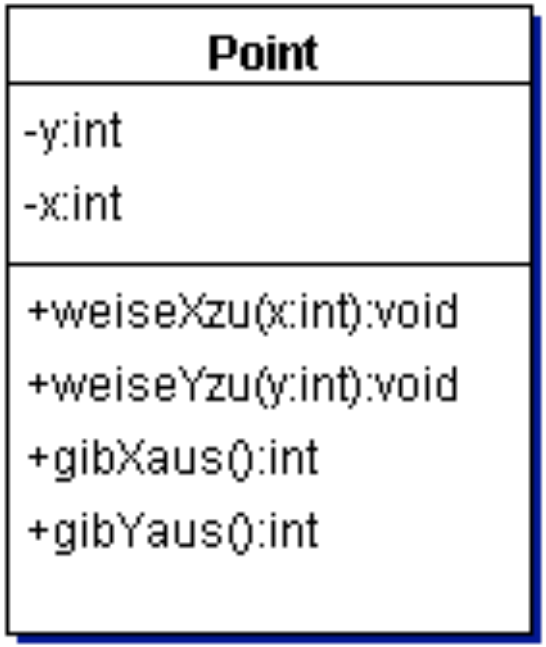
Sogenannte ***Adornments*** (dt.: Schmuck, Zierde) ergänzten in der Booch-Methode Notationselemente und wurden durch ein Dreieckssymbol dargestellt.

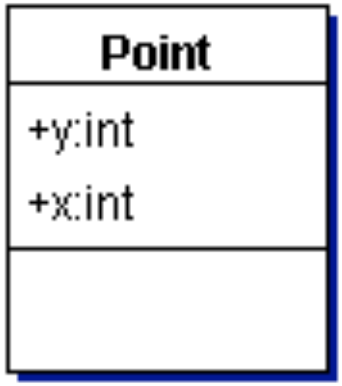
Methoden und Attribute haben in UML **grafische Symbole** beigefügt, um die **Zugriffsrechte** (public, protected, private) auszudrücken.

Beispiel:

+schlafe(Stunden:int)

Beispiel: Zugriffsrechte

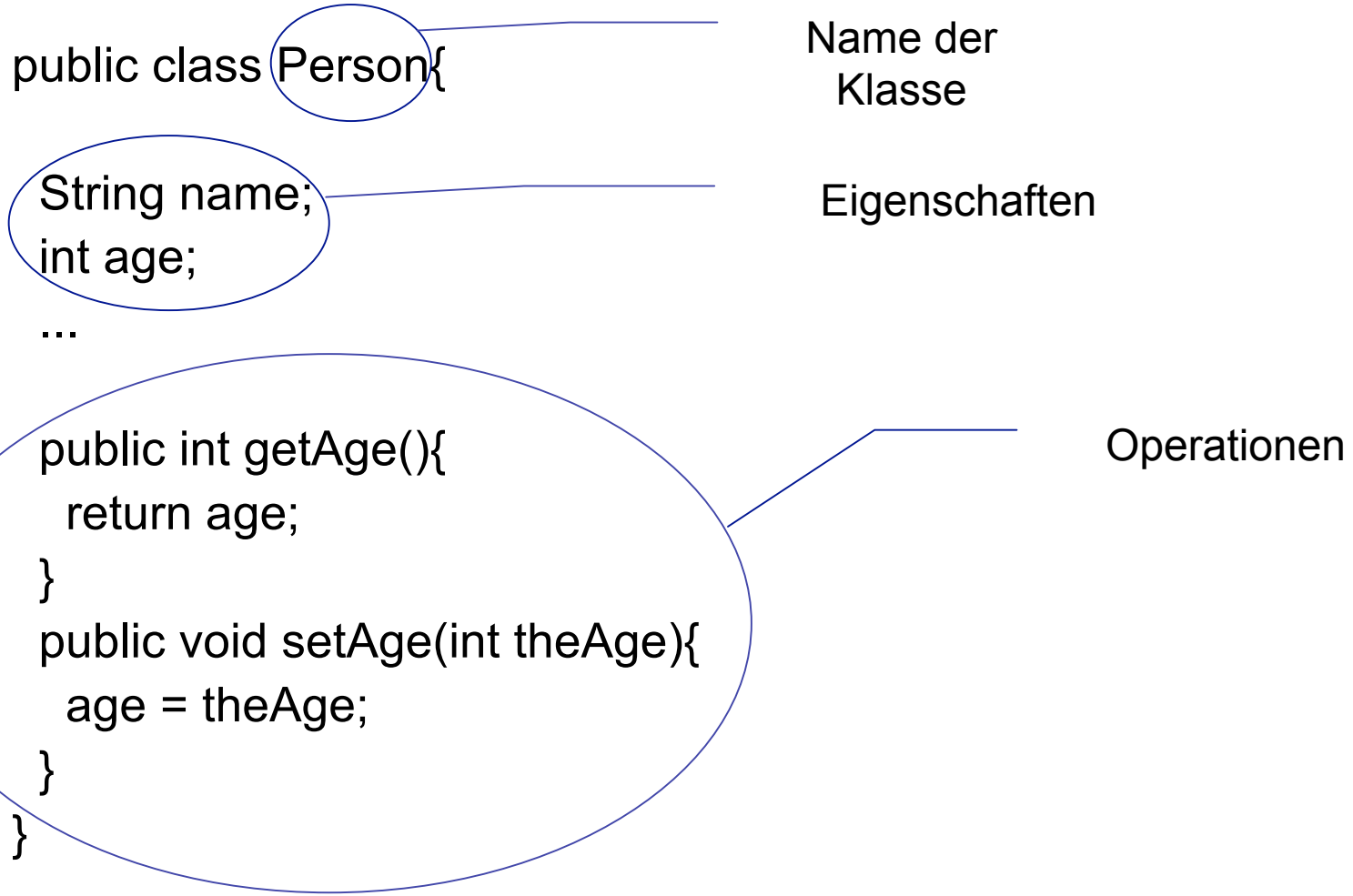




Unnötige Komplexität, da keine Abhängigkeit zw. x und y besteht

besser

Klassen in Java



Verwendung von Klassen in Java

- Klassen werden in Java dazu verwendet, den Typ von Variablen festzulegen und Objekte zu instanziieren
- Schlüsselwort: **new**
- Beispiel

```
Person manager = new Person(„Steffen“);
```

Deklaration der Variable „manager“

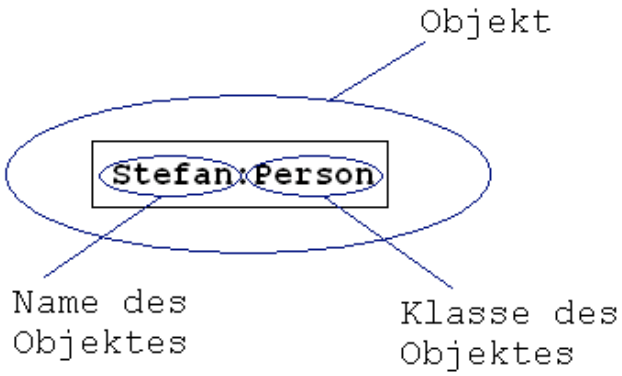
Instanziierung eines Objektes der Klasse Person mit dem Namen Steffen

Beispiel: Hotelreservierung

- Was könnte man bei einerm Hotelreservierungsystem als Klassen modellieren?
- Welche Eigenschaften haben diese Klassen?
- Welche Operationen?
- Welche Instanzen(Objekte) dieser Klassen könnte es geben?

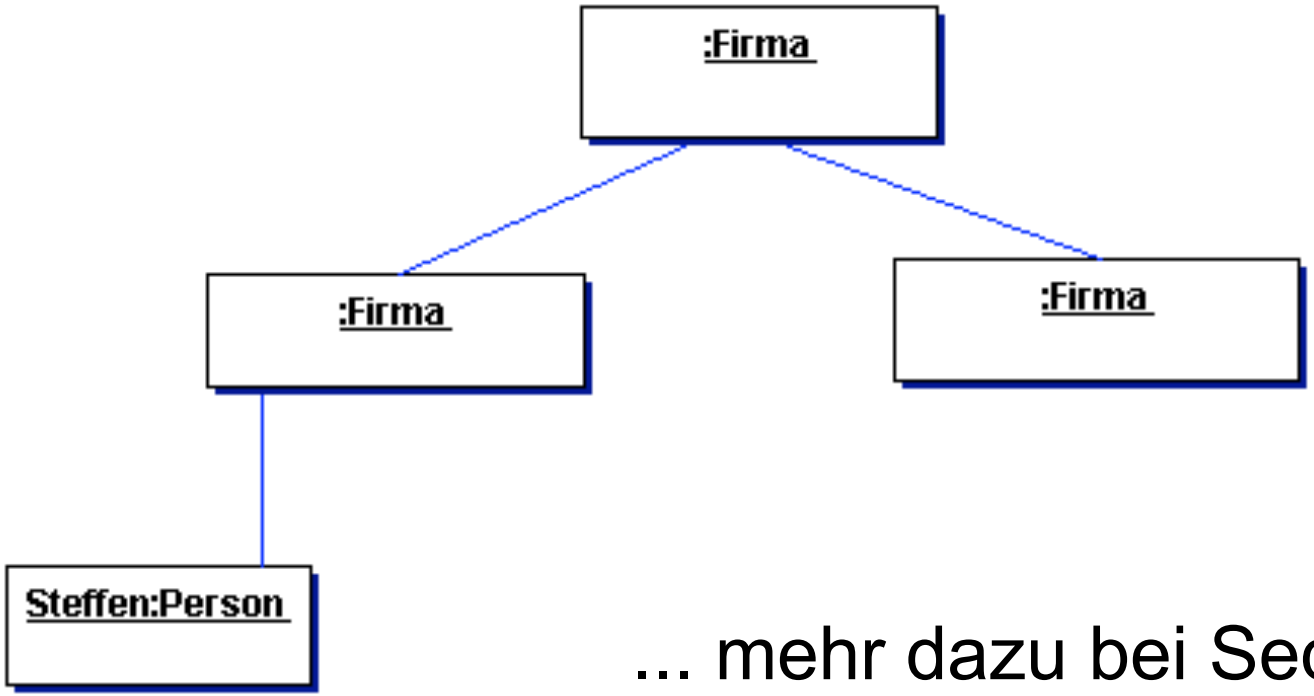
Objekte in UML

- Notation in UML



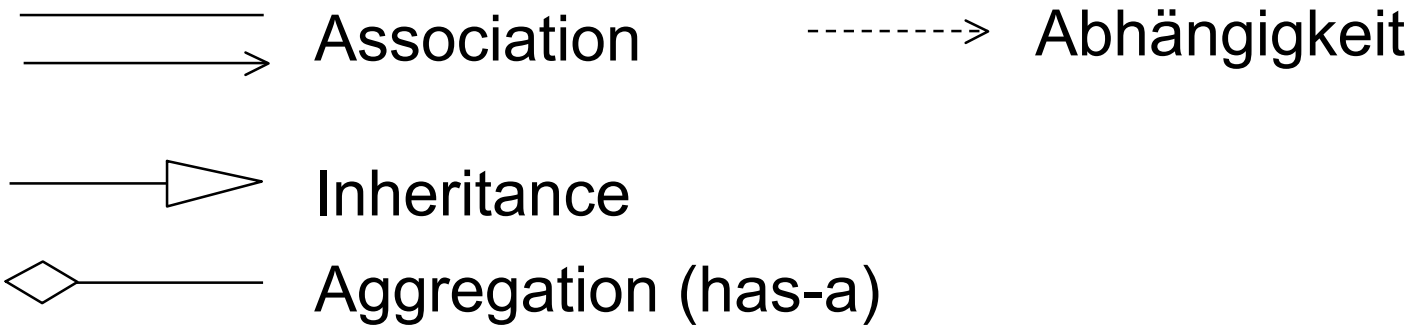
Objektdiagramme stellen einen Laufzeit-Schnappschuss des Systems dar. Dabei werden Objekte und Verbindungen zwischen den Objekten dargestellt, die ausdrücken, daß zwischen den Objekten navigiert werden kann

Objektdiagramme



... mehr dazu bei Sequenz-diagrammen

Klassenbeziehungen (I)



Eine Association zwischen zwei Klassen kann durch die anderen Beziehungen verfeinert werden.

Oft modelliert man zunächst nur die Tatsache, daß zwei Klassen untereinander in Beziehung stehen und verfeinert später dieses allgemeine Notationselement.

Klassenbeziehungen (II)

Jede der Beziehungen kann durch einen **Text-Label** genauer gekennzeichnet werden (→ ER-Modell).

Zusätzlich können an den Enden einer Association **Rollen-Namen** spezifiziert werden.

Eine Klasse kann **auch eine Association auf sich selbst** haben. Damit wird ausgedrückt, daß Objekte der gleichen Klasse in Beziehung stehen.

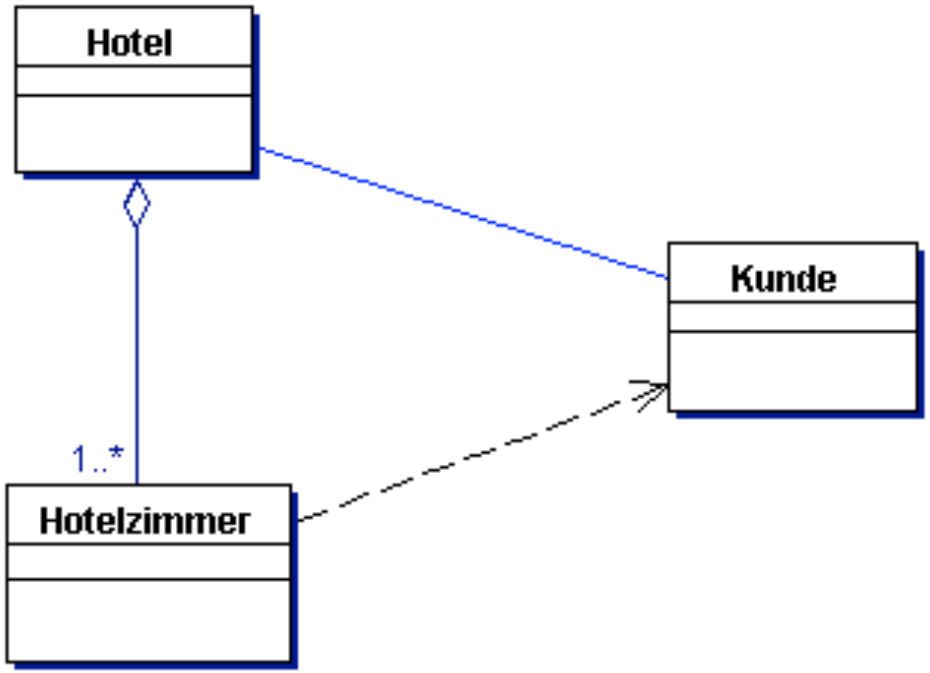
Klassenbeziehungen (III)

Die Kardinalität von Beziehungen kann wie in der ER-Notation jeweils am Ende einer Beziehung angegeben werden:

1	genau eins
*	beliebig (0 oder mehr)
0..*	beliebig (0 oder mehr)
1..*	1 oder mehr
0..1	0 oder eins
2..5	Wertebereich
1..5, 9	Wertebereich oder genaue Zahl

Klassenbeziehungen (IV)

Beispiel:



Vererbung
Polymorphismus
Dynamische Bindung

Vererbung(I)

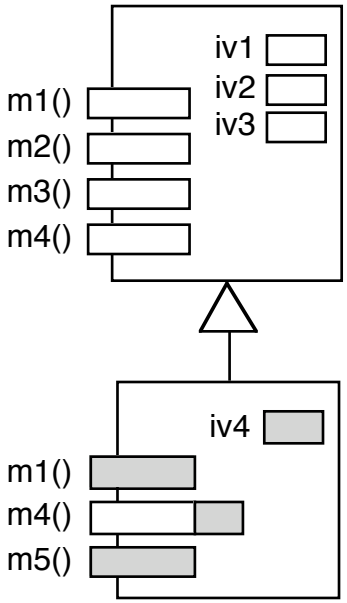
- Klassen definiren den Typ eines Objektes
- Modelliert man beispielsweise eine Klasse **Kunde** und eine Klasse **Firmenkunde**, so sollte jedes Objekt vom Typ **Firmenkunde** auch vom Typ **Kunde** sein. Der Typ **Firmenkunde** ist ein Untertyp vom Typ **Kunde**.

Vererbung(II)

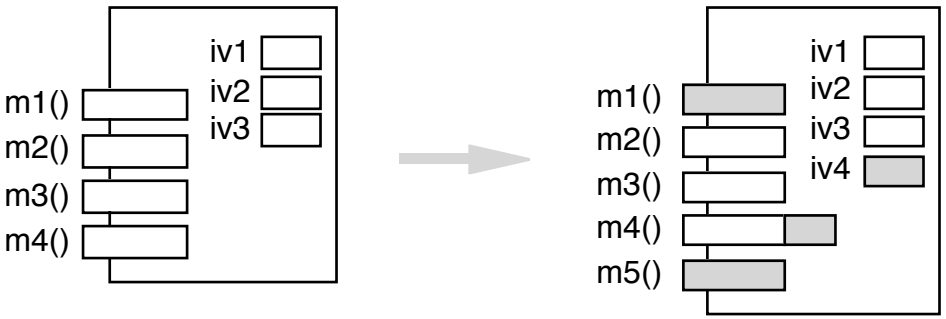
- Eine Oberklasse verallgemeinert eine Unterklasse
- Eine Unterklasse spezialisiert eine Oberklasse
- Eine Unterklasse **erbt** alle Methoden und Eigenschaften einer Oberklasse.

Vererbung(III)

- Eine Unterklasse hat folgende Möglichkeiten, ihr Verhalten zu spezialisieren:
  - Neue Operationen und Eigenschaften definieren
  - Die Implementation bestehender Operationen modifizieren (eine Methode „überschreiben“).

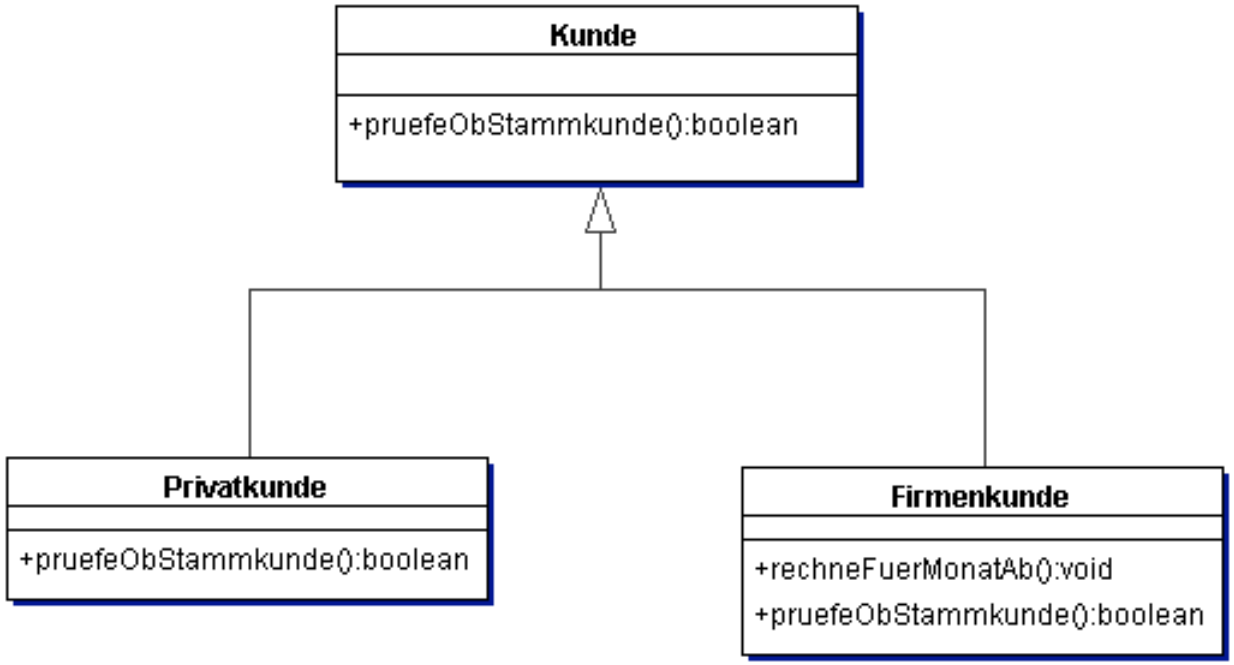


Flache Sicht:



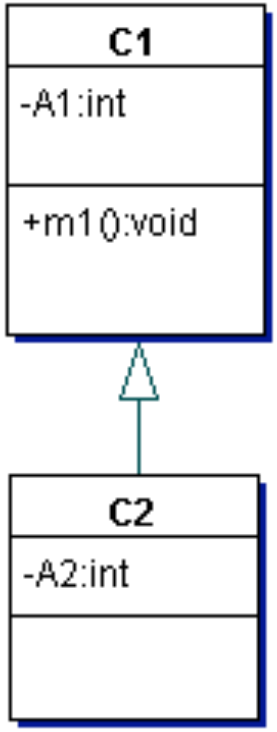
Vererbung(IV)

- UML Notation:

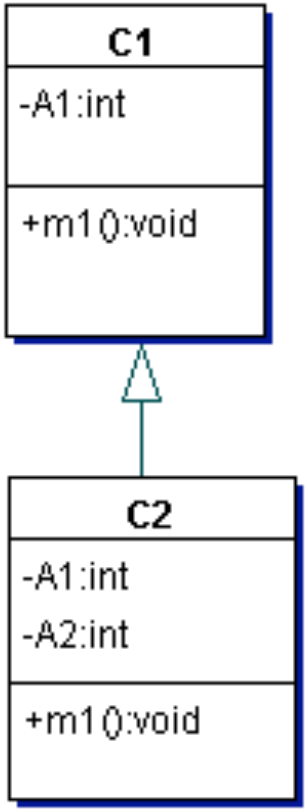


Vererbung (V)

"Delta"-Sicht



„Flache" Sicht
(nicht in Standard UML!)



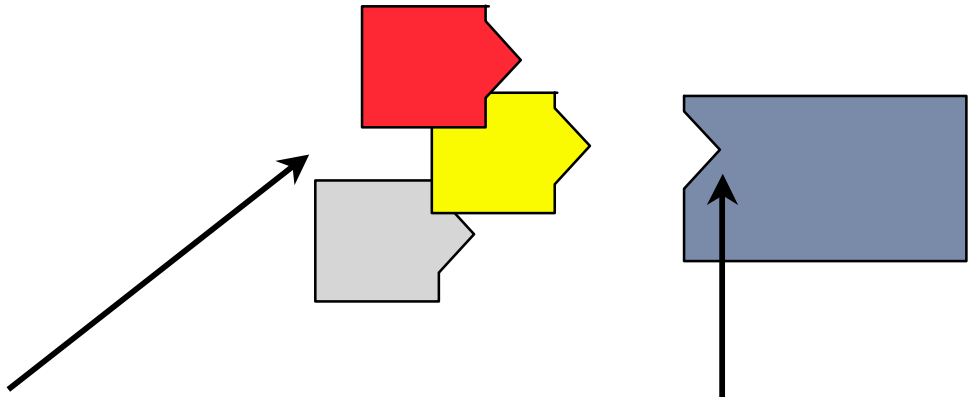
Vererbung (VI)—in Java

- Java unterstützt einfache Vererbung, d.h. Jede Klasse hat höchstens eine Oberklasse.
- Das Schlüsselwort ist **extends.**

```
public class Firmenkunde extends Kunde{
    ...
}
```

Polymorphismus (I)

Sogenannte Objekttypen sind **poly** (= viel) **morph** (= Gestalt). Anschaulich ist das mit **„Steckerkompatibilität“** vergleichbar:



Objekte, die zu diesem Stecker kompatibel sind

„Stecker“-Standard

Polymorphismus (II)

Objekte vom Typ Firmenkunde(Unterklasse) halten mindestens den selben Vertrag ein **wie Objekte vom Typ Kunde(Oberklasse)**.

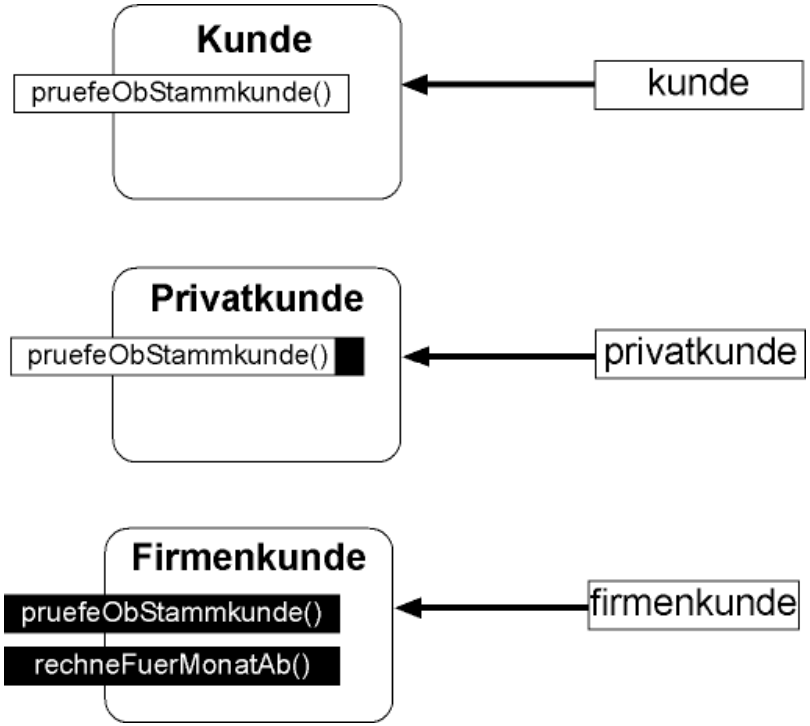
Es ist daher sinnvoll, zu definieren, daß ein Objekt einer Klasse $A_{i,}$ die eine Unterklasse von A ist, nicht nur den Typ A_i sondern auch den Typ aller Oberklassen von A_i hat, insbesondere natürlich den Typ A.

Das heißt, daß **ein Objekt nicht nur einen Typ sondern beliebig viele Typen** hat. Die Anzahl ist abhängig von der Position jener Klasse in der Klassenhierarchie, aus der das jeweilige Objekt generiert wurde.

Der Objekttyp ist also **poly** (= viel) **morph** (= Gestalt).

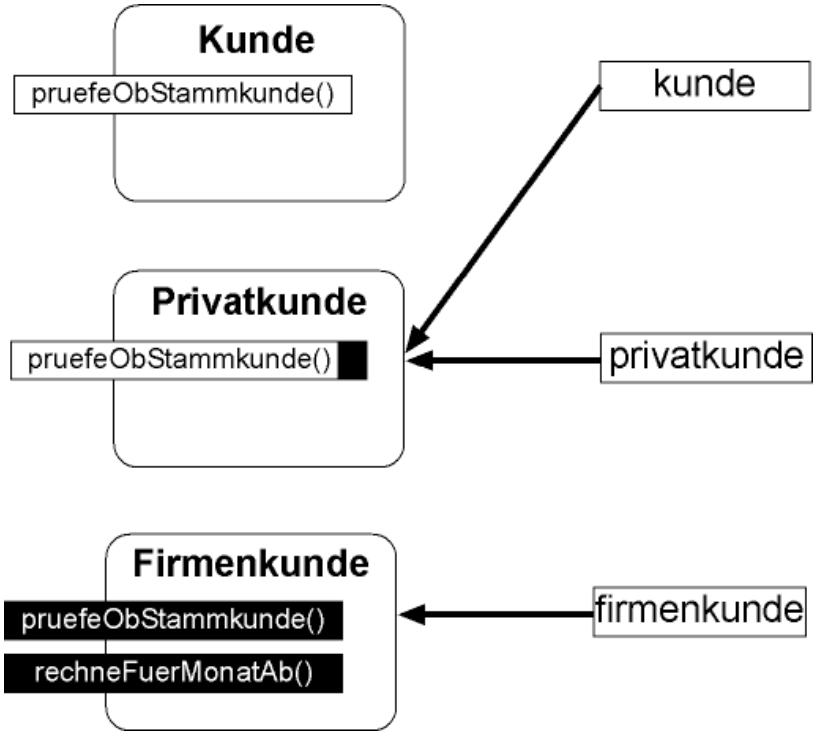
Polymorphismus-Beispiel (I)

```
Kunde kunde= new Kunde();
Privatkunde privatkunde = new Privatkunde();
Firmenkunde firmenkunde = new Firmenkunde();
```

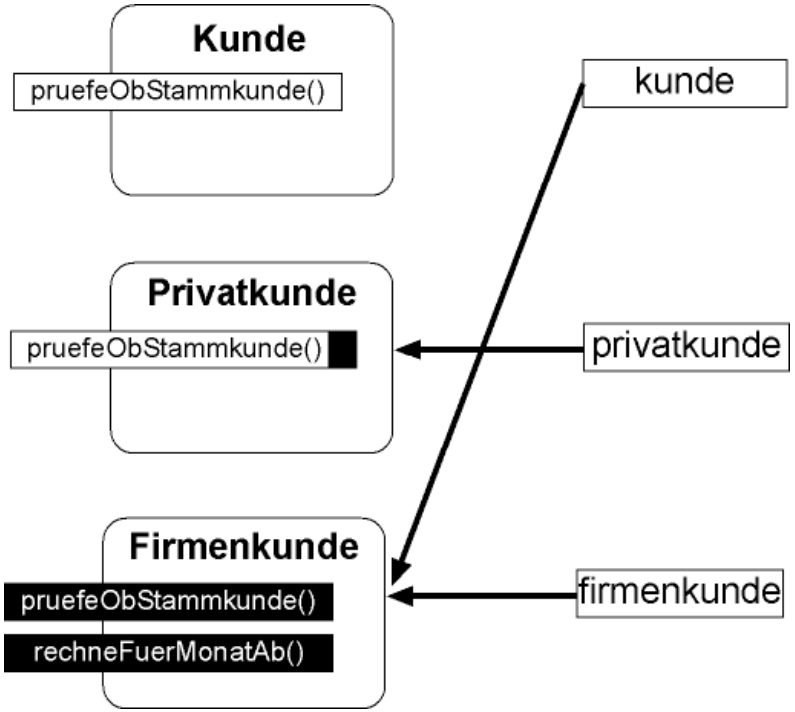


Polymorphismus-Beispiel (II)

kunde=privatkunde; //OK

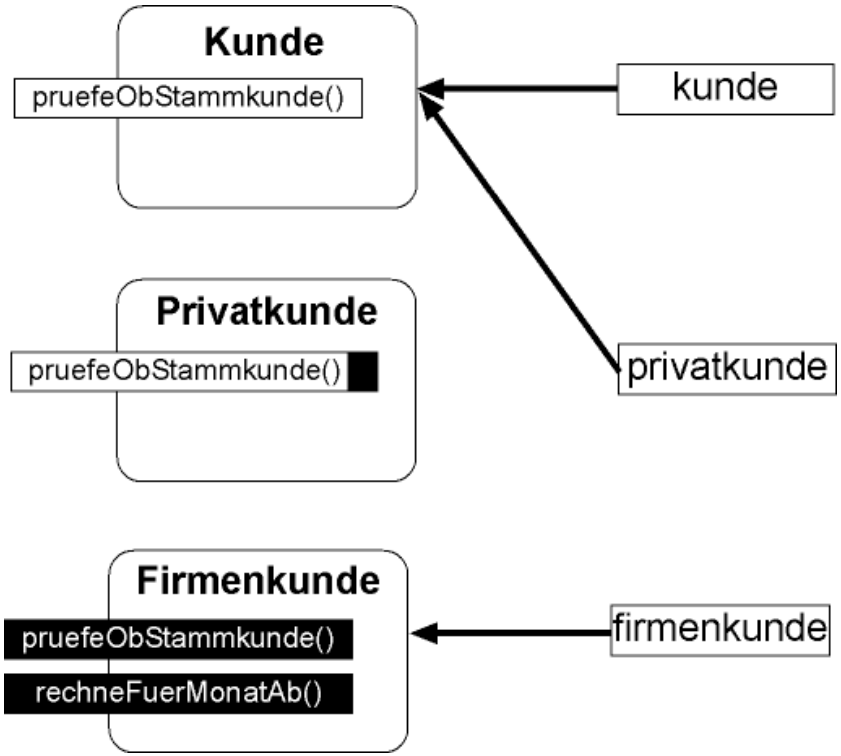


kunde=firmenkunde;//OK

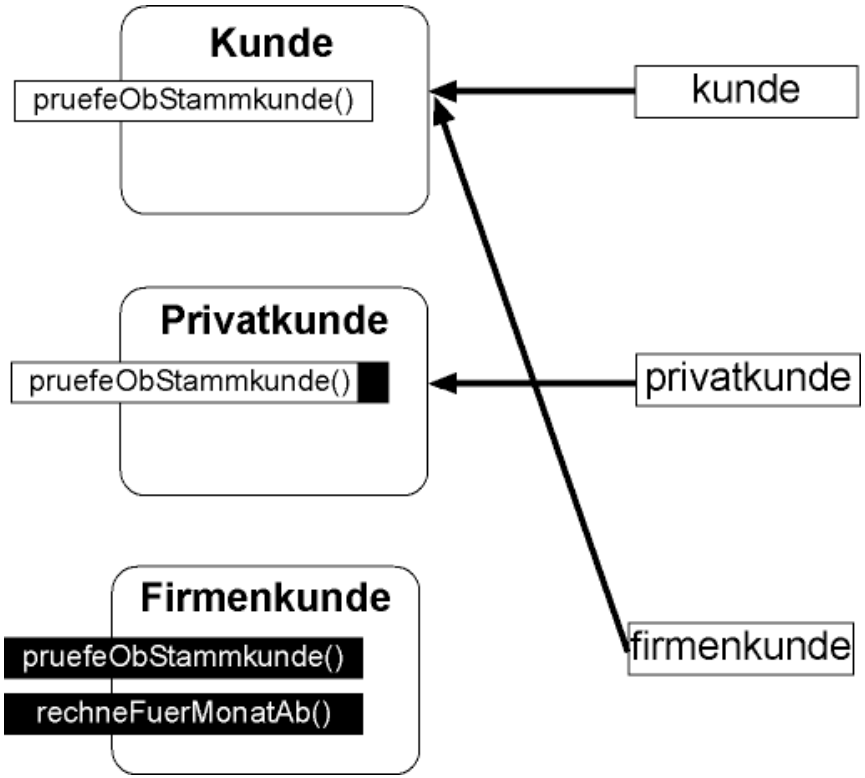


Polymorphismus-Beispiel (III)

privatkunde = kunde; //Fehler



firmenkunde = kunde ; //analog



Polymorphismus-Beispiel (IV)

Der Grund liegt darin, daß ein von „kunde“ referenziertes Objekt (d.h.:, wenn es eine Instanz von Kunde ist) nicht unbedingt Methodenaufrufe verstehen muß, die Instanzen von Unterklassen von Kunde verstehen würden:

```
firmenkunde=kunde;
firmenkunde.rechneFuerMonatAb();
```

würde in einem Laufzeitfehler resultieren (**"keine Methode zum Methodenaufruf gefunden"** = "Methodenaufruf nicht verstanden"), wenn kunde (und nach der Zuweisung auch firmenkunde) ein Objekt referenziert, das z.B. aus Kunde generiert wurde.

Statischer und dynamischer Typ von Variablen

In objektorientierten Sprachen mit strenger Typenprüfung (wie in Java, Eiffel, Oberon, C++) muß man zwischen einem statischen und einem dynamischen Datentyp unterscheiden:

- der **statische Typ** einer Variable ist der Typ aufgrund der Variablendeklaration im (statischen) Programmtext
- der **dynamische Typ** einer Variable ist der Typ des referenzierten Objektes zur Laufzeit

Eine Variable hat somit **exakt einen statischen Typ**. Zur Laufzeit kann sie **mehrere dynamische Typen** annehmen (abhängig von der Breite und Tiefe der Klassenhierarchie).

Im zuvor präsentierten Beispiel hat die Variable kunde den statischen Typ Kunde. Nach der Zuweisung **kunde = firmenkunde** hat sie nach wie vor den statischen Typ Kunde, jedoch den dynamischen Typ Firmenkunde, da sie nun eine Instanz der Klasse Firmenkunde referenziert.

Dynamische Bindung (I)

Dynamische Bindung heißt, daß der **Compiler nicht festlegt, welche Methode zur Laufzeit aufgerufen wird.** Welche Methode tatsächlich zur Laufzeit aufgerufen wird, hängt

- vom Methodennamen
- vom dynamischen Typ der Variable

ab.

Beispiel (basierend auf zuvor präsentierter Klassenhierarchie):

```
Kunde kunde= new Firmenkunde;  // durch Polymorphismus möglich
kunde.pruefeObStammkunde();
```

Dynamische Bindung (II)

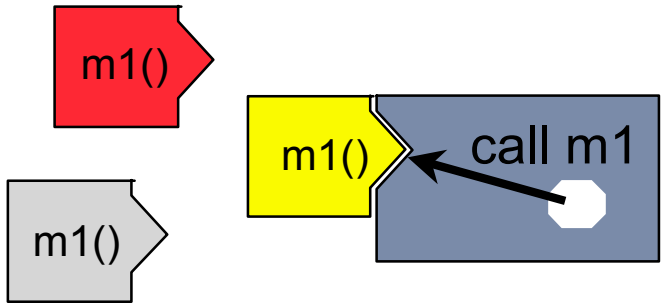
Die Variable kunde referenziert ein Objekt, das aus der Klasse Firmenkunde generiert wurde (kunde hat also den dynamischen Typ Firmenkunde). Es wird daher die Methode **pruefeObStammkunde(), wie** sie **in Firmenkunde implementiert** ist, ausgeführt.

In Java sind alle Methoden dynamisch gebunden, außer man markiert sie explizit, indem man das Schlüsselwort static vor die Methodendefinition stellt.

(In C++ müssen hingegen Methoden explizit als "dynamisch gebunden" markiert werden. Dazu wird das Schlüsselwort virtual vor die Methodendeklaration gestellt.)

Dynamische Bindung (III)

Dynamische Bindung heißt, daß es **vom eingesteckten Objekt abhängt, welche Methode tatsächlich ausgeführt wird.** Das gelbe Objekt implementiert m1() zB anders als das rote Objekt:



Type-Test und Type-Guard in Java

Type-Test: Abfrage des dynamischen Typs

Type-Guard: Laufzeit-geprüfte Typumwandlung (::Type-Cast)

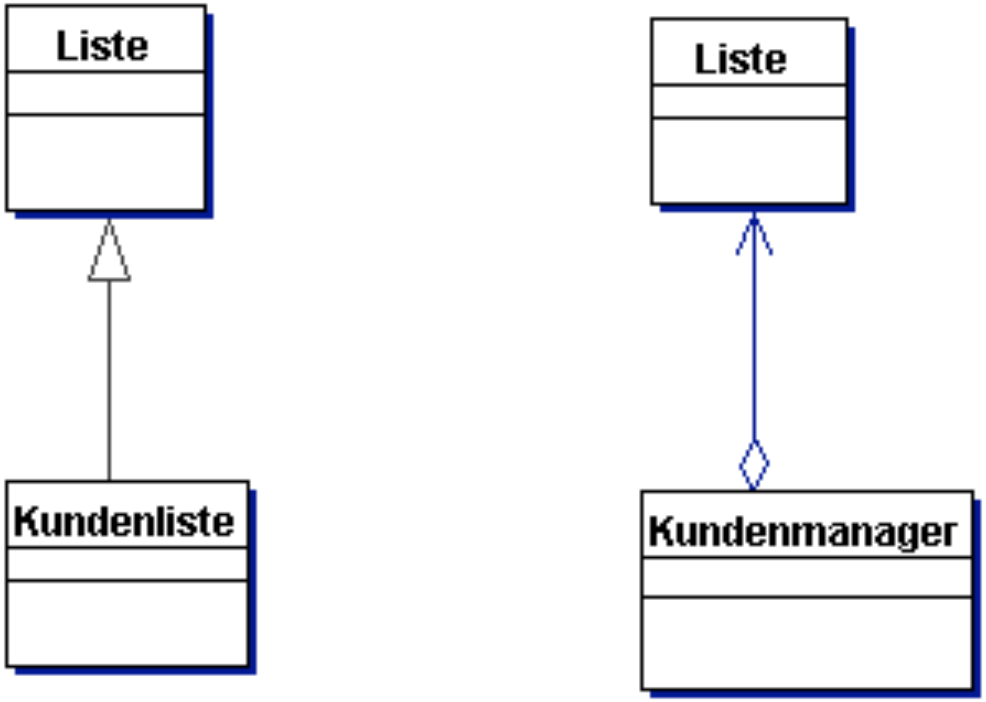
Beispiele:

```
if (kunde instanceof Firmenkunde) {          // type test
  Firmenkunde fKunde= (Firmenkunde )kunde;          // type guard
            ...
}

if (kunde instanceof Firmenkunde)
  ((Firmenkunde)kunde).rechneFuerMonatAb();
```

Achtung: Is-A versus Has-A

Typischer Fehler: Is-A statt Has-A



Verstehen der Interaktionen zwischen Objekten

Object Game

„Durchspielen“ einer Hotelzimmerreservierung

Abstrakte Klassen und abstrakte Kopplung

Warum abstrakte Klassen?

Objektorientierte Sprachen werden in vielen Software-Projekten ähnlich wie modulorientierte Sprachen (Modula-2, Ada) **zur Erstellung von Legobausteinsammlungen** verwendet:

- Klassen sind ein Sprachkonstrukt zur Implementierung von Modulen / abstrakten Datentypen.
- durch Unterklassenbildung können derartige Software-Bausteine an neue Projekte angepaßt werden.

Um jedoch das volle Potential objektorientierter Sprachen auszuschöpfen, also **die Wiederverwendbarkeit von Softwarearchitekturen** zu ermöglichen, ist eine geschickte Kombination von Unterklassenbildung (und somit Polymorphismus) sowie dynamischer Bindung in Form von abstrakten Klassen essentiell.

Abstrakte Klassen (I)

Eigenschaften ähnlicher Objekte **werden in eine gemeinsame Klasse "herausgefiltert"**. Dieser Vorgang kann mit dem "Heraus-faktorisieren" von Termen in mathematischen Ausdrücken verglichen werden.

- Die entstandene Klasse wird **nur wenige Methoden konkret implementieren** können. Wenn auch einige Methoden nicht konkret implementiert werden können, kann man bei **diesen Methoden zumindest Namen und Parameter festlegen und beschreiben, was die Methode "im Prinzip" tun soll**.
- es wird eine **Standardisierung der Klassenschnittstelle** für alle Unterklassen vorgenommen

Abstrakte Klassen (II)

Die durch Herausfaktorisieren von gemeinsamen Eigenschaften entstandenen Klassen **spiegeln meist nicht Objekte der realen Welt wider**. Es handelt sich vielmehr um Abstraktionen davon. Deshalb nennt man diese Klassen abstrakte Klassen.

Ein weiterer Grund für diese Namensgebung ist, daß es **keinen Sinn** ergibt, **Instanzen aus solchen Klassen zu generieren**. Abstrakte Klassen enthalten ja "dummy"-Implementierungen oder keine Implementierungen (→ **abstrakte Methoden**) für einige Methoden.

Abstrakte Kopplung (I)

Andere Klassen können basierend auf abstrakten Klassen implementiert werden. Die **Kopplung** zwischen einer Klasse (zB Klasse B) und einer abstrakten Klasse (zB Klasse A) **kann auf mehrere Arten erfolgen:**

- B hat eine Instanzvariable vom statischen Typ A
- eine oder mehrere Methoden von B haben einen Parameter vom statischen Typ A
- B greift auf eine globale Variable vom statischen Typ A zu

Abstrakte Kopplung (II)

Diese mit einer abstrakten Klasse **gekoppelten Klassen können** dank Polymorphismus und dynamischer Bindung **ohne Änderung mit Objekten beliebiger Unterklassen der abstrakten Klassen**, auf denen sie basieren, **arbeiten**.

Das Verhalten dieser Komponenten wird also nicht durch direkten Eingriff verändert, sondern durch softwaretechnisch saubere Modifikation der abstrakten Klassen in Unterklassen.

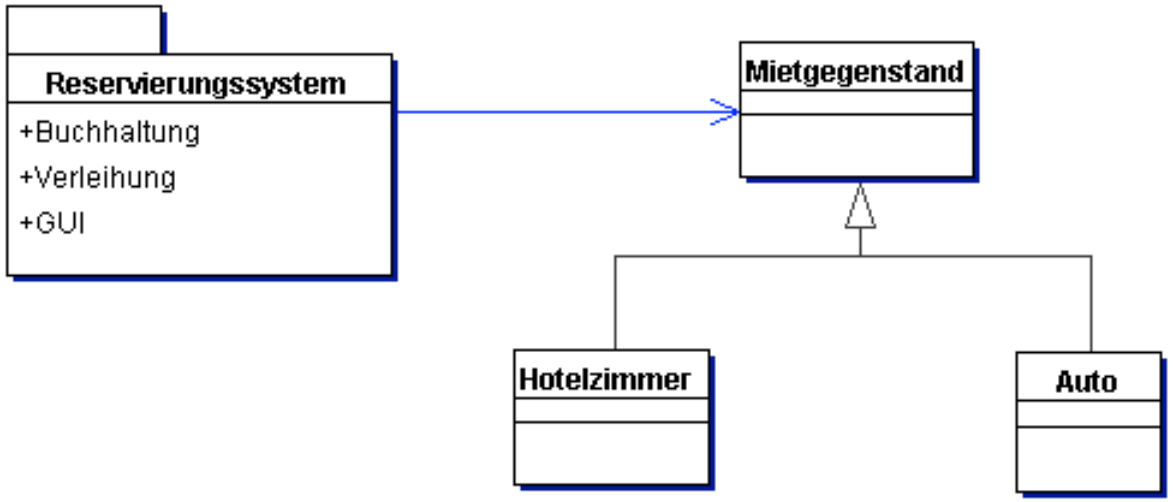
Abstrakte Klassen + abstrakte Kopplung bilden somit die Basis für OO Frameworks (= Halbfertigfabrikate).

Abstrakte Kopplung (III)

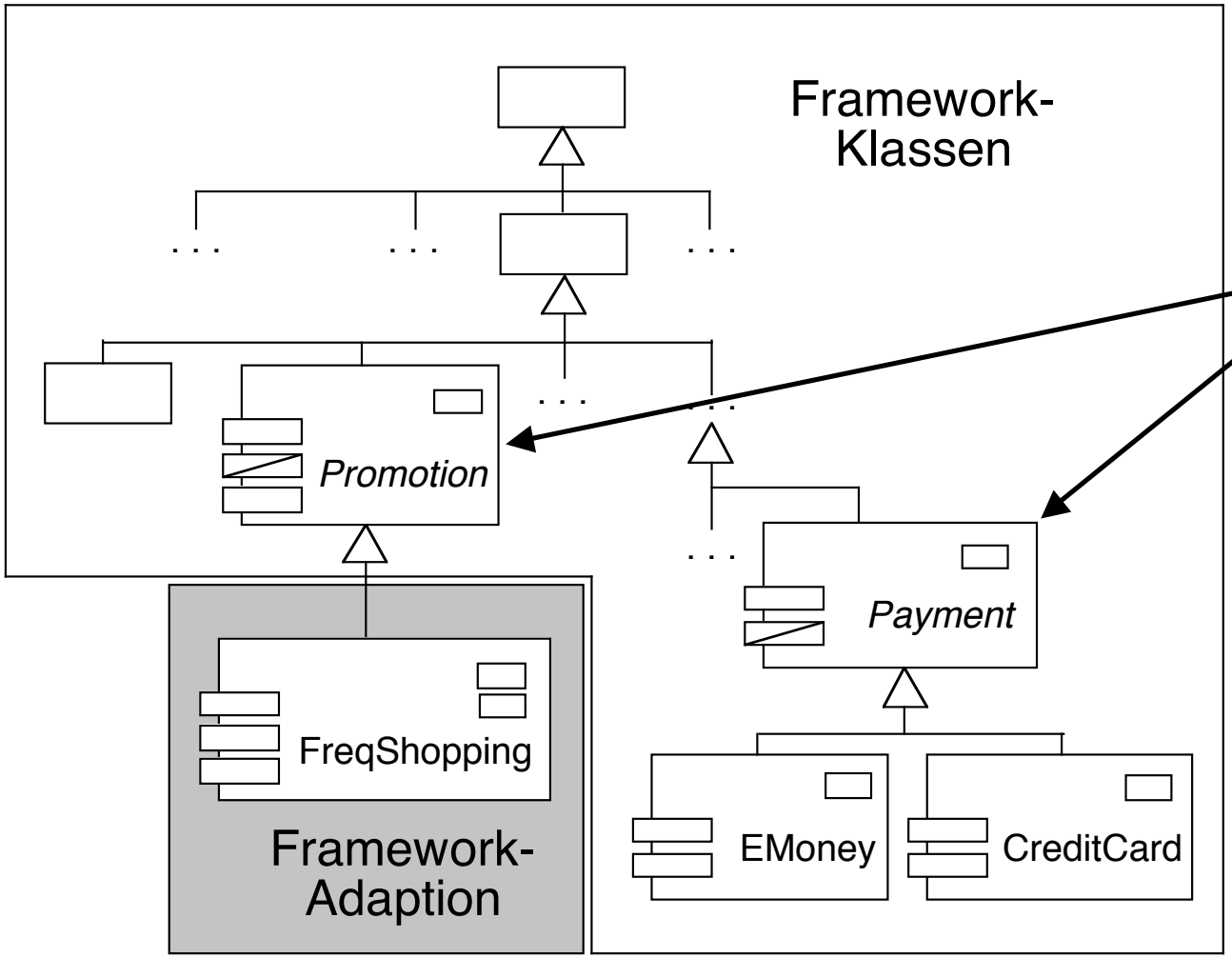
Das **Hauptproblem** ist, **gute Abstraktionen** zu finden, sodaß andere Software-Komponenten darauf aufbauend realisiert werden können.

Abstrakte Klassen entwickeln sich typischerweise erst im Zusammenspiel mit den mit ihnen gekoppelten Klassen evolutionär weiter.

Beispiel Hotelreservierung

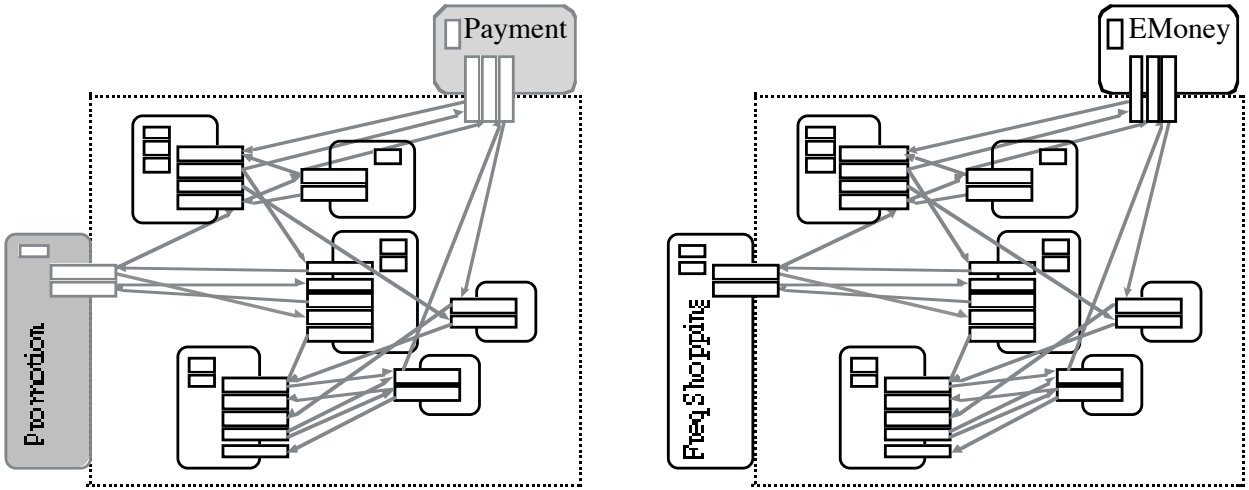


Frameworks— Statische Sicht



abstrakte Klassen
(hier mit jeweils einer abstrakten Methode)

Black-Box versus White-Box Framework-Teile



vor der Adaptierung

nach der Adaptierung

Hands-On Übung

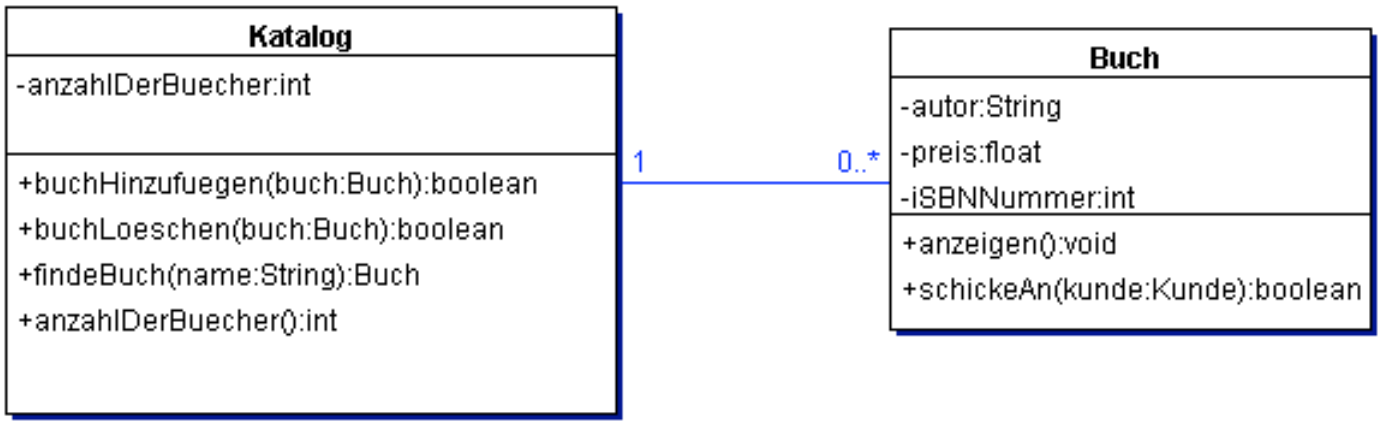
Webshop

Case Study Webshop (I)

- Es soll ein Webshop konstruiert werden, mit Hilfe dessen man Bücher über das Internet kaufen kann.
- Ein Bestandteil soll der „Katalog“ sein, der die Bücher verwaltet.

Case Study Webshop (II)

- Erster Entwurf:

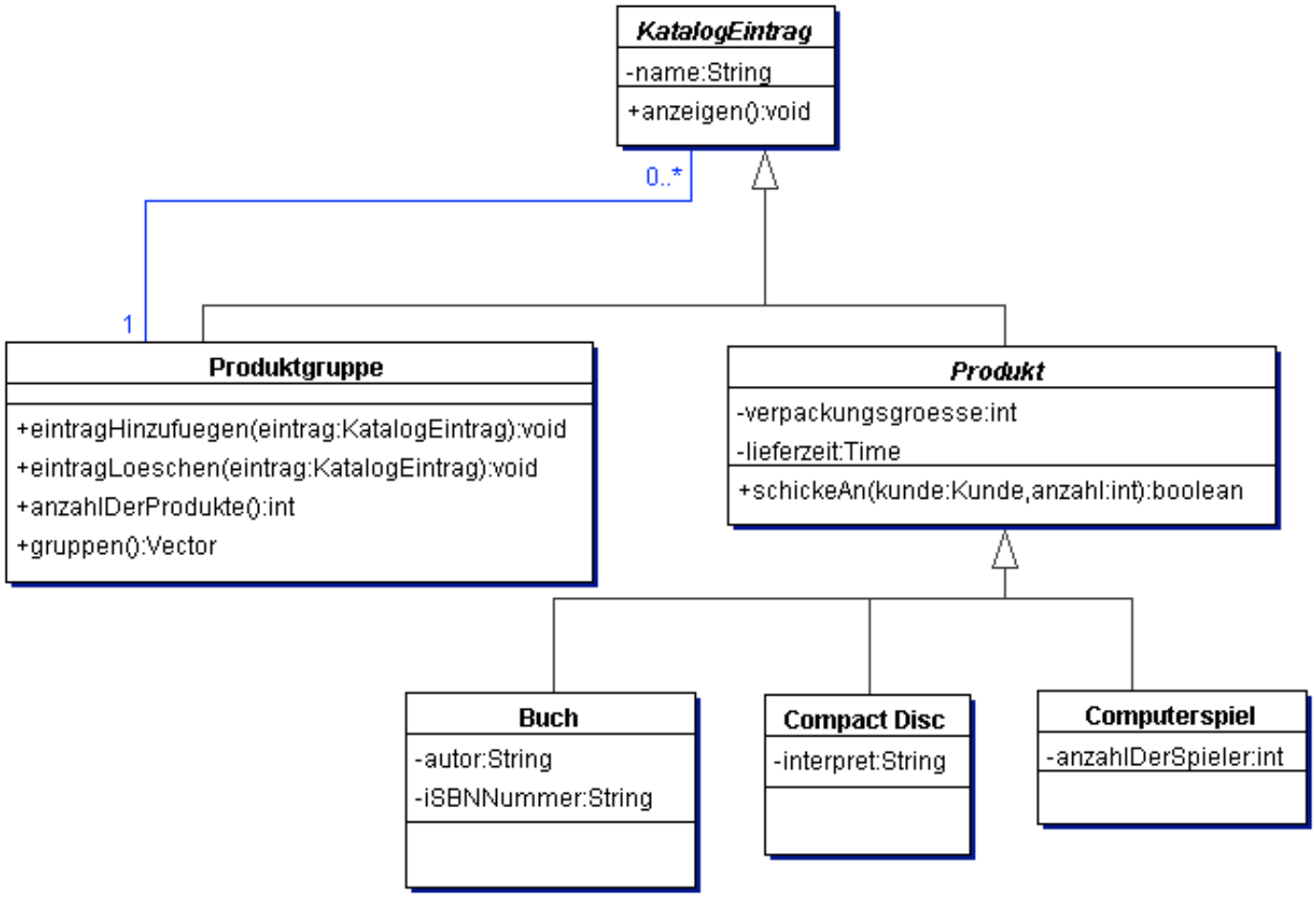


Case Study Webshop (III)

- Problem: Es sollen nun auch CDs und Computerspiele mitangeboten werden; Der Katalog muß also erweitert werden.

Case Study Webshop (IV)

- Neuer Entwurf:



Collaboration & Sequence Diagramme

Collaboration-Diagramm (I)

In einem *Collaboration*-Diagramm gibt es nur einfache Beziehungen zwischen Objekten
(———).

Optional kann auch der Message-Fluß zwischen Objekten dargestellt werden. (Dazu sind aber meist *Sequence*-Diagramme besser geeignet.)

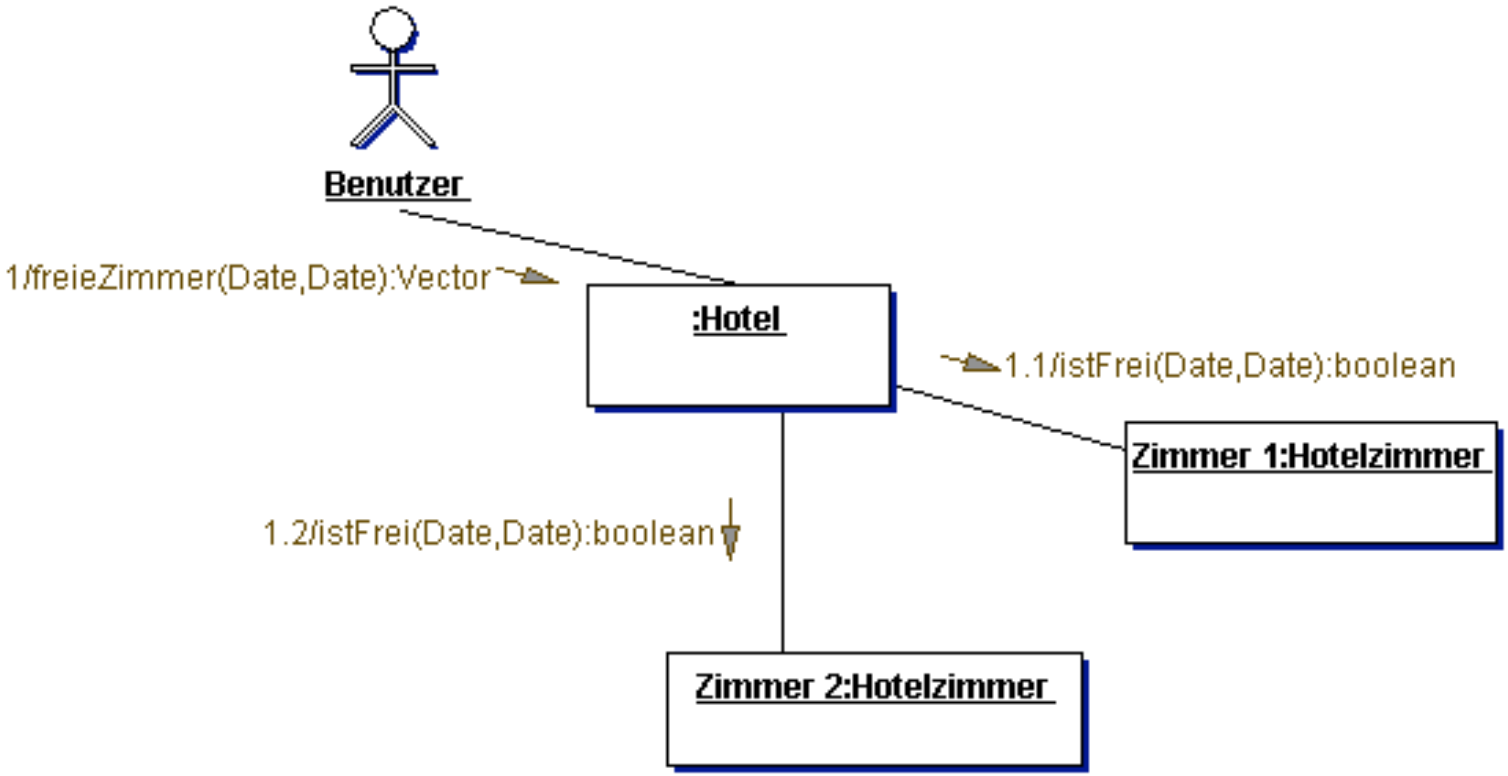
message, ...

message ist: **[no:] method()**

- **method()** wird wie bei Klassendiagrammen angegeben
- **no** ist eine optionale Nummer, die die Reihenfolge der Methodenaufrufe definiert.

Collaboration-Diagramm (II)

Beispiel:



Sequence-Diagramm (I)

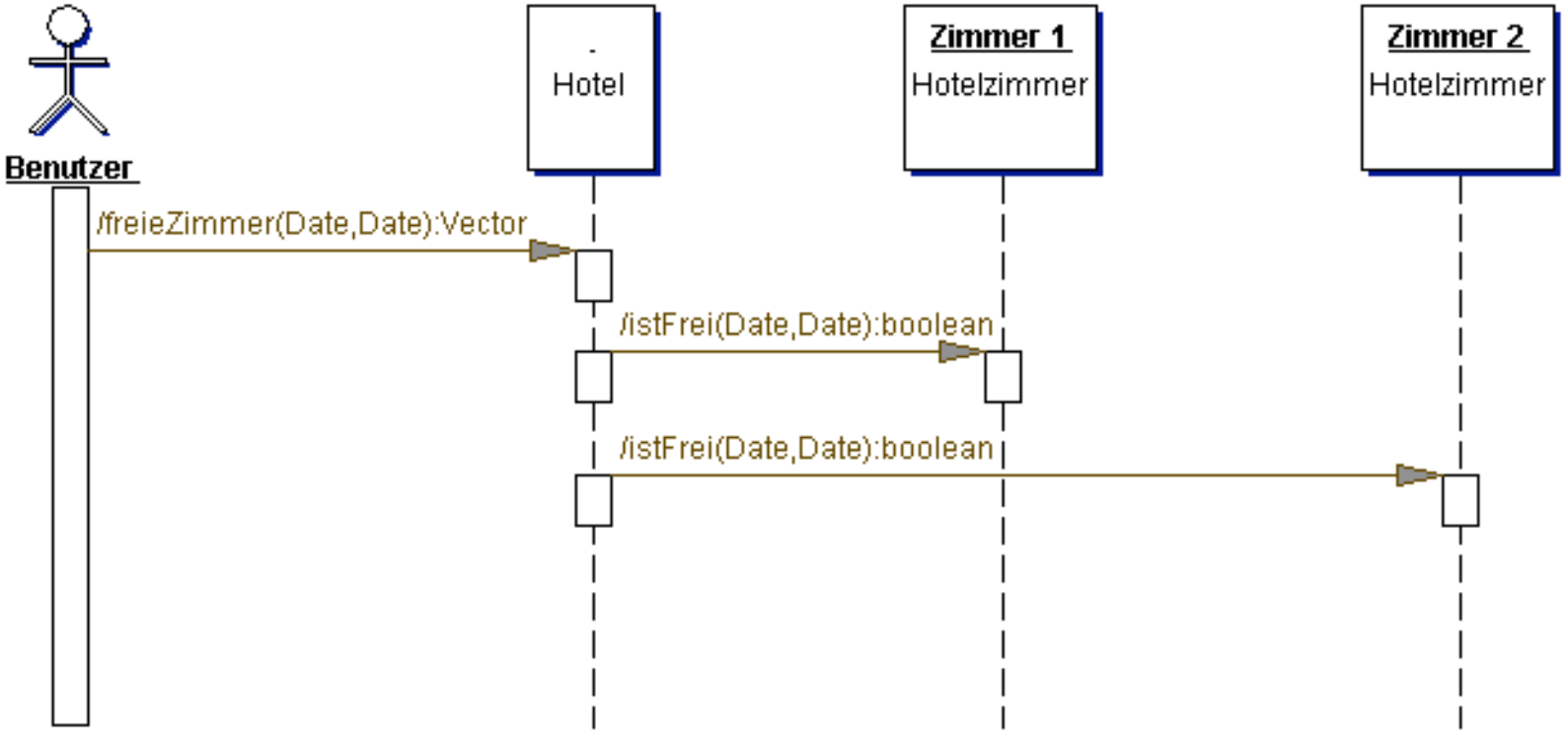
Ein *Sequence*-Diagramm drückt im wesentlichen die gleiche Semantik aus wie ein *Collaboration*-Diagramm, ist aber evtl. leichter zu lesen.

Collaboration-Diagramme bieten den Vorteil, daß zusätzliche Informationen darstellbar sind (zB Beziehungen zwischen Objekten).

Collaboration-Diagramme können automatisch in *Sequence*-Diagramme überführt werden.

Sequence-Diagramm (II)

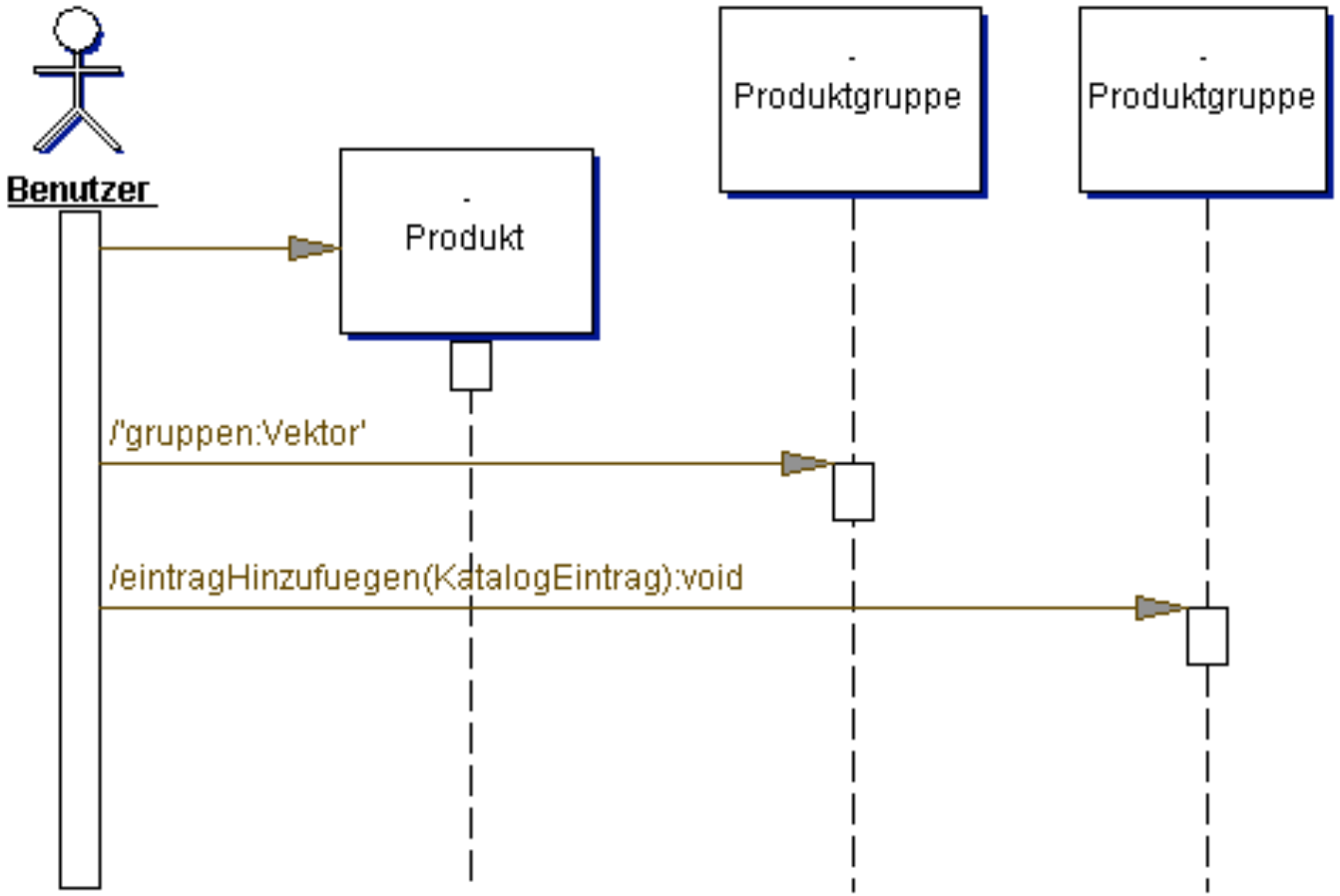
Beispiel:



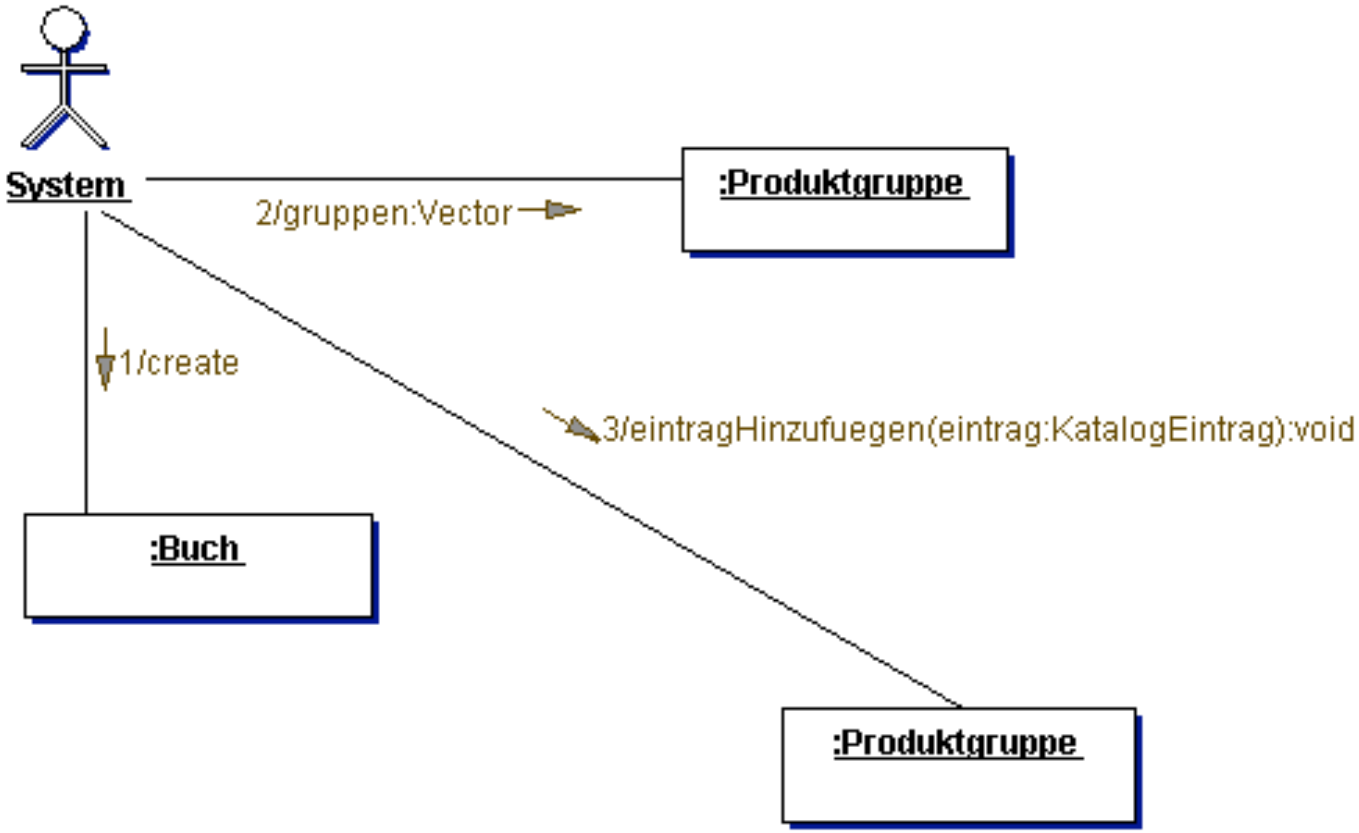
Case Study: Webshop (I)

- Katalog dynamisch:
  - Wie werden Produkte in den Katalog eingefügt?
  - Wie sieht das Sequence – Diagramm aus?
  - Wie sieht das Collaborations – Diagramm aus?

Case Study: Webshop (II)



Case Study: Webshop (III)



Use Cases

Use Case: Erstes Artefakt

- Use Cases helfen,
  - die Anforderungen eines Systems besser zu verstehen.
  - die Anforderungen eines Systems zu dokumentieren.

  Use Cases verbinden die verschiedenen Modelle eines Systems

Papierübung

- Lehrveranstaltungssystem
  - Professoren tragen ihre Veranstaltungen ein
  - Studenten wählen ihre Veranstaltungen
  - System berechnet Gebühren
- Was sind die Anforderungen?
- Wie könnte man die Anforderungen dokumentieren?
- Wie könnte man die Anforderungen visualisieren?

Use-Case: Kommunikations-grundlage

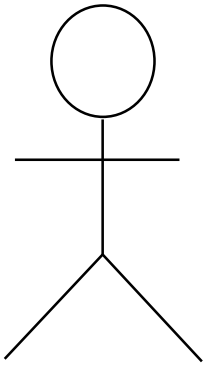
Use Cases (Szenarien) stellen ein **wichtiges Kommunikationsvehikel** dar, mit Hilfe dessen sich die Endanwender/Benutzer eines Systems und die Entwickler verständigen.

Bestandteile eines Use-Case-Modells:

- Systemfunktionen (**Use Cases**)
- Umgebung (**Actors**)
- Beziehungen zwischen Use Cases und Actors (**Use Case Diagrams**)

Actors

Darstellung in UML:



Es sollte nicht für jede Rolle, die jemand hat, ein Actor definiert werden.

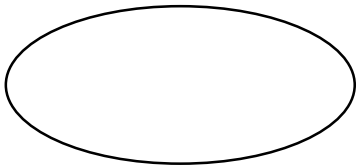
Beispiele für Actors:

- Studenten, die sich für Lehrveranstaltungen anmelden
- (externes) Abrechnungssystem
- Rezeptionist, der ein Hotelreservierungssystem bedient

Use Cases (I)

Ein Use Case **modelliert einen Dialog zwischen einem Actor und dem System.** Damit wird beschrieben, welche Funktionalität das System einem Actor bietet.

Darstellung in UML:



Use Cases (II)

Hilfreiche Fragen, um Use Cases zu definieren:

- Was sind die Aufgaben eines Actors?
- Wird ein Actor Informationen im System erzeugen, speichern, ändern, löschen oder lesen?
- Welche Use Cases werden diese Informationen erzeugen, speichern, ändern, löschen oder lesen?
- Muß ein Actor über bestimmte Ereignisse im System informiert werden?
- Können alle funktionalen Anforderungen mit den Use Cases erfüllt werden?

Use Cases (III)

Beispiel für die **Kurzbeschreibung eines Use Cases:**

Bezeichnung: Anmeldung zu einer Lehrveranstaltung (LVA)

Dieser Use Case wird durch einen Studenten gestartet. Es wird die Möglichkeit geboten, einen Stundenplan für ein bestimmtes Semester zu erstellen, löschen, ändern und/oder anzuschauen.

Ereignisfluß (Flow of Events)

- wird in Form eines Text-Dokuments (zB Word) beschrieben
- Vorschlag für eine Schablone:
  - Pre-Conditions
  - Main Flow und eventuelle Sub-Flows
  - Alternative Flows

Use Cases (IV)

Beispiel: Auswahl von LVAs (durch Professoren), die angeboten werden

- **Pre-Conditions**

Der Use Case “Anbieten von Kursen” muß ausgeführt sein, bevor dieser Use Case beginnt.

- **Main Flow**

Dieser Use Case beginnt, wenn sich ein Professor im LVA-Verwaltungssystem anmeldet und sein/ihr Passwort eingibt. Das System verifiziert, ob das Passwort gültig ist (E-1) und fordert den Professor auf, das aktuelle Semester oder ein künftiges Semester auszuwählen (E-2). Danach wählt der Professor die gewünschte Tätigkeit: Hinzufügen, Löschen, Anzeigen, Drucken oder Beenden.

Use Cases (V)

Wenn Hinzufügen gewählt wurde, wird der Sub-Flow S-1: *Hinzufügen eines LVA-Angebots* ausgeführt.

...

- Sub-Flows

S-1: Hinzufügen eines LVA-Angebots

Über entsprechende Eingabefelder können LVA-Bezeichnung und Nummer eingegeben werden (E-3). Das System verbindet den Professor mit der angebotenen LVA (E-4). Der Use Case beginnt von neuem.

...

- Alternative Flows

E-1: Ein falscher Name oder ein flasches Passwort wurden eingegeben. Der Benutzer kann beides neu eingeben oder den Use Case beenden.

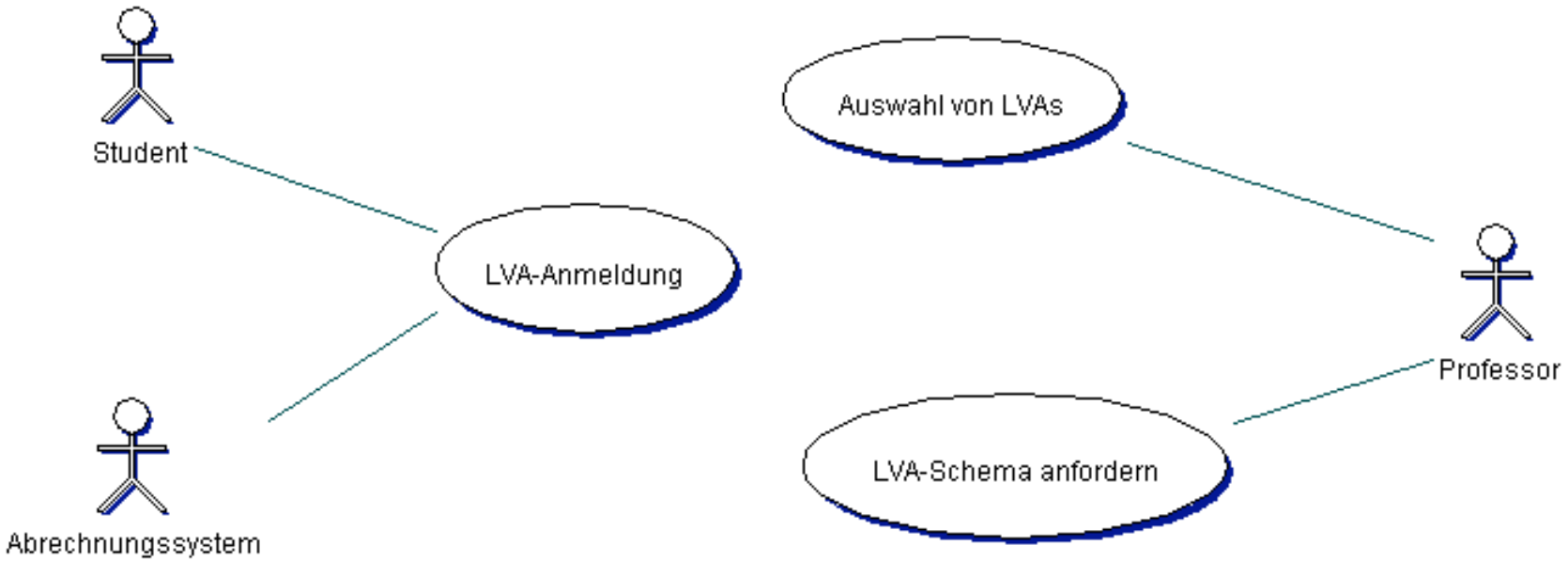
Use Case Diagramm (I)

Diese zeigen bestimmte oder alle Actors, Use Cases sowie Beziehungen zwischen diesen Entitäten. Typischerweise gibt es

- ein Main Use Case Diagram, welches die wichtigsten Actors und die Hauptfunktionalität grafisch darstellt
- beliebig viele weitere Use Case Diagrams, zB
  - ein Diagramm, welches alle Use Cases für einen bestimmten Actor zeigt
  - ein Diagramm, welches einen Use Case und alle seine Beziehungen zeigt

Use Case Diagramm (II)

Beispiel:



Use Case Diagramm (III)

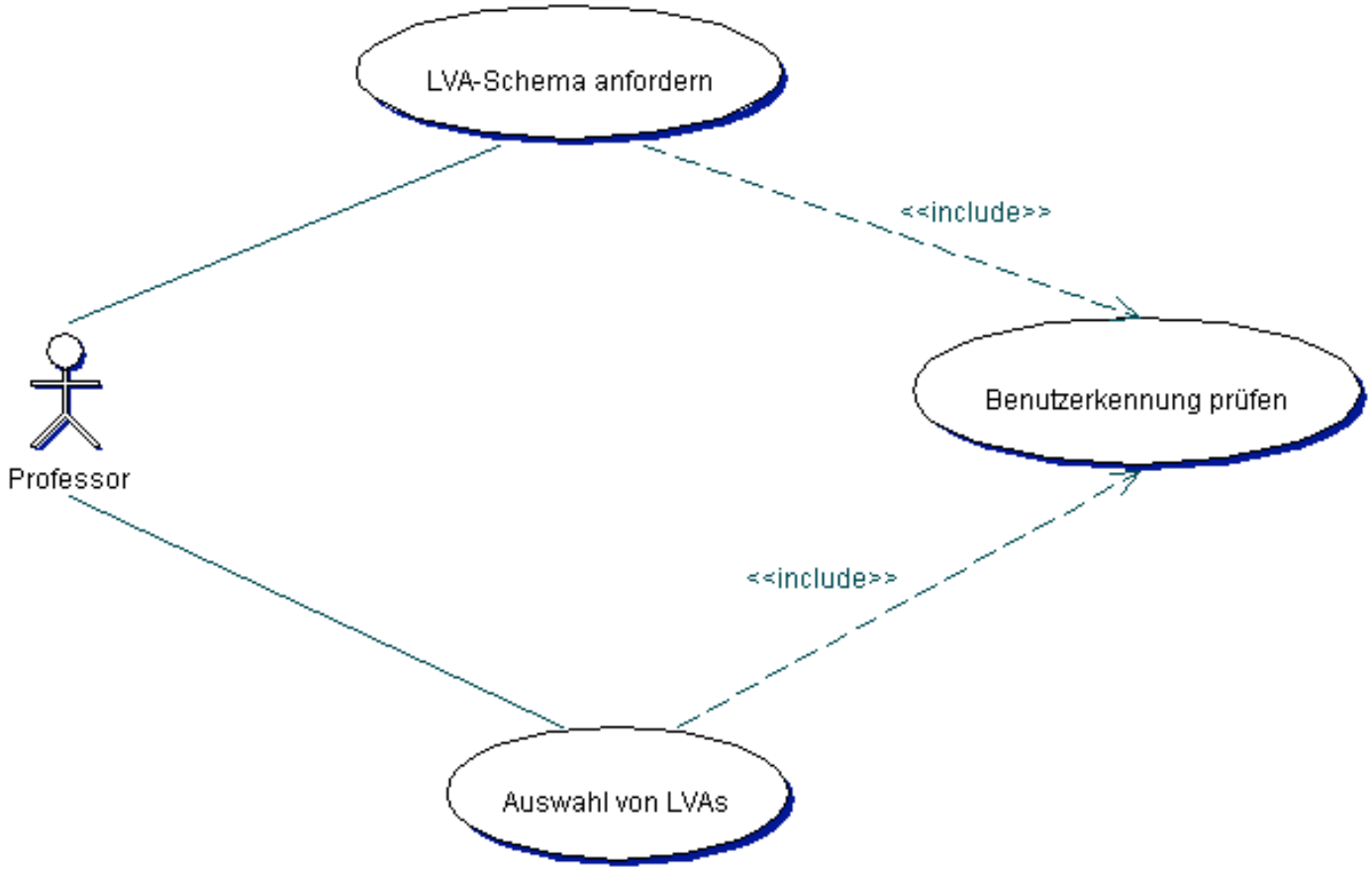
- Die “Uses”-Beziehung zeigt, daß Funktionalität in mehreren Use Cases benötigt wird.
- Die “Extends”-Beziehung drückt optionales Verhalten eines Use Cases aus.

Beide Beziehungen werden durch einen Abhängigkeits-Pfeil dargestellt und durch Stereotypen-Namen bezeichnet:

In UML gibt es das sogenannte **Stereotype-Konzept**, mit Hilfe dessen die grundlegenden Modellierungselemente erweitert werden können. Die Namen von Stereotypen sind zwischen << und >>. Stereotypen werden zB benutzt, um die Beziehungen zwischen Use Cases zu beschreiben.

Use Case Diagramm (IV)

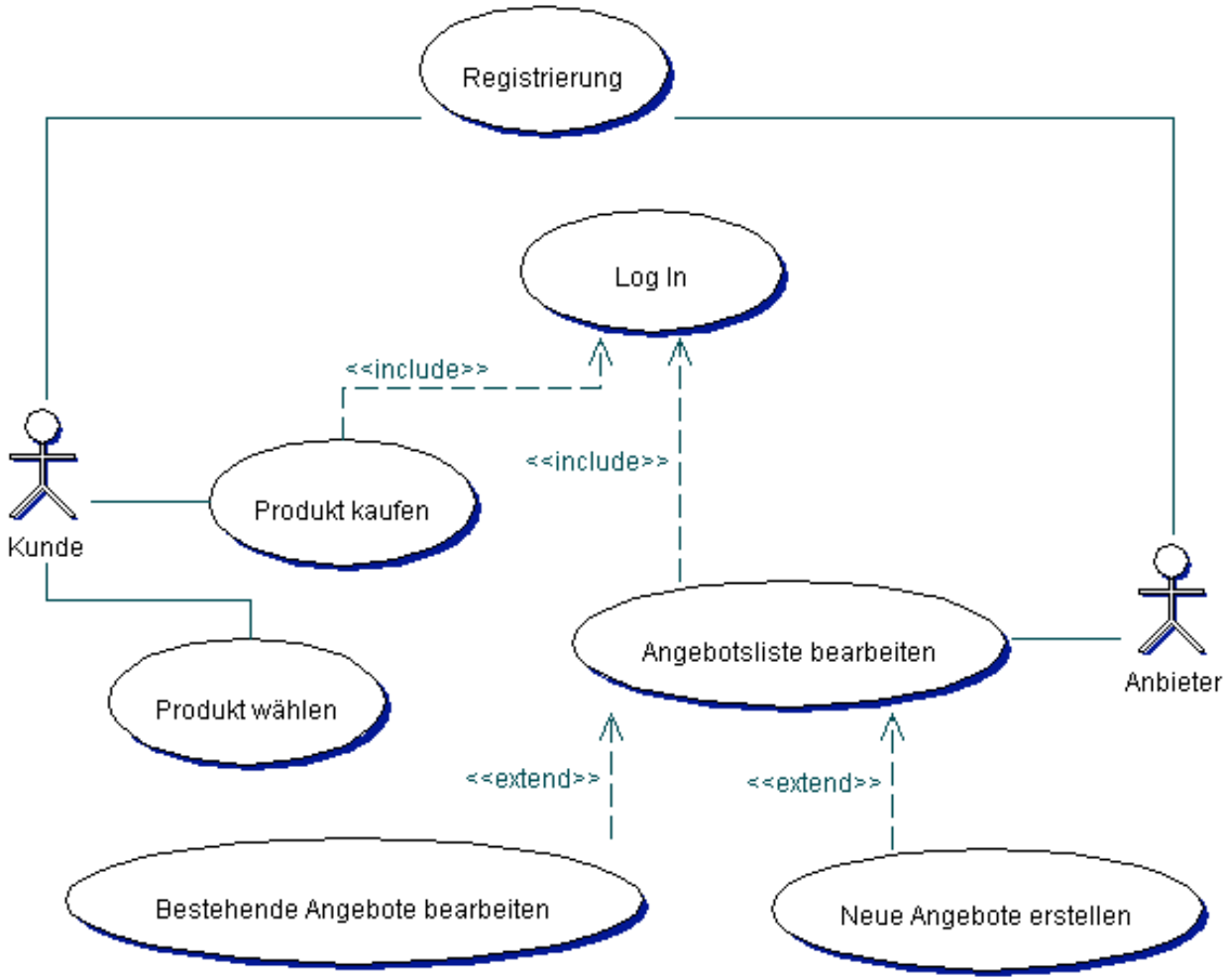
Beispiel:



Hands-On Übung

- Webshop
  - Vgl. Amazon.com
  - Kunde surft durch das Angebot, wählt Produkt, bezahlt mit Karte oder Bankeinzug...
  - Anbieter kann neue Produkte in den Katalog stellen...

Hands-On Übung



Hands-On Übung: Registrieren

- Mainflow

  Der Use Case beginnt, wenn der Benutzer den Punkt „registrieren“ wählt. Das System fordert ihn auf, ein Formular auszufüllen, in dem er seinen Namen, Adresse, Alter, Nickname und Passwort angibt(E-1). Danach schickt das System eine E-Mail an den Benutzer und zeigt ihm an, dass er sich erfolgreich registriert hat.

Case Study: Registrieren

- Alternative Flows

  E-1: Falls der Benutzer das Formular unvollständig ausgefüllt hat, wird er aufgefordert, die unausgefüllten Felder auszufüllen.

  E-1: Falls ein Nickname vom System schon vergeben wurde, wird er aufgefordert, einen anderen Nickname zu wählen

  ...

CRC-Karten

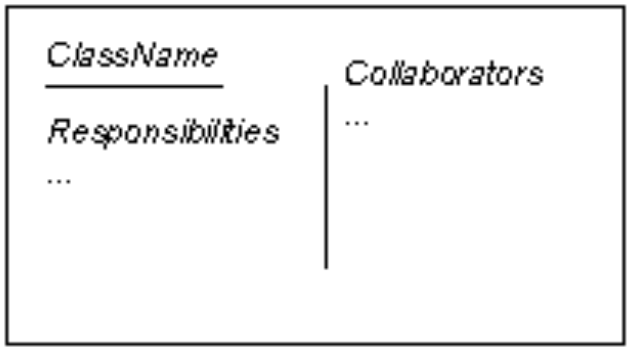
CRC Karten(I)

- Welche Klassen werden gebraucht, um ein Szenario zu modellieren?
- Wie arbeiten diese Klassen zusammen?

CRC Karten(II)

- Class, Responsibility, Collaboration
- Beck und Cunningham, OOPSLA89
  - Entwickelten CRC – Karten, um den Paradigmenwechsel (prozedural → OO) anschaulich unterrichten zu können.
  - Unmittelbare Einführung in die Idee von „Verantwortungen“ (Wirfs-Brock 1990)

CRC-Karten(III)



- 4x6 Index Karten
- Spezifizieren:
  - Klassennamen
  - Verantwortungen
  - Zusammenarbeit

Beispiel:

Hotel	
Verwalte Kunden	Kunde
Verwalte Zimmer	Hotel-zimmer

Hotelzimmer	
Belegungsplan	Date, Reser-vierung
Erstelle Rechnungen	Kunde, Date

CRC Karten(IV)

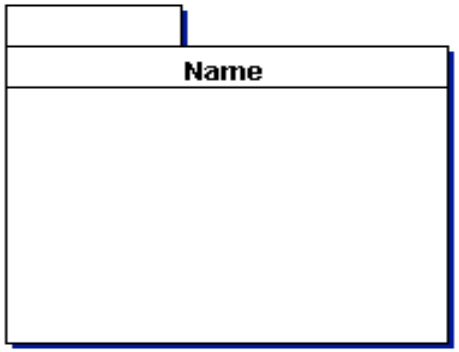
- Vorteile
  - Kommunikation zwischen Designern
  - Weg von Datenbehältern – hin zu „Verantwortungen“
  - Kollaboration zwischen Klassen wird leichter verstanden (kann nachgespielt werden)
  - Grösse der Karten sorgt für richtige Granularität der Klassen und forciert eine High-Level Spezifikation der Klassen

Pakete und Paketdiagramme

Pakete

Pakete weden verwendet, um Klassen zu gruppieren. Klassen, die logisch zusammengehören, werden in Paketen strukturiert.

UML Notation:



Pakete(II)

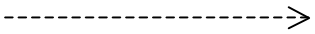
Pakete können geschachtelt sein, um komplizierte Architekturen besser strukturieren zu können.

In UML können optional die Klassennamen, die zu einem Paket gehören, aufgelistet werden.

Paketdiagramme

Folgende Beziehungen zwischen Paketen können definiert werden:

- Abhängigkeit

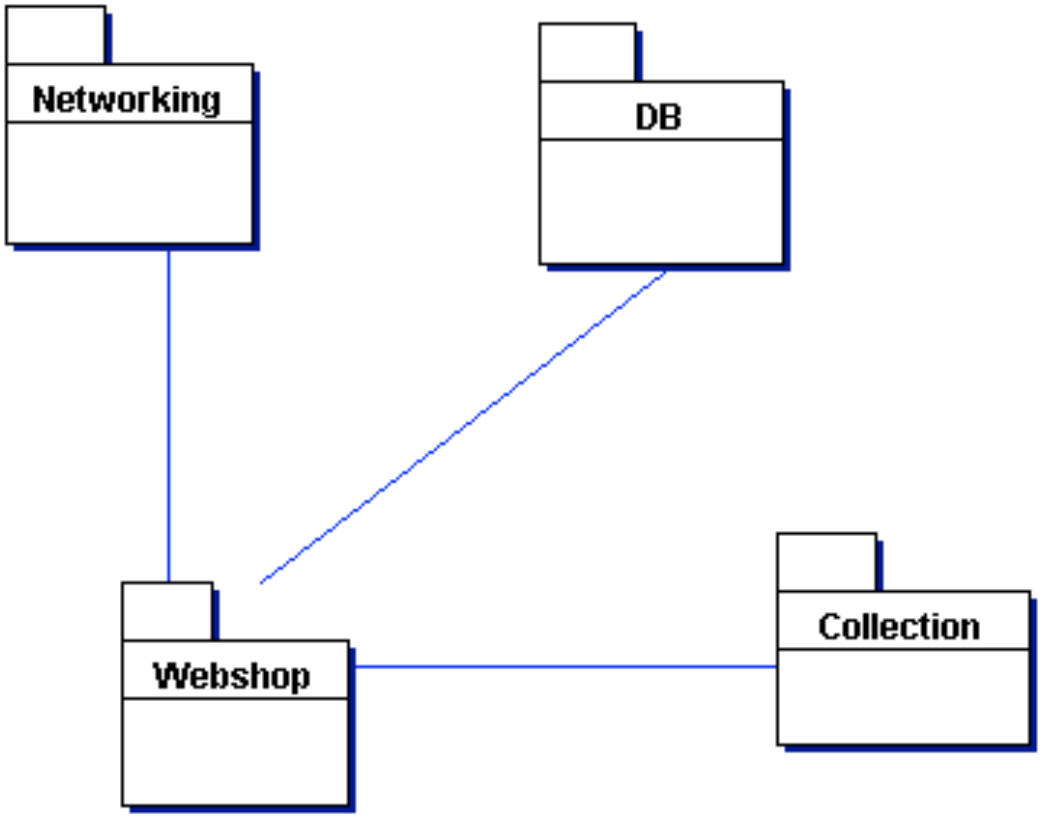


Wird verwendet, um auszudrücken, daß die Klassen des einen Pakets Klassen des anderen Pakets verwenden

- Verallgemeinerung

Wird verwendet, wenn die Klassen des einen Pakets die Verträge der Klassen des anderen Pakets erfüllen

Beispiel: E-Commerce-Anwendung

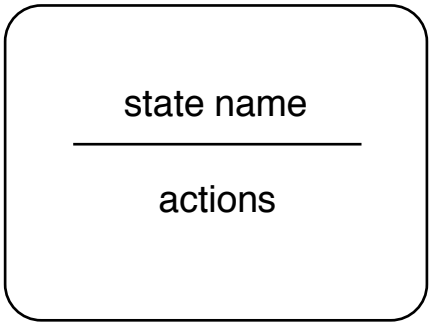


Zustands-übergangs-diagramme

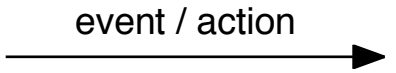
Notationselemente (I)

Zustandsübergangsdiagramme zeigen das dynamische Verhalten einer Klasseninstanz oder eines ganzen Systems.

- Symbol für einen Zustand:



- Symbol für den Zustandsübergang:



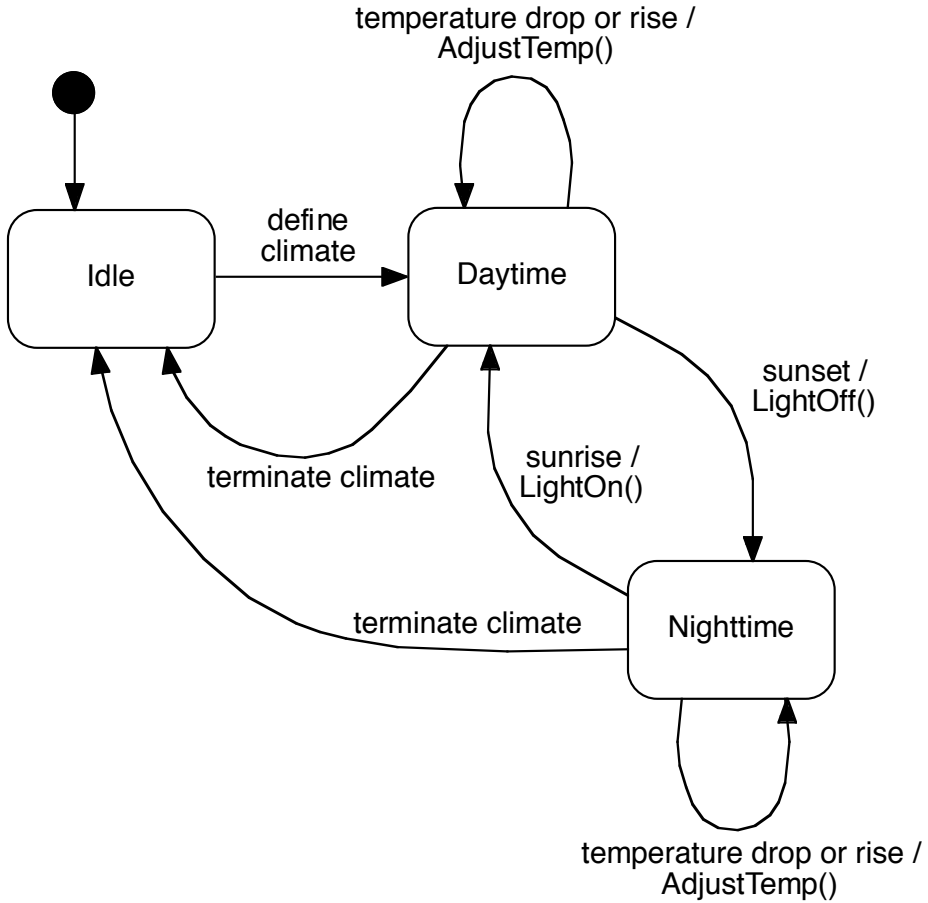
Notationselemente (II)

Eine Aktion kann wie folgt geschrieben werden:

- converter.ReadFile() method call
- DeviceFailure event triggering
- start Converting begin some activity
- stop Converting stop some activity

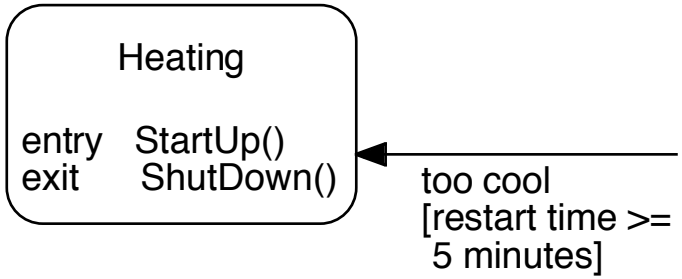
Beispiel

Controller in einem Glashaus:



Notationszusätze (I)

- Innerhalb eines Zustandes können Aktionen definiert werden,
  - wenn das System in diesen Zustand kommt, bzw. ihn verläßt:



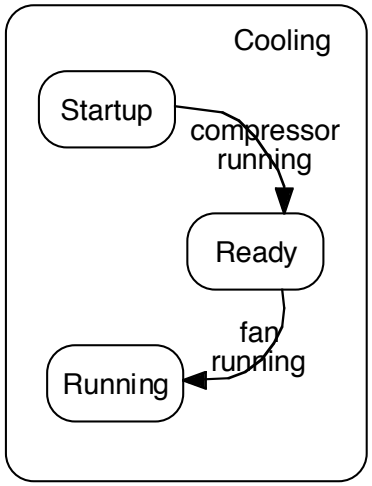
  - sich in einem Zustand befindet:
    do Heating
- Zustandsübergänge können an Bedingungen geknüpft werden, die in eckigen Klammern angegeben werden.

Notationszusätze (II)

- Bedingungen können auch Zeitlimits enthalten:

  timeout(Heating, 30s) TRUE, wenn System länger als 30 Sek. im Zustand Heating ist

- Zustände können beliebig geschachtelt werden:



Notationszusätze (III)

- Zustände mit “Gedächnis”:

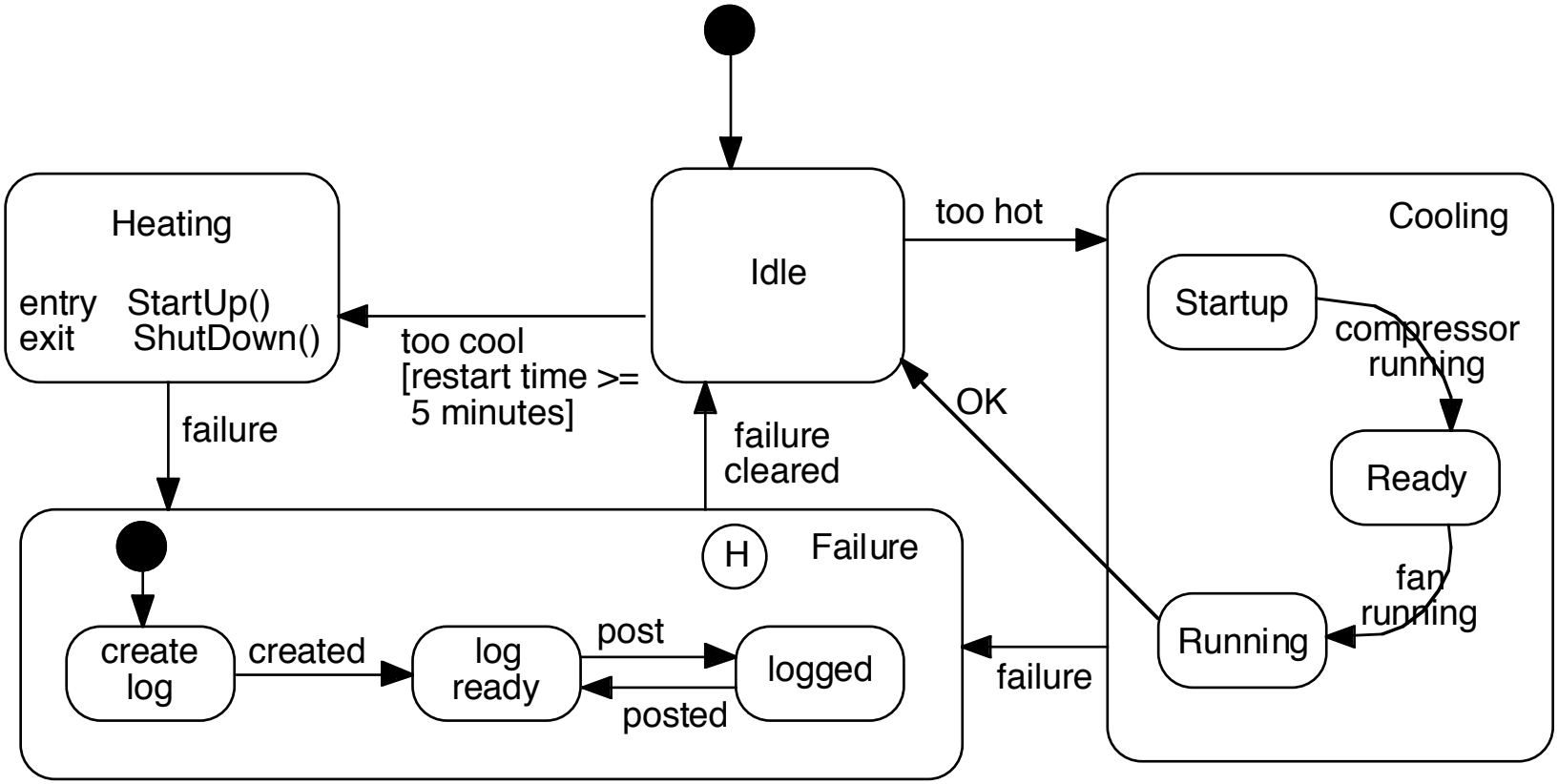
  Ein Zustand, der weitere Unterzustände enthält, kann ein "Gedächnis" bekommen, also sich merken, in welchem Unterzustand er sich befand, wenn der Zustand verlassen wird.

  Dies wird durch das Adornment

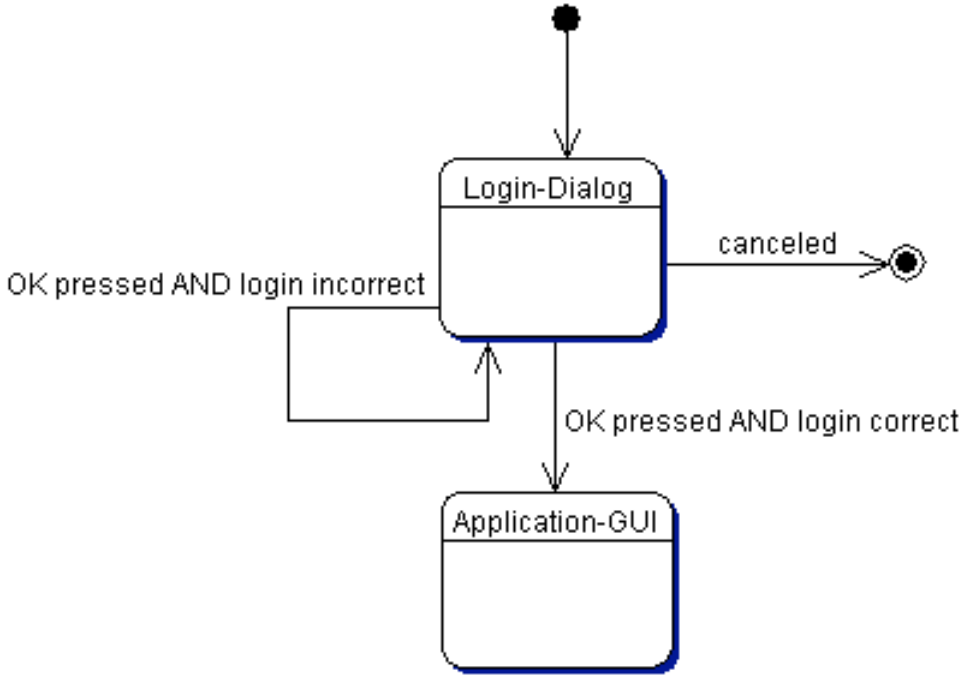
  ausgedrückt.

Beispiel


Heating
entry StartUp()
exit ShutDown()
Idle
too hot
Cooling
Startup
compressor running
Ready
fan running
Running
too cool
[restart time >= 5 minutes]
failure
failure cleared
OK
H
Failure
create log
created
log ready
post
posted
logged
failure

Beispiel: GUI

Abfolge von Dialogen als Zustandsübergangsdiagramm:

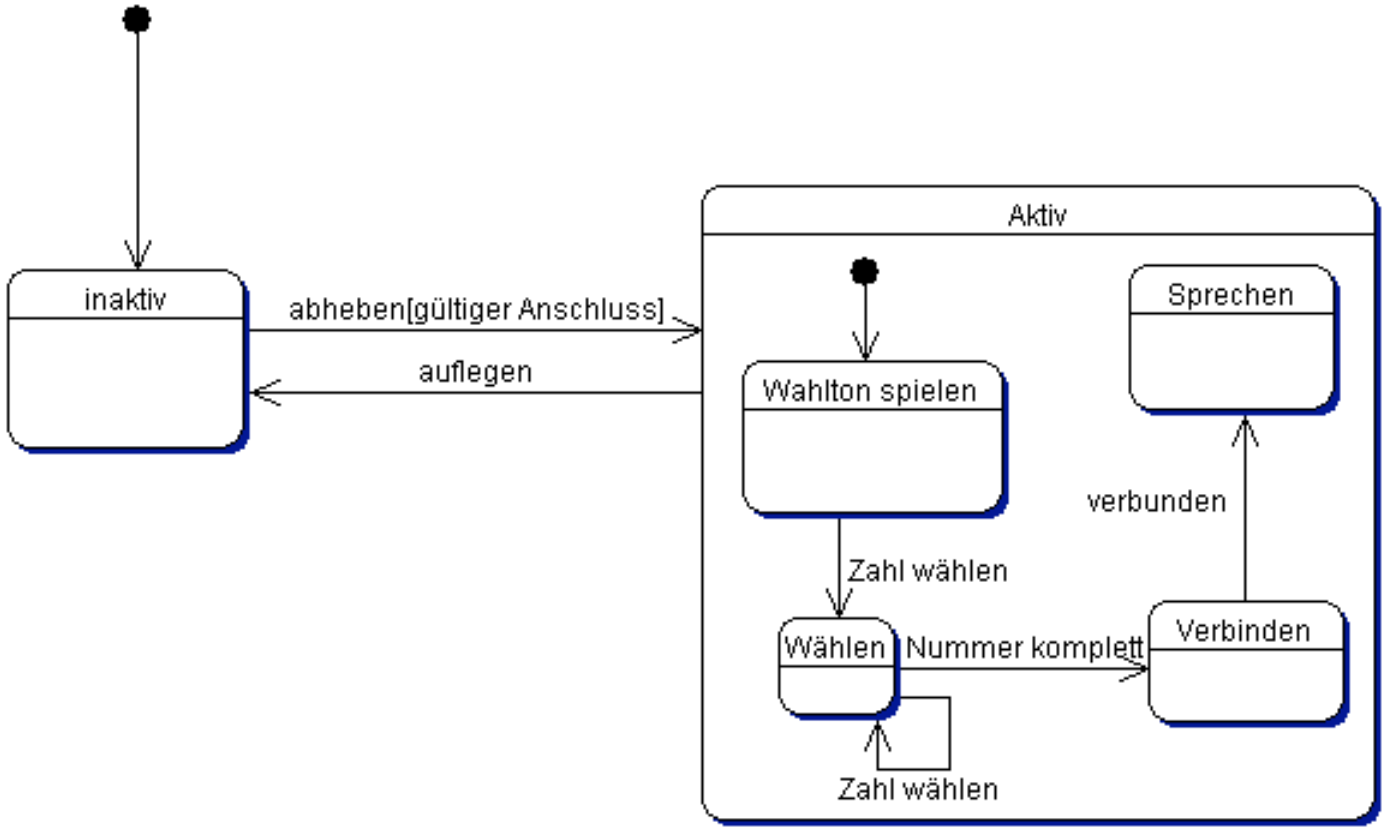


Hands-On Übung

Hands-On Übung

- Welche Zustände kann ein Telefon haben?
- Gibt es Unterzustände?
- Weche Zustandsübergange gibt es?
- Gibt es Bedingungen für diese?

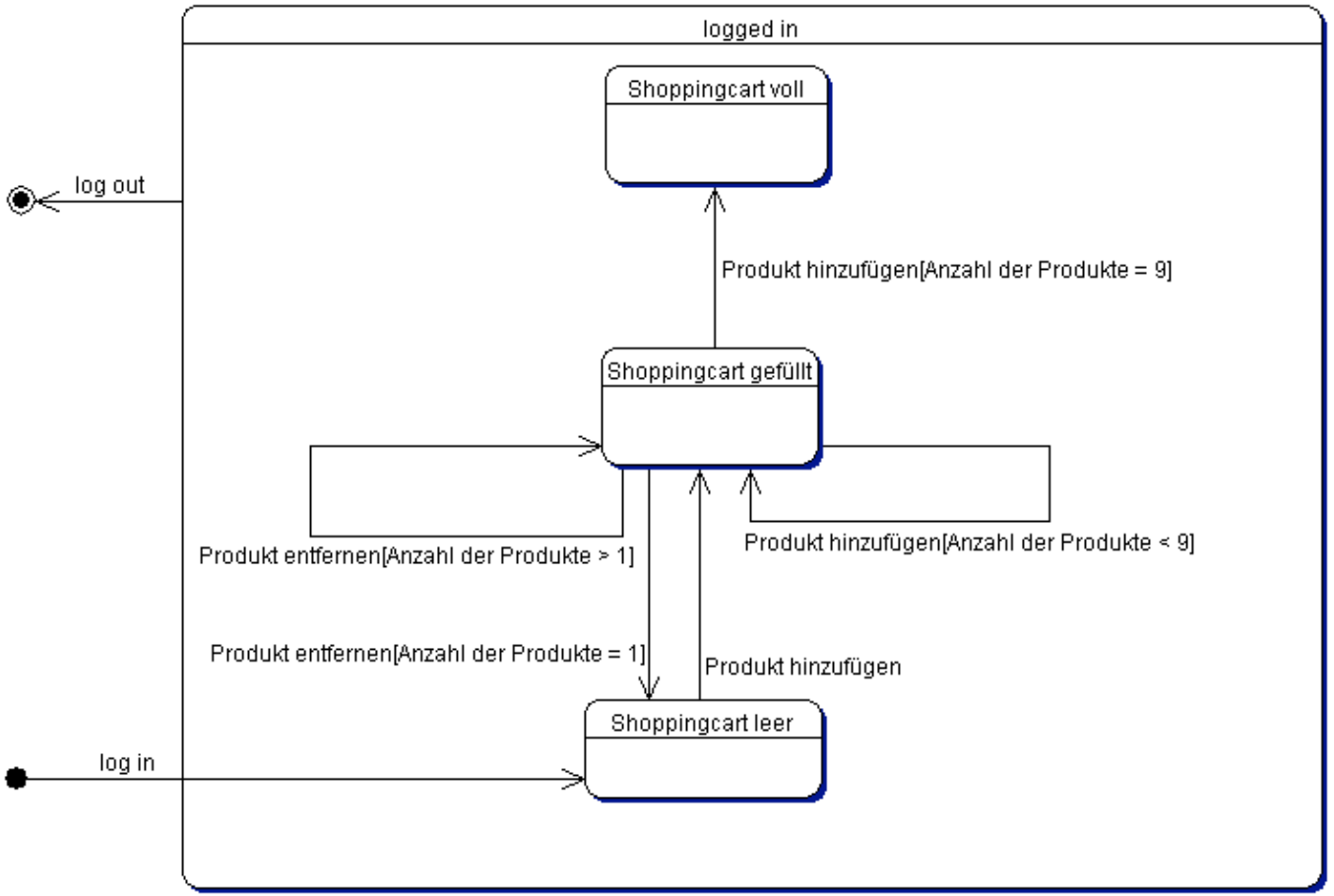
Hands-On Übung



Case Study: Webshop (I)

- Shoppingcart
  - Welche Zustände hat ein Einkaufswagen?
  - Um es spannender zu machen; es sollen nie mehr als 10 Elemente im Wagen sein.

Case Study: Webshop (II)



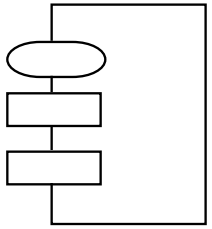
SOFTWARE RESEARCH LAB

Component-Diagramme

Notation

Klassen entsprechen Komponenten. Analog zu Packages können mehrere Klassen in eine Komponente zusammengefasst werden.

In UML-Notation wird eine Komponente wie folgt dargestellt:



Komponenten entsprechen Modulen in modulorientierten Sprachen.

C++: Nachbau von Modulen durch .h, .C Files

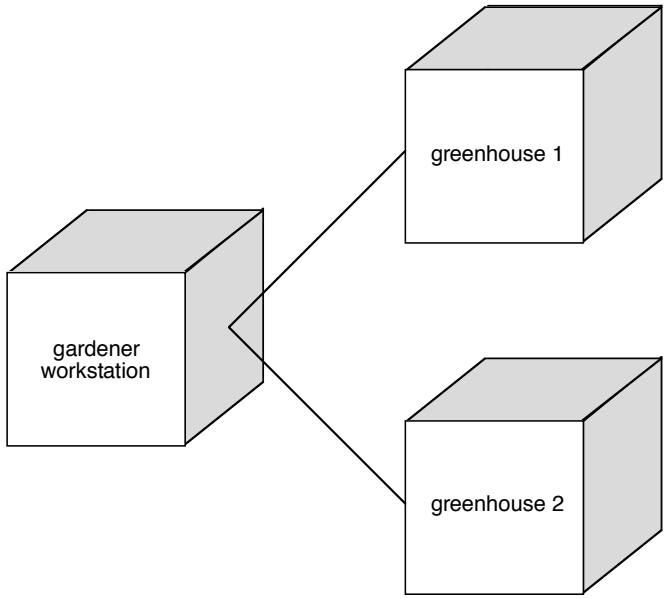
Smalltalk: Klassengruppen, keine Module

Oberon und Java: Modularisierung direkt durch die Sprache unterstützt

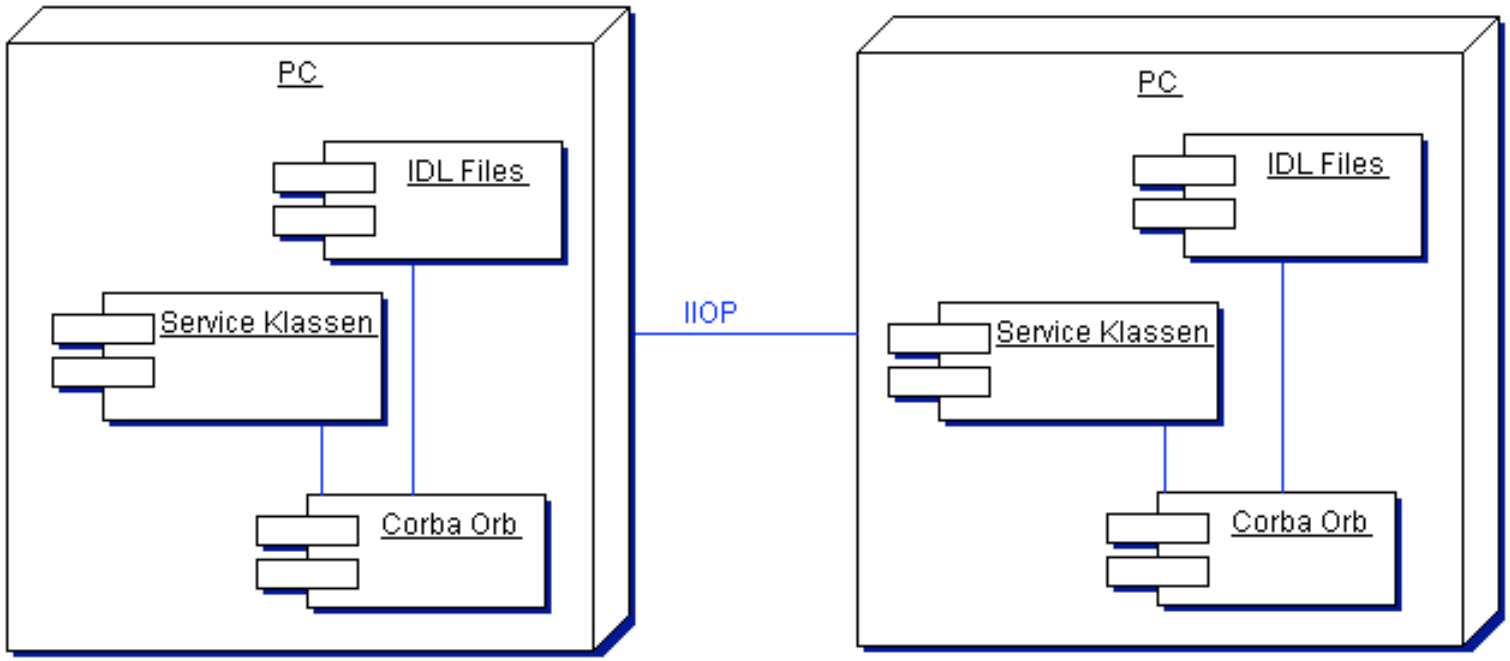
Deployment-Diagramm

Notation

Diese Darstellung ist aus Booch's Prozeßdiagramm entstanden. Sie drückt aus, welche Hauptprogramme bzw. welche aktiven Objekte welchen Prozessoren zugeordnet sind, wenn ein **System auf mehreren Prozessoren** verteilt läuft.



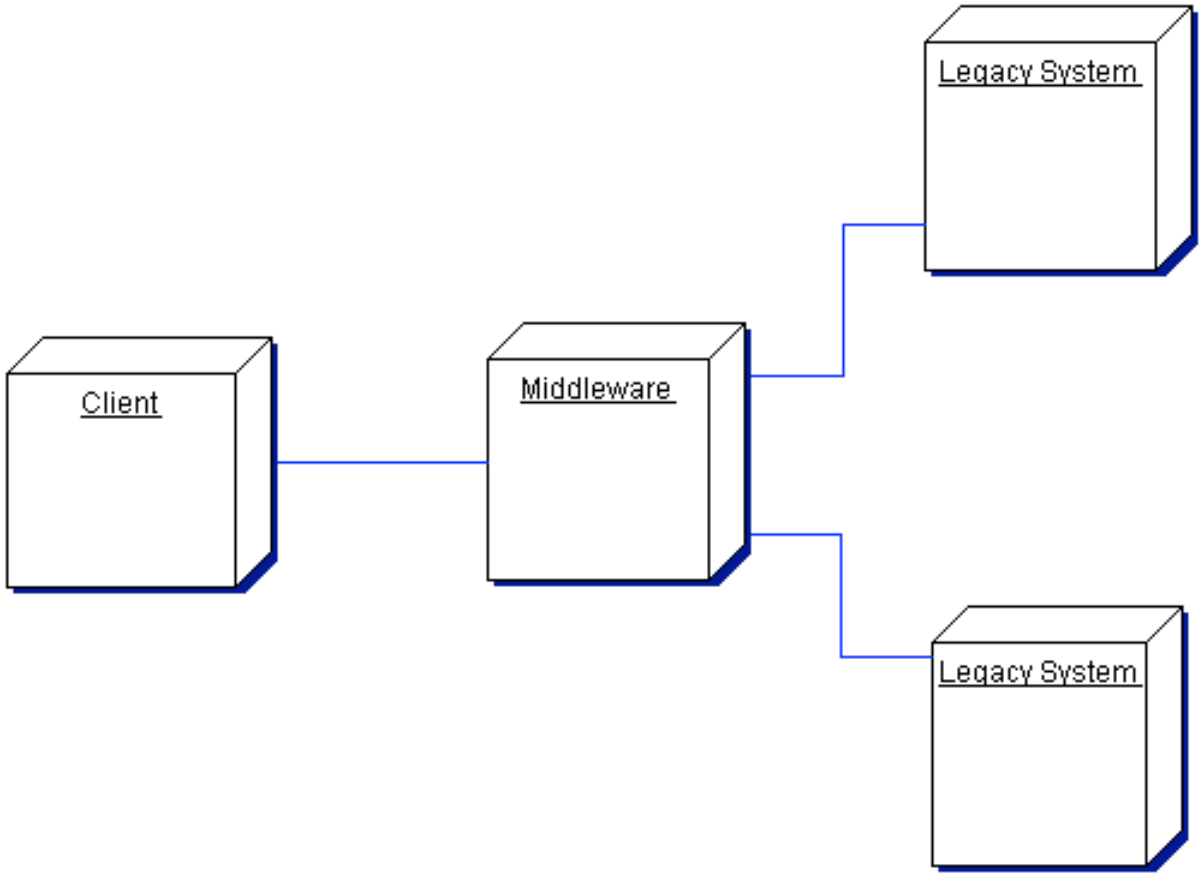
Beispiel: CORBA



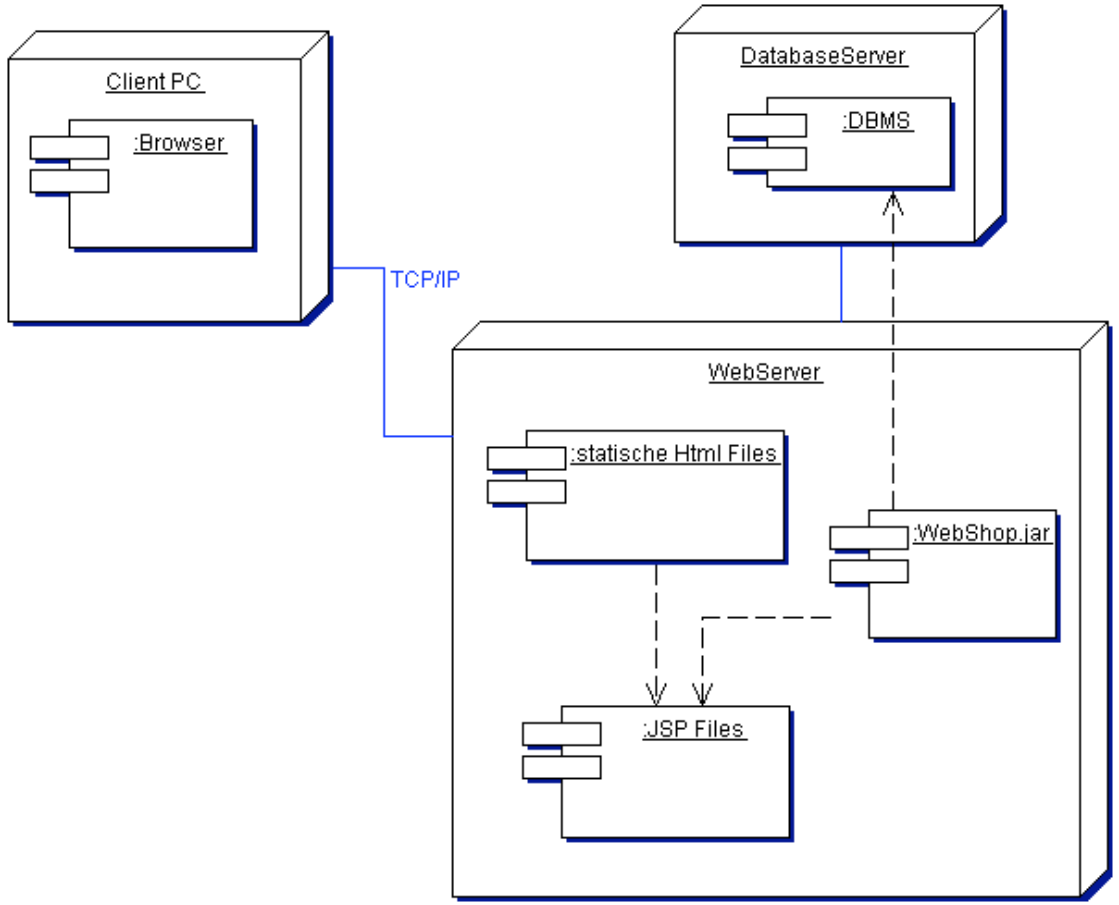
Hands-On Übung: Webshop

- Ein Webshop ist typischerweise eine verteilte Anwendung. Normalerweise sind drei Schichten beteiligt.
- Wie könnte die Topologie des Systems aussehen?
- Welche Komponenten befinden sich auf welchen computational-nodes?

3-tier Applikationen



Webshop: Topologie



Anhang A
Literaturhinweise

Literaturhinweise (I)

Booch G., Jacobson I, Rumbaugh J. (1999)

Objektorientierte Analyse und Design. Mit praktischen Anwendungsbeispielen

Das UML-Benutzerhandbuch (Unified Modeling Language User Guide)

Unified Modeling Language Reference Manual,

The Objectory Software Development Process,

Addison-Wesley

Österreich B. (2000) Erfolgreich mit Objektorientierung, Oldenburg Verlag

Balzert H. (2000) Objektorientierung in 7 Tagen, m. CD-ROM. Vom UML-Modell zur fertigen Web-Anwendung, Spektrum Akadem. Verlag

Budd T. (1997) An Introduction to Object-Oriented Programming, Addison-Wesley

Fayad M., Schmidt D., Johnson R. (1999) Building Application Frameworks: Object-Oriented Foundations of Framework Design, Wiley

Fayad M., Schmidt D., Johnson R. (1999) Implementing Application Frameworks: Object-Oriented Frameworks at Work, Wiley

Fayad M., Schmidt D., Johnson R. (1999) Domain-Specific Application Frameworks: Manufacturing, Networking, Distributed Systems, and Software Development, Wiley

Fowler M. (1997) Analysis Patterns, Addison-Wesley.

Fowler M. (1999) UML Distilled (= UML konzentriert)

Literaturhinweise (II)

Gamma E., Helm R., Johnson R. and Vlissides J. (1995). Design Patterns—Elements of Reusable OO Software. Reading, MA: Addison-Wesley (auch als CD verfügbar)

Pree W. (1995). Design Patterns for Object-Oriented Software Development. Reading, Massachusetts: Addison-Wesley/ACM Press

Goldberg A., Rubin K. (1995) Suceeding with Objects—Decision Frameworks for Project Management, Addison-Wesley

Fontoura M., Pree W., Rumpe B. (2001) The UML-F Profile for Framework Architectures, Addison Wesley

Larman C. (1998) Applying UML And Patterns

Rumbaugh J. (1992) Object-Oriented Modeling and Design, Prentice-Hall

http://www.rational.com (diverse aktuelle Informationen zu Rational Tools, UML)

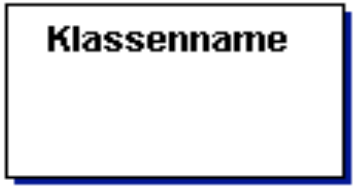
http://www.together.com (diverse aktuelle Informationen zu Together Tools, UML)

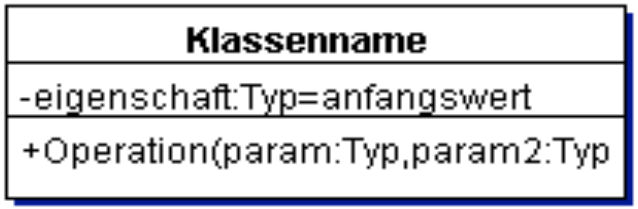
SOFTWARE RESEARCH LAB

Anhang B
UML Essentials

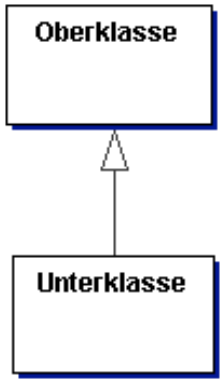
Anhang:
UML: Klasse, Vererbung

Klassen:



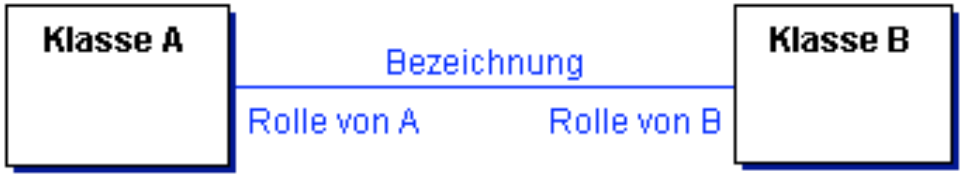


Vererbung:

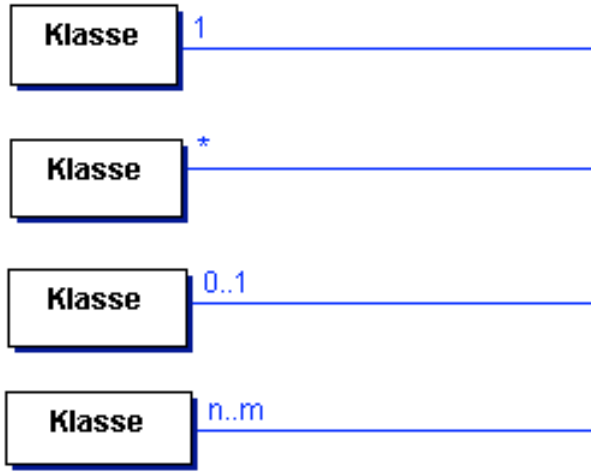


Anhang:
UML: Klassenbeziehungen

Association:



Vielfachheiten:

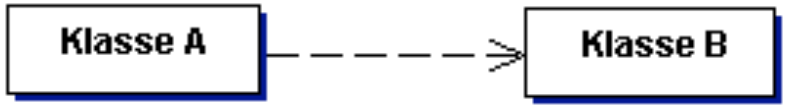


Anhang:
UML: Klassenbeziehungen

Aggregation:

Navigierbarkeit:

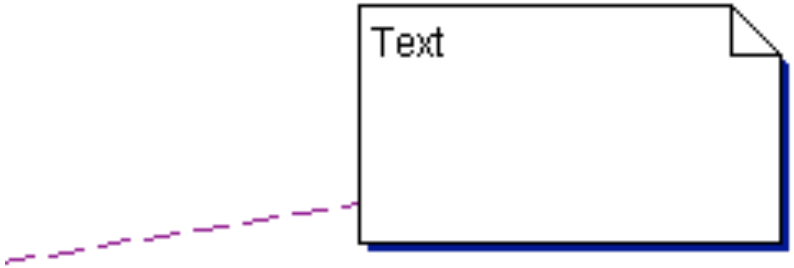
Abhängigkeit:



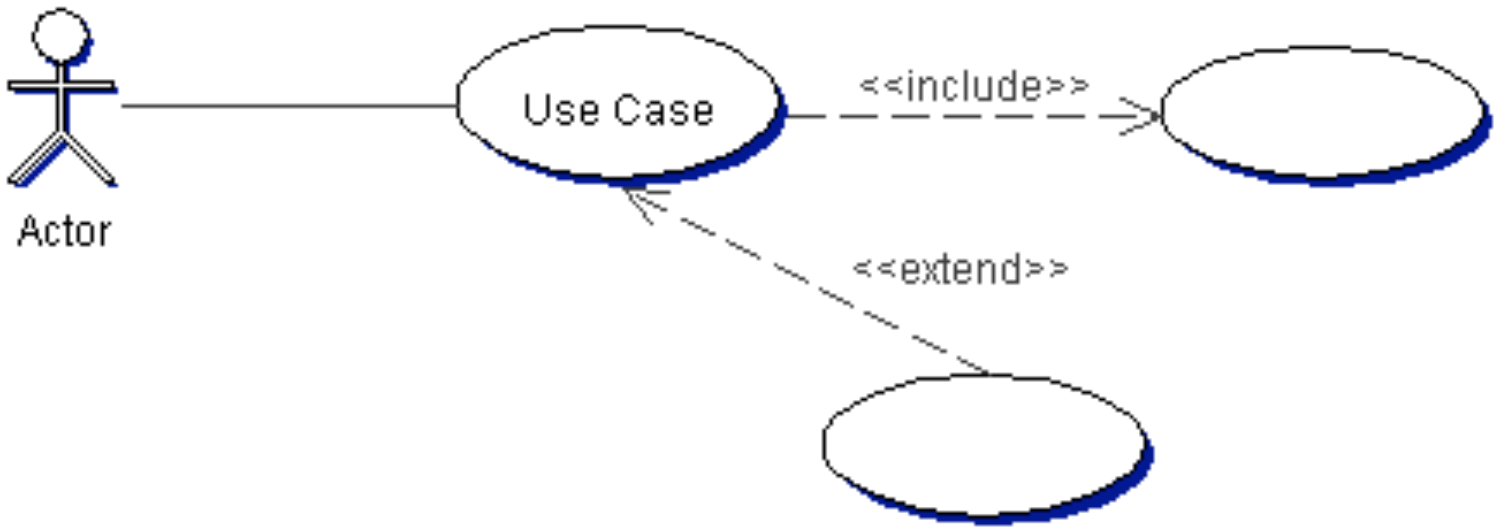
Anhang:
UML: Objekte, Bemerkungen

Objekt:

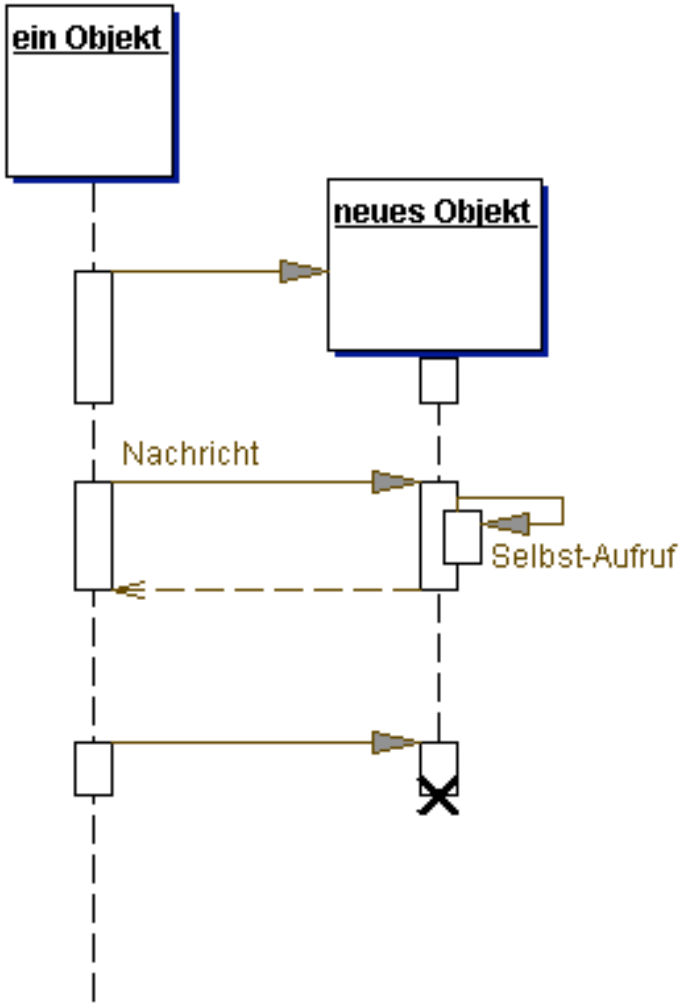
Bemerkung:



Anhang:
UML: Use Case Diagramm

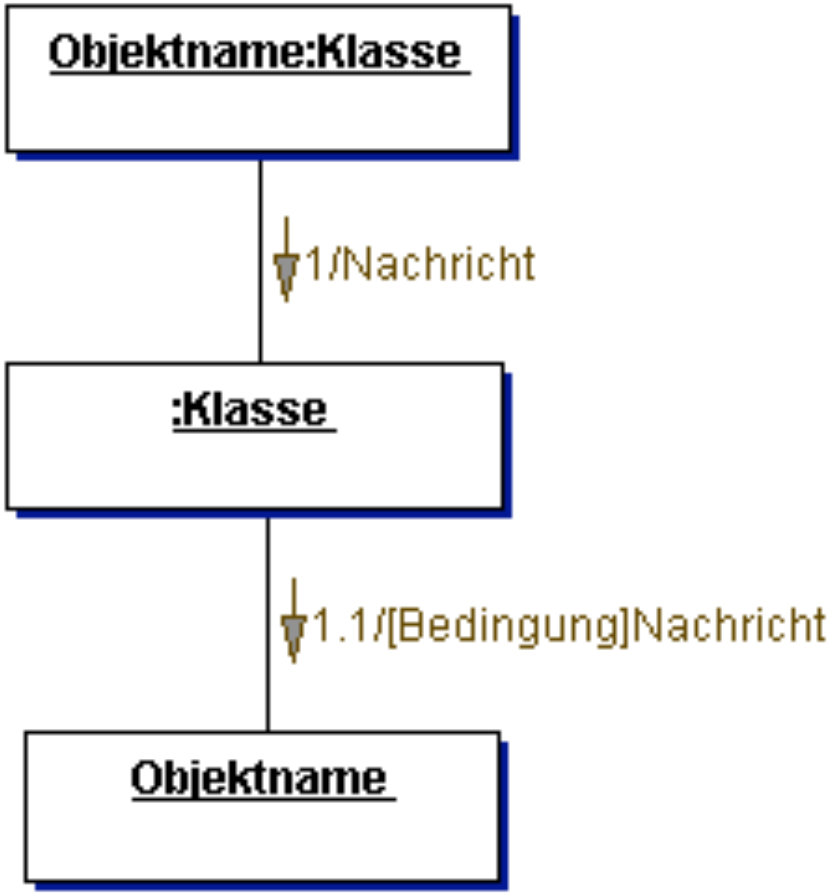


Anhang:
UML: Sequence Diagramm

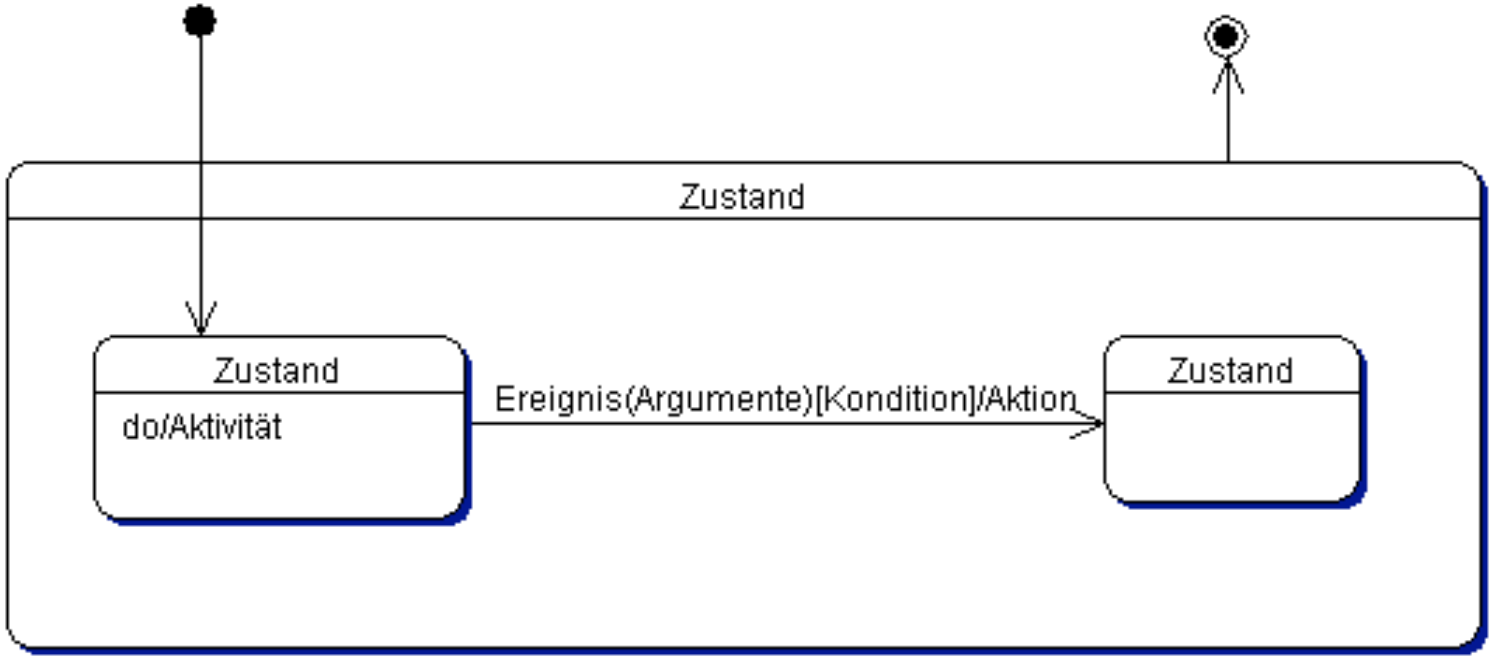


Anhang:

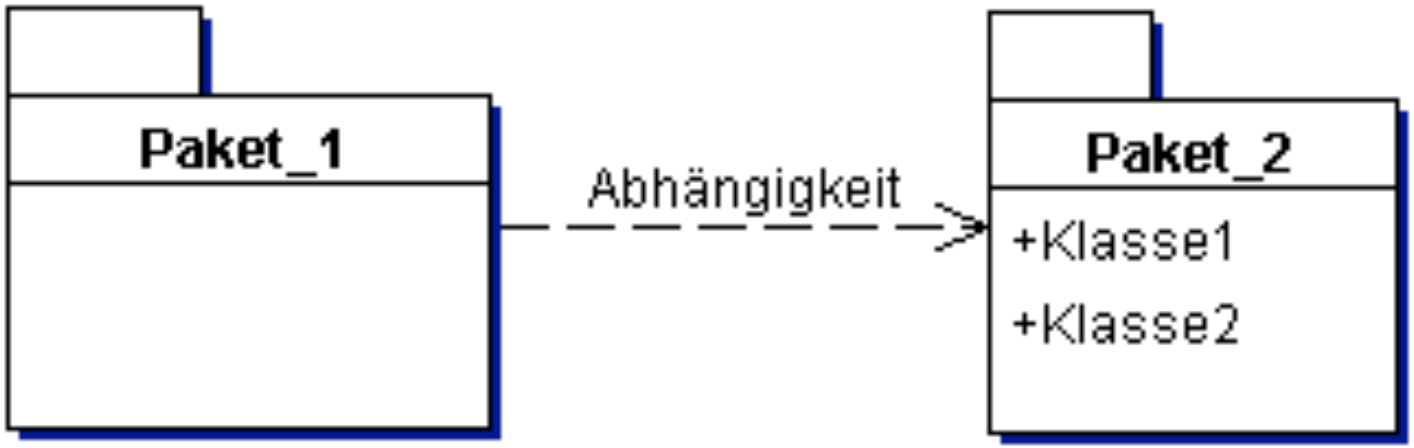
UML: Collaboration Diagramm



Anhang:
UML: Zustandsdiagramm

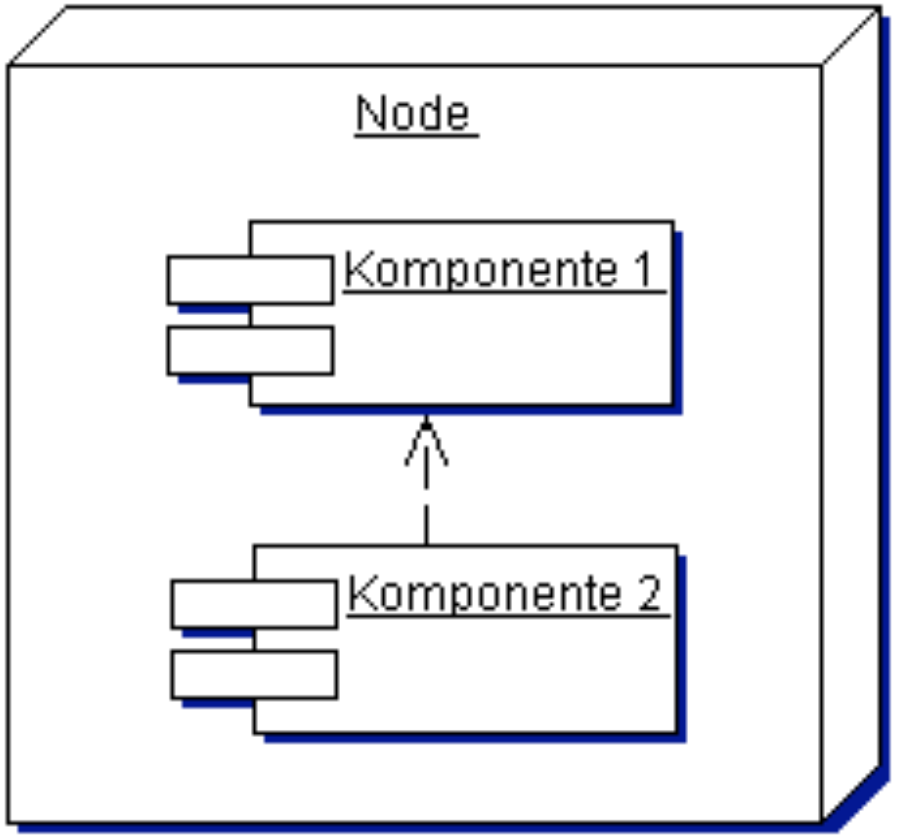


Anhang:
UML: Paketdiagramm



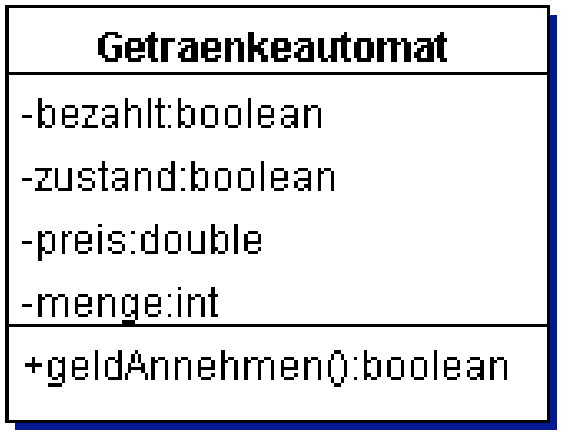
Anhang:

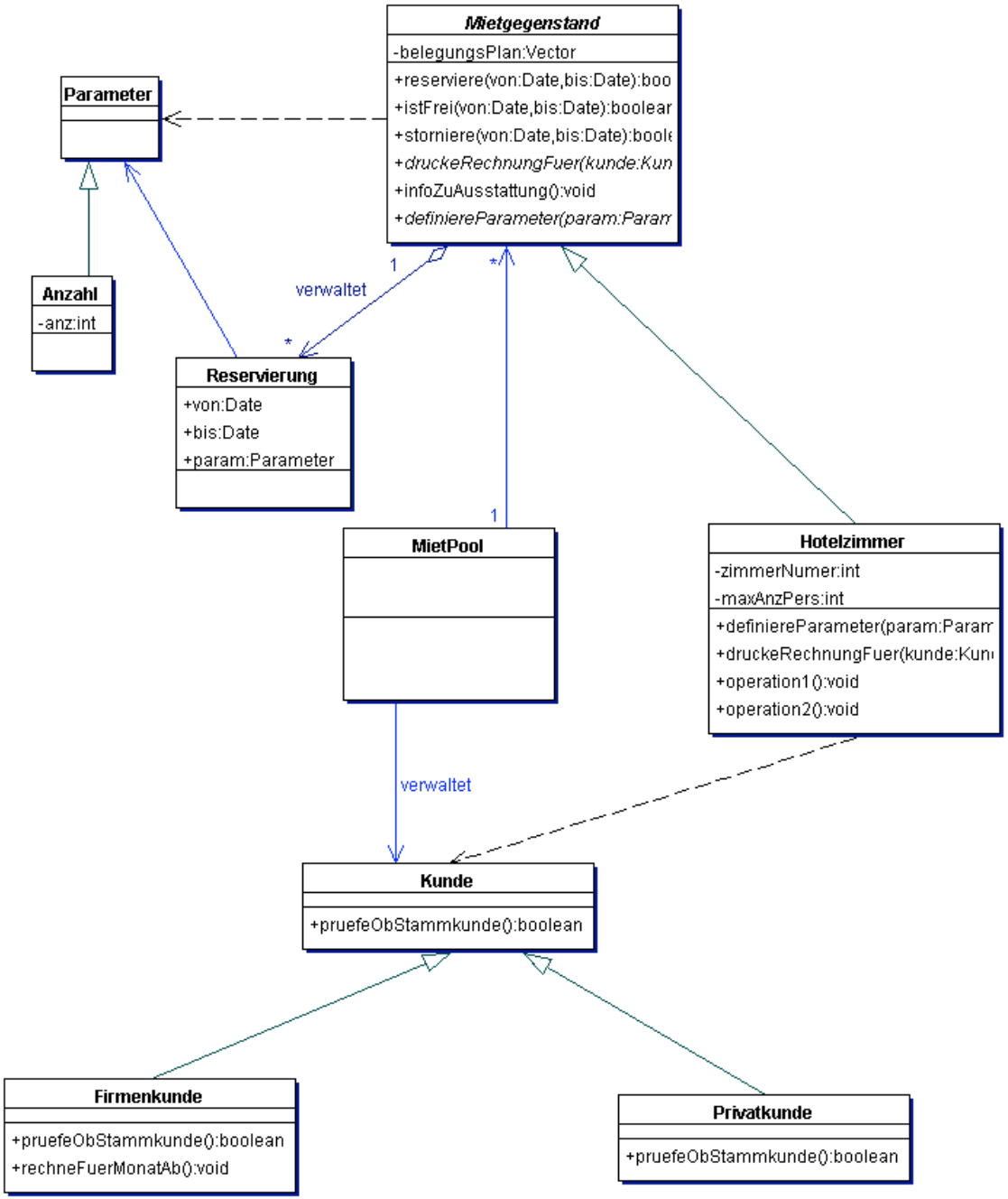
UML: Deployment Diagramm



Anhang C
UML-Beispieldiagramme

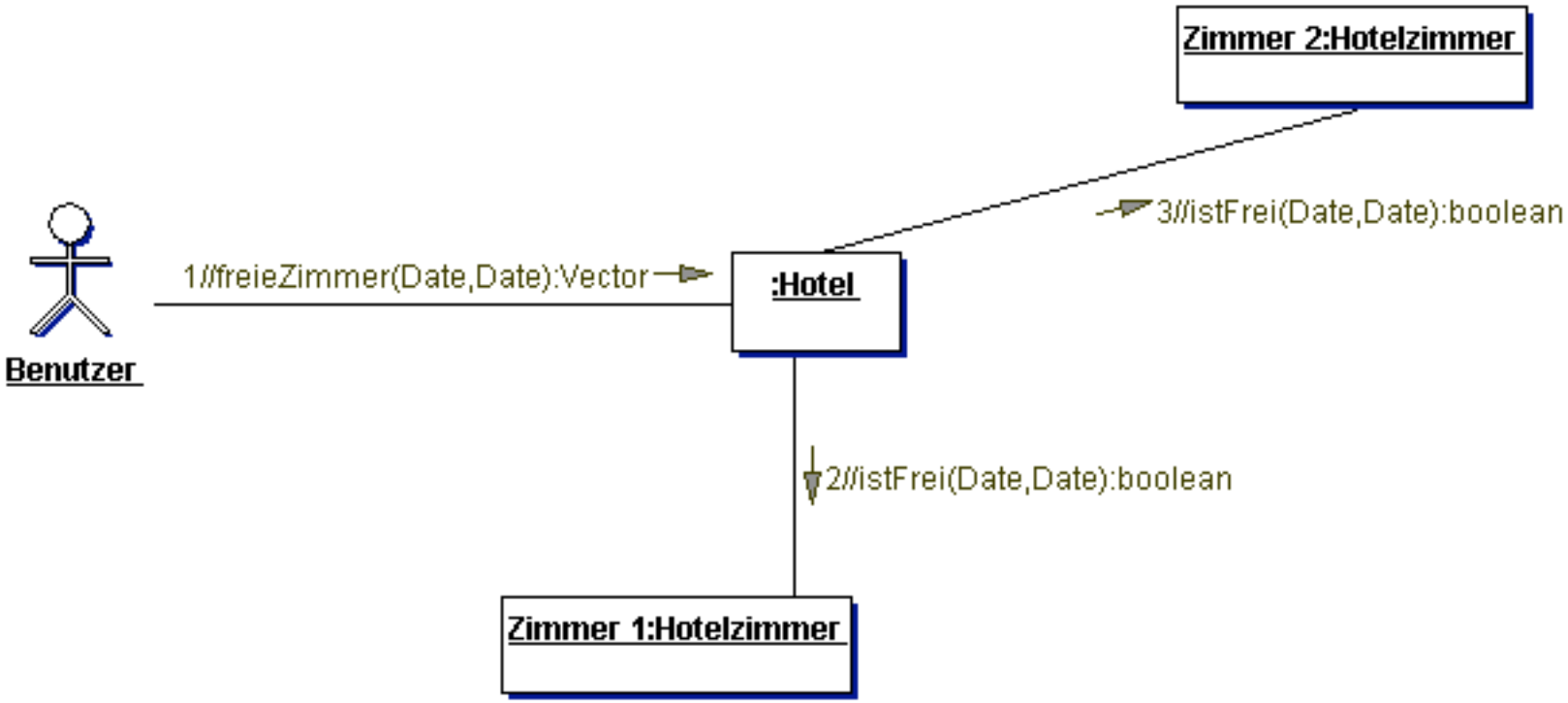
Getränkeautomat



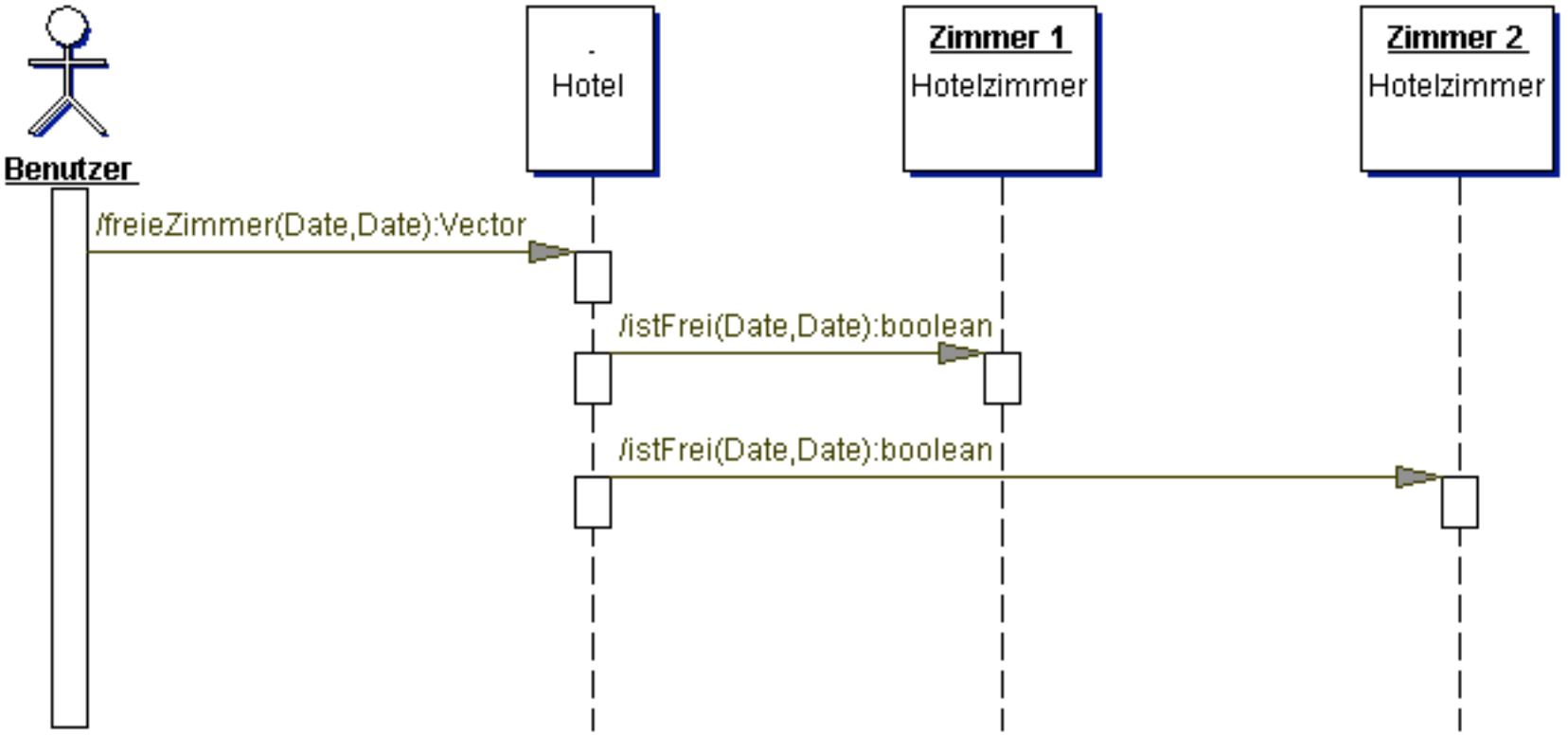


Hotel-reservierungs-system

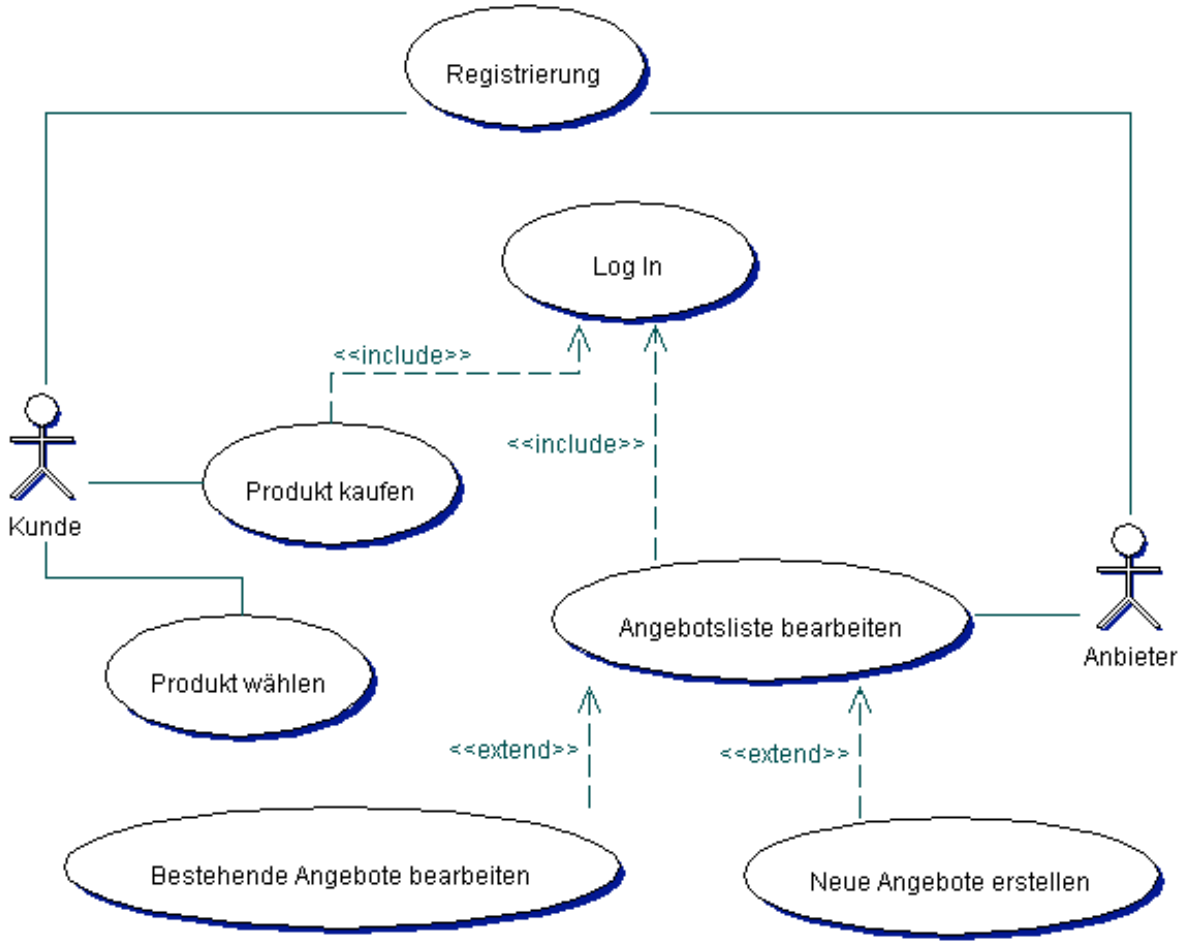
Hotel – Collaboration



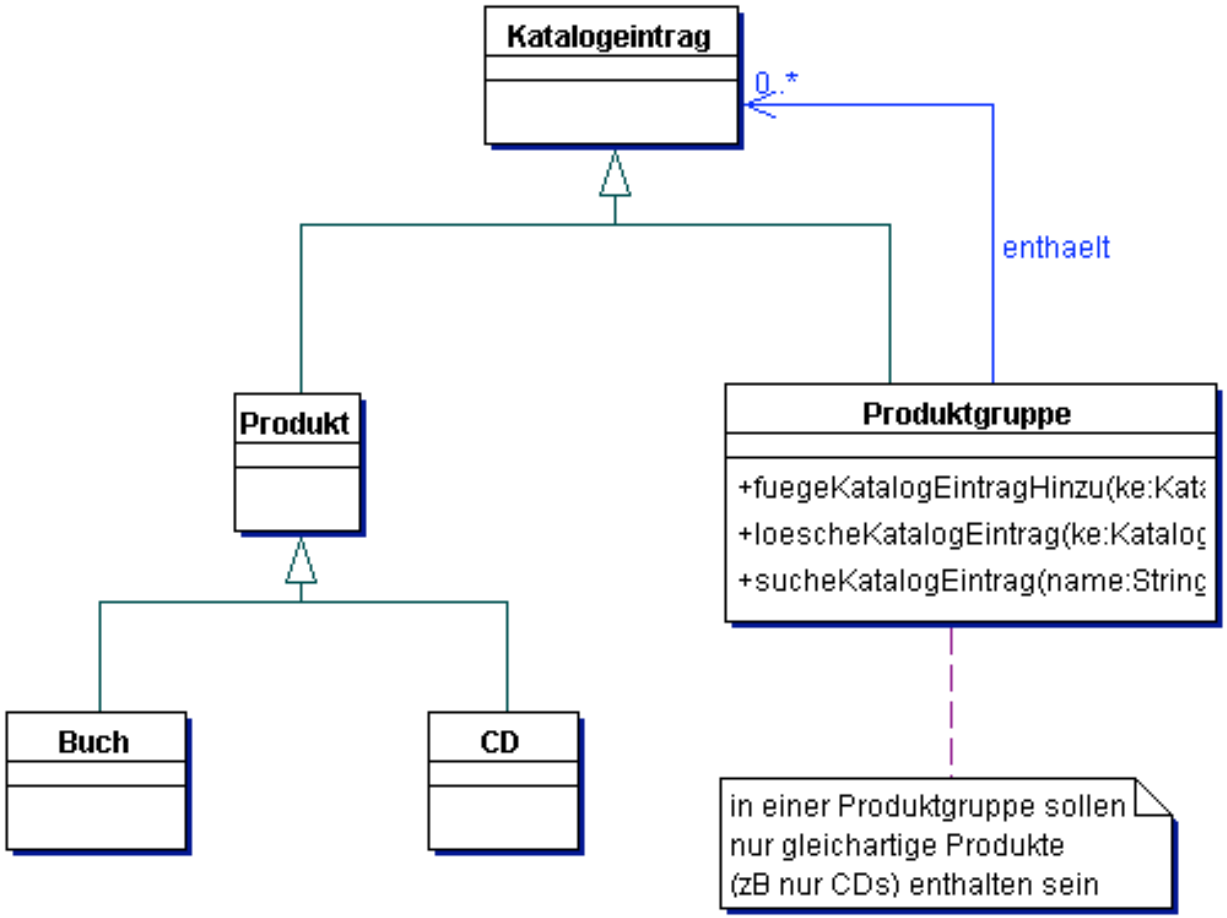
Hotel – Sequence



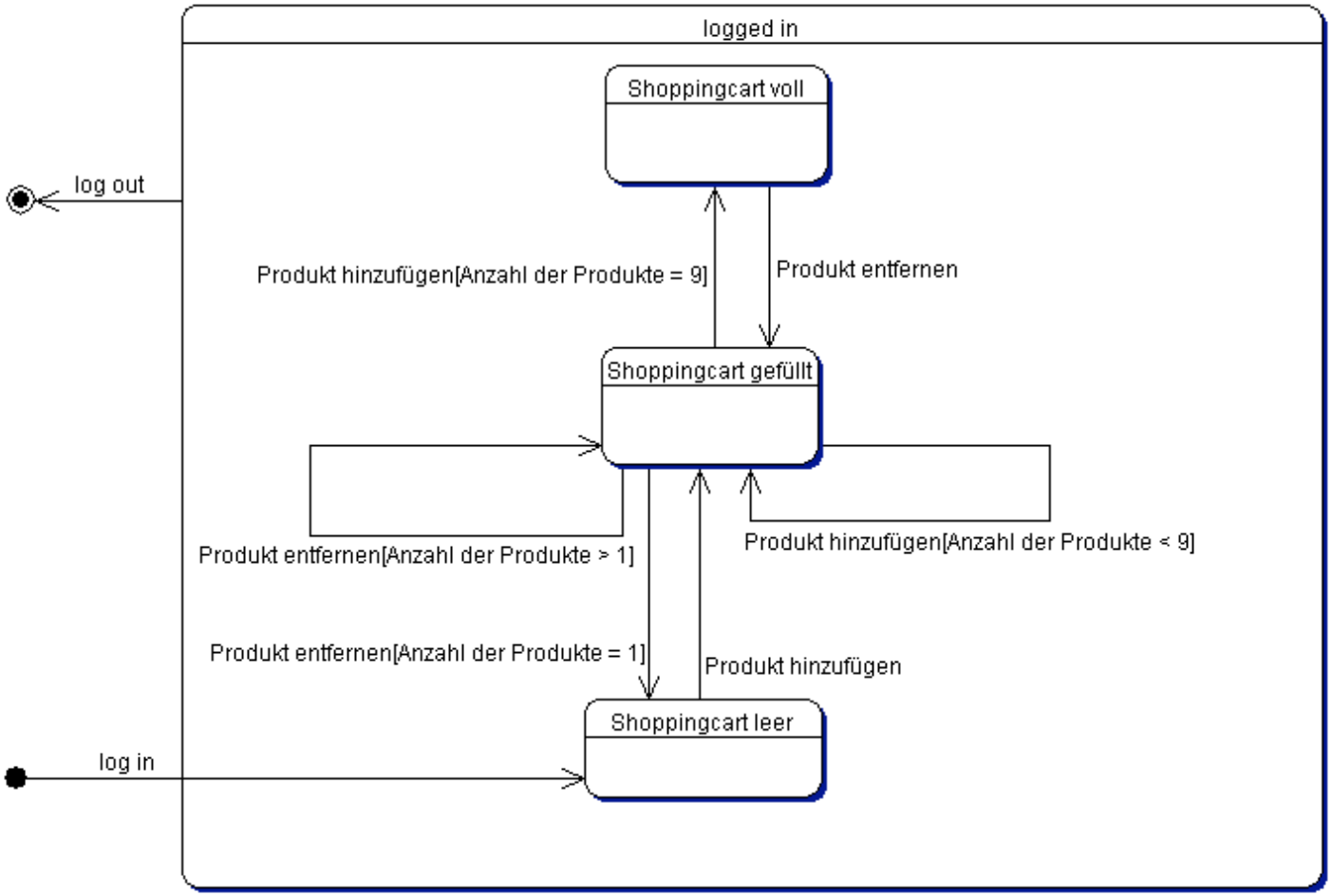
WebShop – Use Case



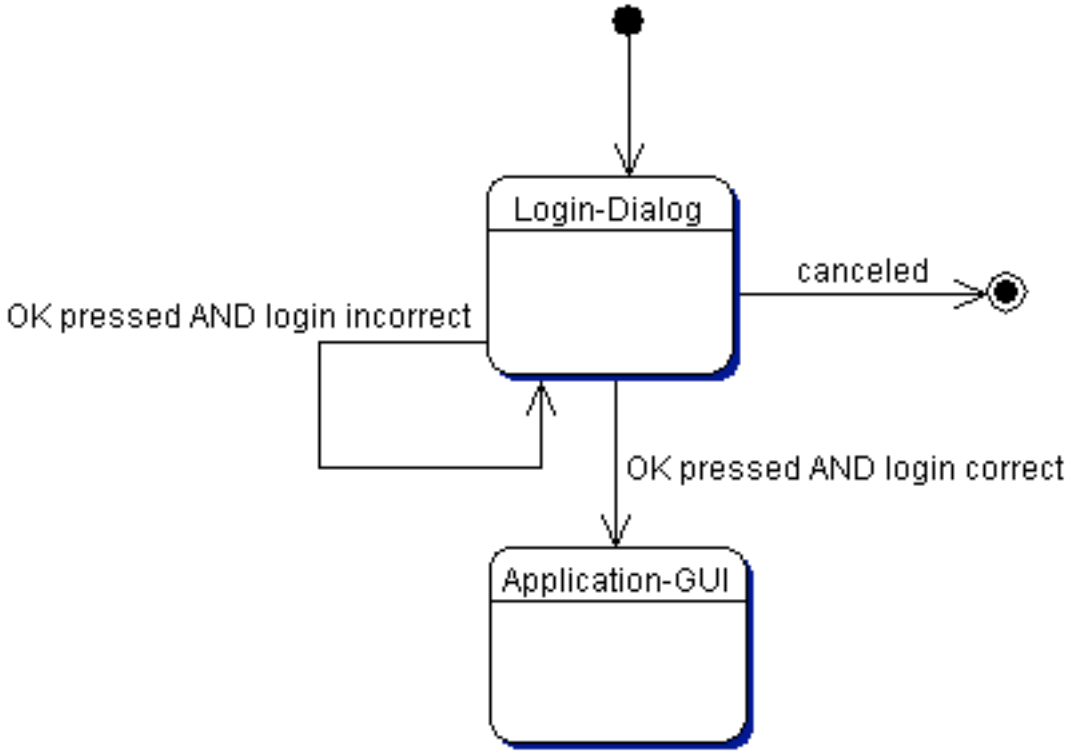
WebShop – Katalog



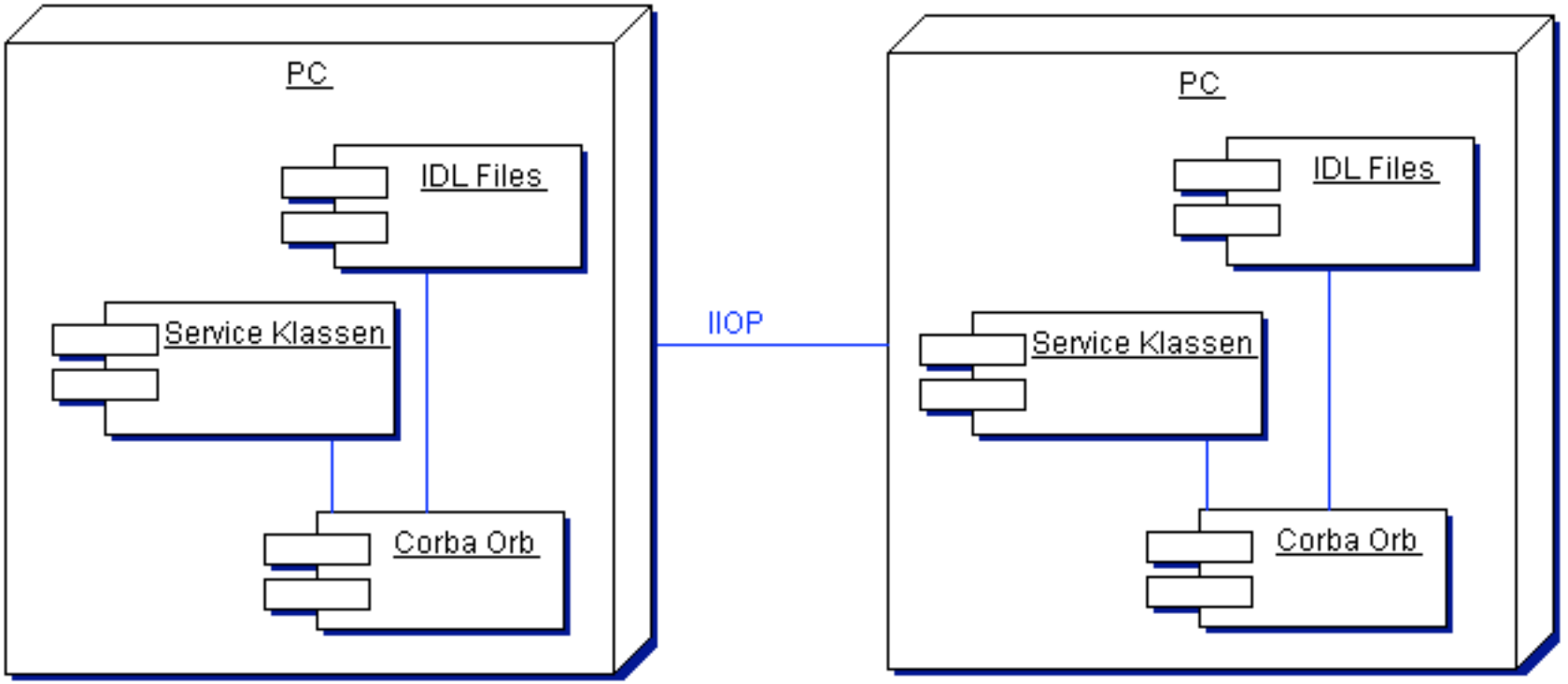
WebShop – ShoppingCart



Login-Dialog



CORBA – Deployment



3-Tier – Deployment