

Einführung in die Informatik II
UML

Prof. Bernd Brügge, Ph.D
Institut für Informatik
Technische Universität München

Sommersemester 2001

2. - 7. Mai 2001

Überblick über diesen Vorlesungsblock

❖ Sprachen, Notation:

- Eine genauere Einführung in die Modellierungssprache UML
 - Klassendiagramme
 - Anwendungsfalldiagramme

❖ Konzepte und Techniken:

- Identifizierung von Objekten und Assoziationen
- Modellierung von Objekten
- Modellierung der Interaktion zwischen Objekten
- Taxonomie der Benutzer von Klassen nach der Art der Nutzung in Analyse, Entwurf und Implementation

Zur Wiederholung

- ❖ Ein System kann als eine Menge von Untersystemen beschrieben werden, die rekursiv wieder jeweils eine Menge von Untersystemen enthalten können, bis wir auf Subsysteme stoßen, die Komponenten sind.
- ❖ Komponenten sind elementare Objekte mit Merkmalen (Zustand, Verhalten). Jedes Objekt gehört zu einer Menge von Objekten mit gleichen Merkmalen. Wir nennen diese Menge auch Klasse.
- ❖ **Definition Klasse:** Die Menge aller Objekte mit gleichen Merkmalen, d.h. mit gleichen Attributen und Operationen.
- ❖ **Definition Instanz:** Ein Objekt ist eine Instanz einer Klasse K , wenn es Element der Menge aller Objekte der Klasse K ist.
- ❖ Ein System hat Eigenschaften, die wir mit Diagrammen beschreiben können. Ein Klassendiagramm beschreibt statische Eigenschaften eines Systems.

Ziele des Vorlesungsblockes

- ❖ Wir definieren Klassendiagramme noch einmal genauer in UML-Notation und führen Anwendungsfall-Diagramme ein.
- ❖ Sie verstehen die unterschiedlichen Sichtweisen, die man von einer Klasse haben kann.
- ❖ Sie verstehen, warum man oft unterschiedliche Modelle während der Analyse, Entwurf und Implementation benutzt.
- ❖ Sie können ein Analysemodell für ein gegebenes Problem erstellen.
- ❖ Sie können aus einem Analysemodell ein Spezifikationsmodell mit Schnittstellen erstellen.

UML ist eine Sprache

- ❖ UML ist eine *Modellierungssprache*, keine *Software-Methode*:
 - Eine **Software-Methode** enthält eine (überwiegend grafische) Modellierungssprache und einen Software-Prozess.
 - ❖ Die **Modellierungssprache** beschreibt Modelle von Systemen.
 - ❖ Der **Software-Prozess** beschreibt die Entwicklungsaktivitäten, die Abhängigkeit dieser Aktivitäten voneinander und wie man am besten die Aktivitäten durchführt
 - Den Software-Prozess wird in der Softwaretechnik (Hauptstudium) detailliert behandelt.
 - In Info II benutzen wir einen sehr einfachen Software-Prozess, der aus
 - Analyse,
 - Entwurf,
 - detailliertem Entwurf und
 - Implementation
- besteht.

Die Geschichte von UML

- ❖ Die Modellierungssprache UML hat viele Vorläufer, unter anderem:
 - E/R-Notation, 1976: Chen, Modellierung von Datenbanken
 - Booch-Notation, 1985: Grady Booch, Rational.
 - OMT, 1991: James Rumbaugh, General Electric
 - OOSE, 1992: Ivar Jacobsen, Ericsson
- ❖ UML ist die Vereinigung der Konzepte dieser Vorläufer (und vieler anderer Sprachen), deshalb der Name *Unified* Modeling Language.
- ❖ UML 1.3 : Offizieller Standard der OMG (Object Management Group)

UML: Notation und Metamodell

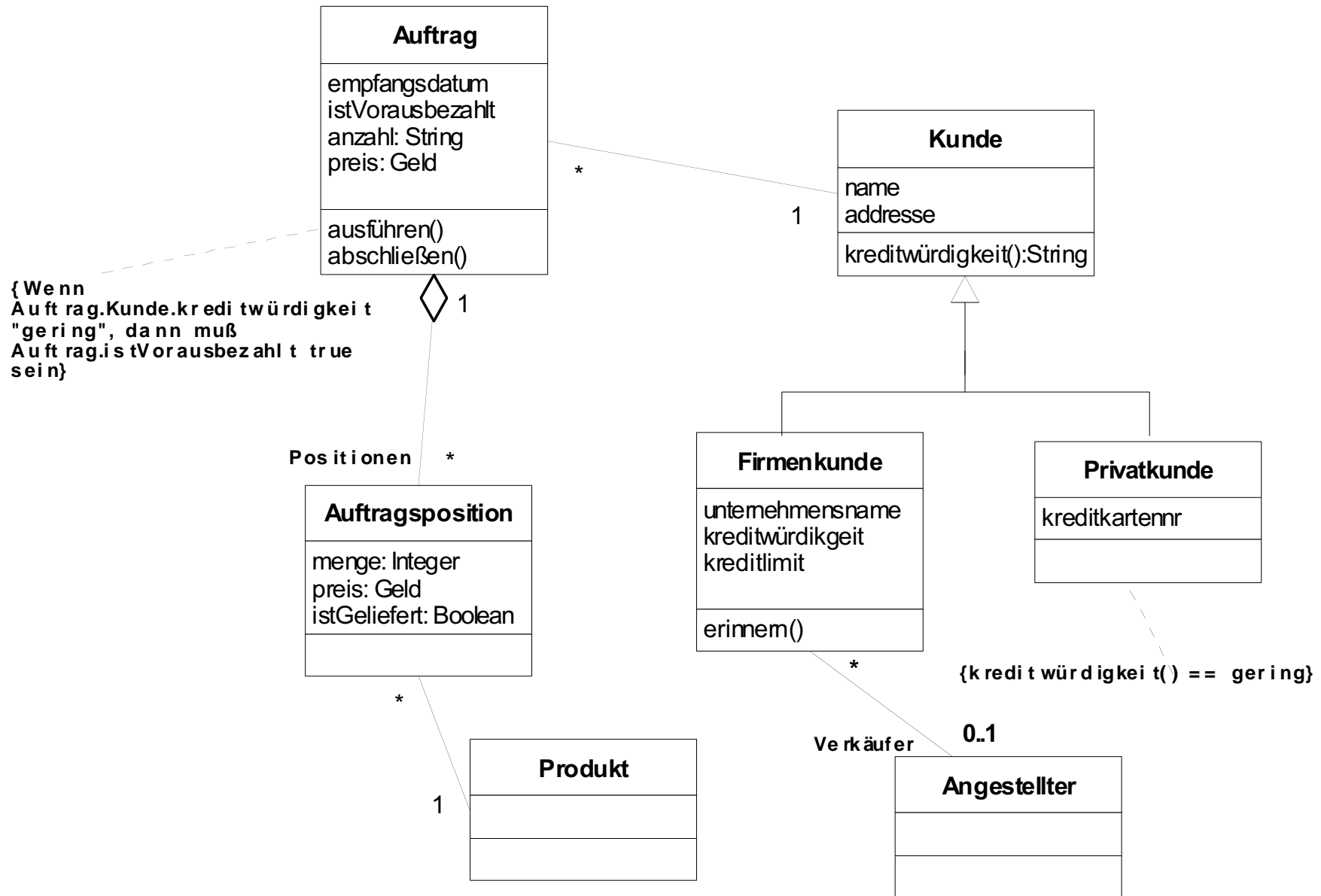
- ❖ UML ist eine Modellierungssprache für Systeme, insbesondere für Informatik-Systeme.
- ❖ UML besteht aus einer Notation und einem Metamodell.
- ❖ Die **Notation** beschreibt die **Syntax** der Modellierungssprache:
 - Die UML-Notation ist 2-dimensional und besteht aus grafischen Elementen (Java ist 1-dimensional und besteht aus textuellen Elementen)
 - Teile der UML-Notation haben wir (informell) bereits mehrfach zur Modellierung in Info I benutzt (Rechtecke, Dreiecke, Pfeile).
- ❖ Das **Metamodell** beschreibt die **Semantik** von UML:
 - Das UML Metamodell beschreibt "genauer", was wir unter Begriffen wie "Klasse", "Assoziation", "Multiplizität" verstehen.

Einschub: *Formale Methoden vs Modellierung*

- ❖ Die Modellierungssprache UML ist nicht exakt. Die Notation ist nicht formal begründet, d.h. es gibt keine formale Grammatik für die Notation. Also kann das Metamodell nicht mathematisch exakt sein.
- ❖ Die Idee einer exakten Entwurfssprache ist das zentrale Thema des Gebietes *Formale Methoden* im Hauptstudium, wo Entwürfe mit einer auf einem Kalkül basierenden Technik mathematisch beschrieben werden (auch **mathematische Spezifikation** genannt).
- ❖ Mathematische Spezifikationen sind exakt und unmißverständlich:
 - Man kann z.B. beweisen, dass ein Programm eine mathematische Spezifikation erfüllt (Wir werden dazu Entwurf durch Verträge und den *Zusicherungs-Kalkül* von Hoare benutzen).
 - Es gibt aber keine Möglichkeit zu beweisen, dass eine mathematische Spezifikation die Anforderungen an ein Informatik-System erfüllt.

⇒ mehr im Hauptstudium

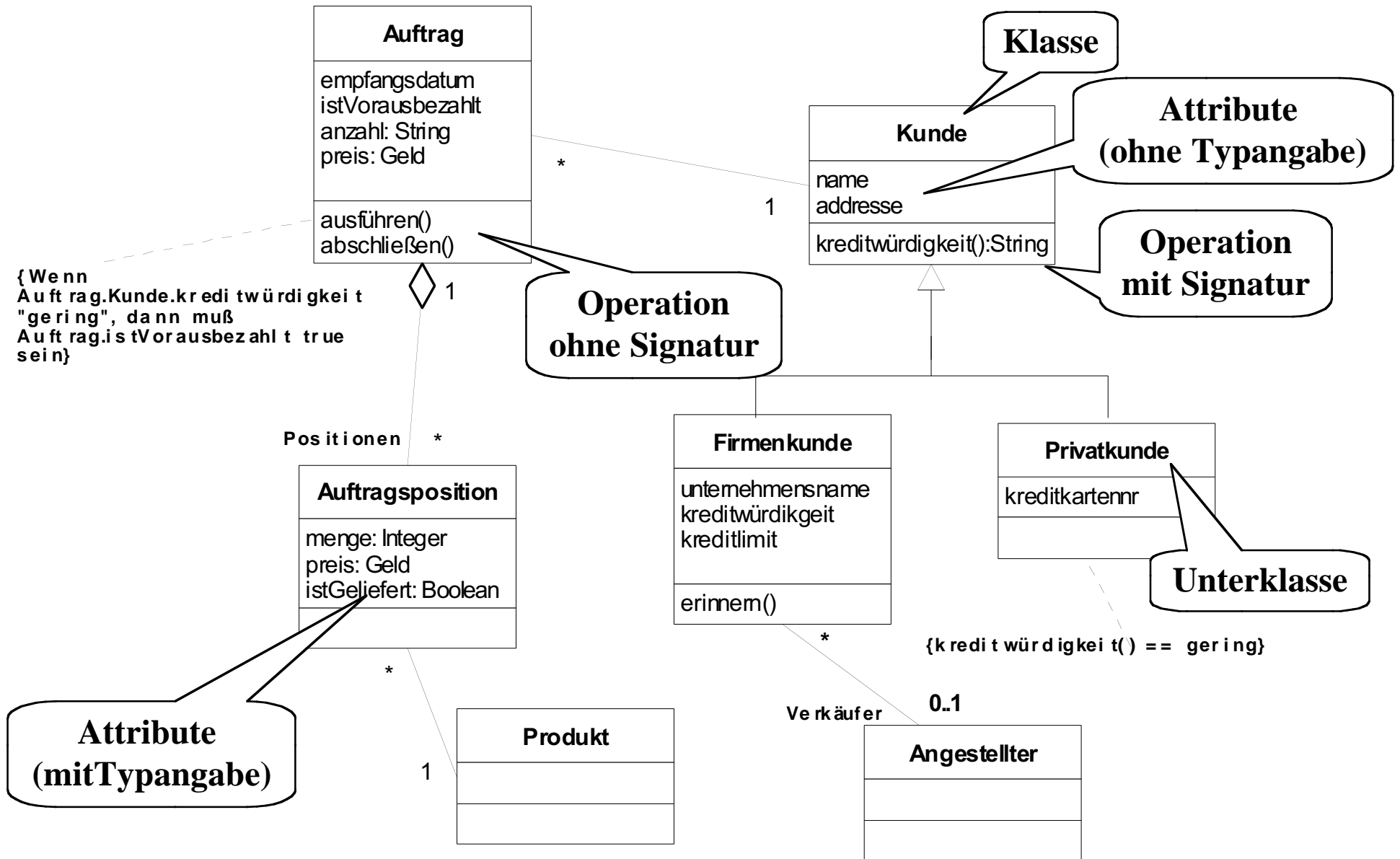
UML-Klassendiagramm: Notation anhand eines Beispiels



Klassendiagramme eignen sich sehr gut zur Kommunikation

- ❖ Klassendiagramme sind *die zentrale Beschreibungstechnik* bei objektorientierten Methoden.
 - Hauptvorteile:
 - Die wichtigsten Abstraktionen sind klar zu erkennen
 - Diagramme enthalten weniger Details als entsprechender Programm-Code
- ❖ Allerdings muss man sich immer fragen:
Wer benutzt Klassendiagramme?
 - Unterschiedliche Benutzer interpretieren Klassendiagramme unterschiedlich!
- ❖ Doch zunächst einmal ein bisschen UML-Syntax ...

UML: Klassen, Attribute und Operationen



UML-Syntax für Attribute

Syntax:

Sichtbarkeit Name : Typ = Voreingestellter Wert

Sichtbarkeit, Typ und Voreingestellter Wert sind optional.

Beispiele für gültige Attributdeklarationen:

preis: Geld = 50

+preis: Geld

-preis: Geld

preis

Beispiele für ungültige Deklarationen:

:Geld

= 50

Sichtbarkeit in UML

- ❖ In UML gibt es 3 **Sichtbarkeitsstufen** (visibilities) für Merkmale:

Öffentlich (public): Das Merkmal ist überall sichtbar.

Notation: +

Beispiel: **+preis: Geld**

Privat (private): Das Merkmal ist nur innerhalb der Klasse sichtbar.

Notation: -

Beispiel: **-preis: Geld**

Geschützt (protected): Merkmal ist in Klasse und Unterklasse sichtbar.

Notation: #

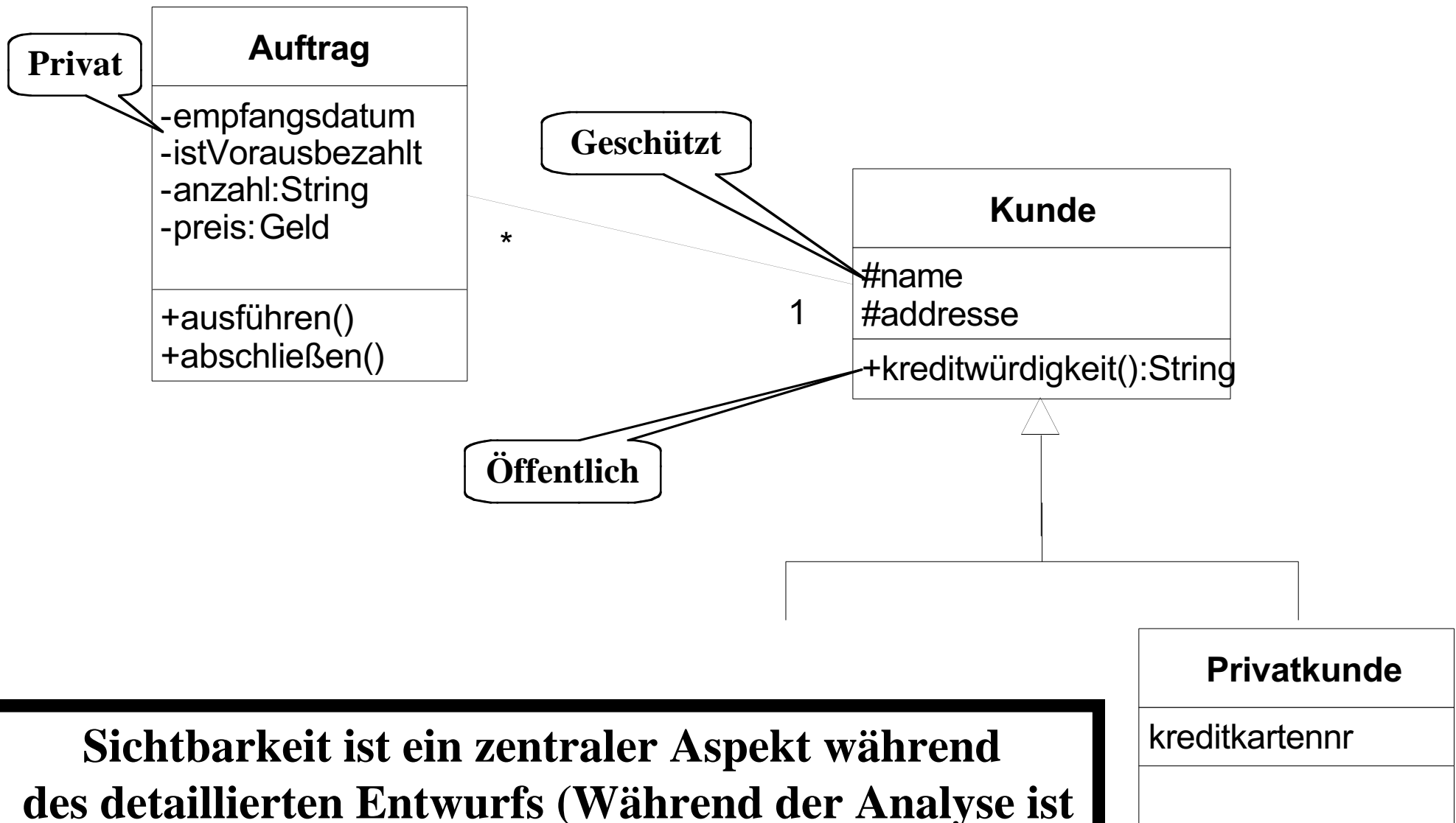
Beispiel: **#adresse: String**

- ❖ Die Angabe von Sichtbarkeitsstufen ist optional.
- ❖ Die Voreinstellung ist **öffentlich**.

Sichtbarkeit: UML vs. Java

- ❖ Java erlaubt das Deklarieren von Sichtbarkeitsstufen durch Angabe eines der Schlüsselworte **public**, **private**, **protected** vor einem Merkmal.
- ❖ Unterschiede zu UML:
 - Die Bedeutung von **protected** ist in Java nicht dieselbe wie in UML.
 - Java bietet eine zusätzliche, in UML nicht vorgesehene Sichtbarkeitsstufe:
Wird die Sichtbarkeitsstufe eines Merkmals nicht explizit angegeben, ist das Merkmal nur für Klassen nutzbar, die im gleichen "Paket" wie die Klasse, für die das Merkmal deklariert ist, enthalten sind.
 - dies kann zu zusätzlichen Komplikationen führen
 - ⇒ Vorlesungsblock über Verträge

Sichtbarkeitssymbole in UML-Klassendiagrammen



Sichtbarkeit ist ein zentraler Aspekt während des detaillierten Entwurfs (Während der Analyse ist Sichtbarkeit nicht so wichtig)

UML-Syntax für Operationen

Sichtbarkeit Name (Parameterliste) : Rückgabetyyp { Eigenschaften }

- *Sichtbarkeit* ist +, # oder –
- *Name* ist eine Zeichenkette
- *Parameterliste* ist eine durch Kommata getrennte Liste von Parametern.
- *Rückgabetyyp* ist der Typ des Resultates.
- *Eigenschaften* zeigen Werte an, die für die Operationen Anwendung finden.

Sichtbarkeit, Parameterliste, Rückgabetyyp und *Eigenschaften* sind optional.
Die runden Klammern müssen angegeben werden.

Beispiele für gültige Operationen:

```
+kontostandAm (datum: Datum) : Geld  
kontostandAm ()
```

Beispiele für ungültige Operationen:

```
kontostandAm  
konstandAm:Geld
```

UML-Syntax von Parametern

Richtung Name : Typ = Voreinstellungswert

- *Richtung* zeigt an, ob der Parameter ein Eingabeparameter (**in**), Ausgabeparameter (**out**) oder beides ist (**inout**).

❖ Beispiele:

+kontostandAm (in datum: Datum) : Geld

+kontostandAm (in datum: Datum; out Betrag: Geld)

+einzahlen (in datum: Datum; inout Konto: Geld)

Wieviel Detail bei Operationen und Attributen ist nötig?

- ❖ Wenn so viele Aspekte optional sind, wie detailliert soll man Attribute und Operationen in Klassendiagrammen beschreiben?
 - **Während der Analyse:**
Nur Namen von Attributen und Operationen
 - **Während des detaillierten Entwurfs:**
Jedes *öffentliche* Attribut hat einen Typ, jede *öffentliche* Operation hat eine Funktionalität
 - **Während der Implementation:**
Jedes Attribut hat einen Typ, jede Operation hat eine Funktionalität

Arten von UML-Operationen

- ❖ **Anfrage (Query):** Eine Operation, die einen Wert aus einem Objekt liefert, ohne den Zustand des Objektes zu verändern. Eine Anfrage hat also keine Seiteneffekte auf das Objekt. Beispiel einer Anfrage:

Betrag = kontostandAm ("30.4.2001")

Anfragen kann man mit der Einschränkung **{query}** markieren (siehe Folie 25).

- ❖ **Modifizierer (modifier):** Eine Operation, die den Zustand eines Objektes ändert.

– Beispiel eines Modifizierers: **setName ("Bruegge") ;**

- ❖ Unterschied zwischen Anfrage und Modifizierer:

– Anfragen können in beliebiger Reihenfolge ausgeführt werden.

– Bei Modifizierern ist die Reihenfolge wichtig.

- ❖ *Dogma:* In Info II sind Rückgabewerte nur bei Anfragen erlaubt. Modifizierer dürfen keine Rückgabetypen enthalten (Warum nicht?)

Operation vs. Methode

- ❖ In Modellierungssprachen werden die Begriffe **Operation** und **Methode** manchmal synonym benutzt. Die beiden Begriffe haben allerdings unterschiedliche Bedeutung beim **Polymorphismus**, der es erlaubt, *eine* Operation an *unterschiedliche* Methoden zu binden (**dynamische Bindung** erlaubt dies zur Laufzeit).
- ❖ Polymorphismus und dynamische Bindung sind zentrale Konzepte des objekt-orientierten Programmierstils.
- ❖ Deshalb *unterscheiden wir bei der Modellierung* zwischen Operation und Methode:
 - **Definition Operation:** Deklaration oder Aufruf eines Algorithmus.
Synonyme: Methodendeklaration, Methodenaufruf
 - **Definition Methode:** Implementation (oder Rumpf) eines Algorithmus.
Synonym: Methodenrumpf

Operationen in Programmiersprachen

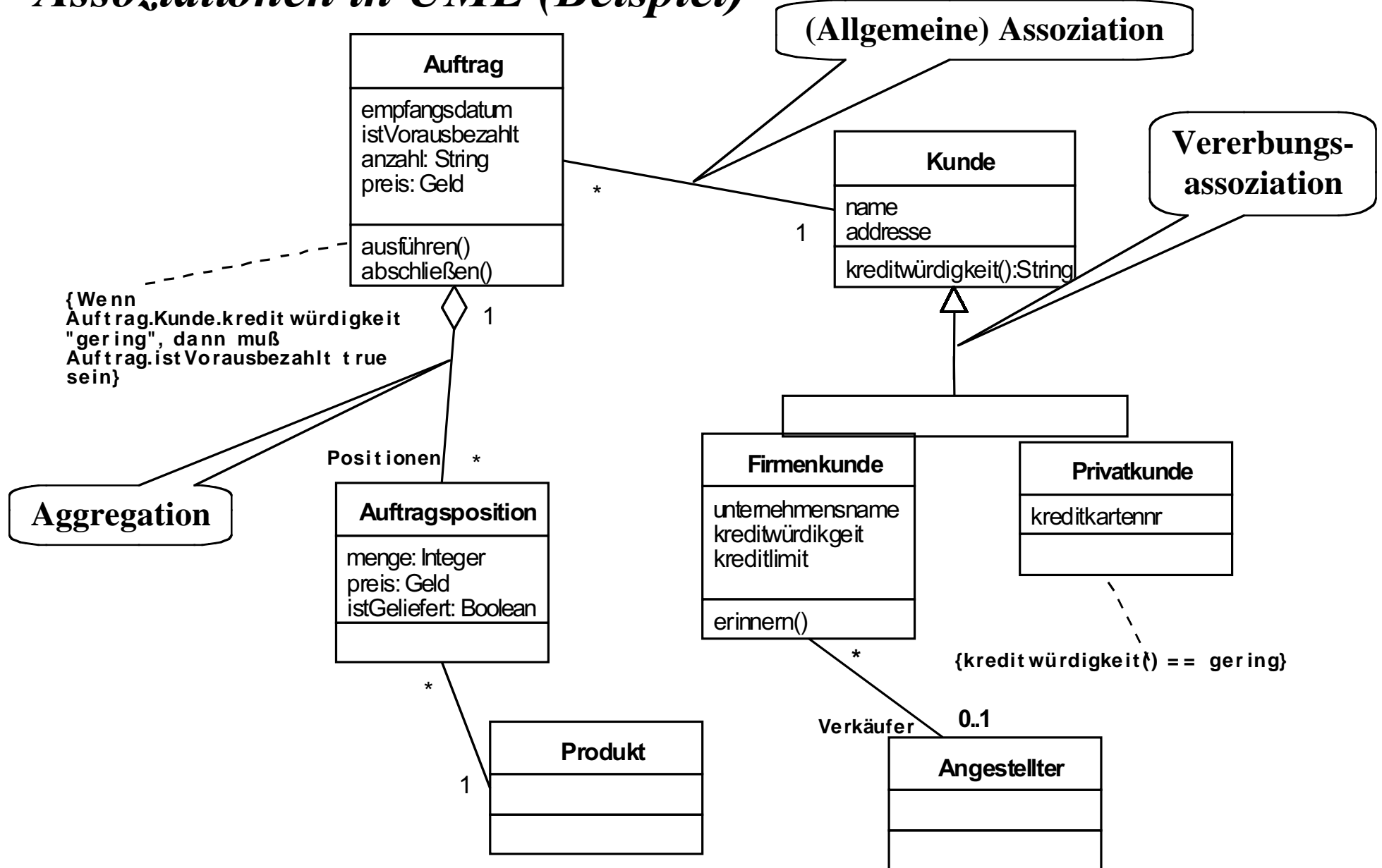
- ❖ Programmiersprachen haben ihre eigenen Namenskonventionen:
 - In Java heißen Operationen *Methoden*.
 - In C++ werden Operationen *Funktionen* der Klasse genannt.

- ❖ In **Programmiersprachen** benutzen wir deshalb nicht den Begriff Operation, sondern unterscheiden zwischen *Methodendeklaration*, *Methodenrumpf* und *Methodenaufruf*.

Assoziationen in UML

- ❖ **Definition:** Eine Assoziation verbindet zwei Klassen (die sogenannten *Zielklassen*) und beschreibt so eine Beziehung zwischen diesen Zielklassen.
- ❖ Die Anknüpfungspunkte an den Zielklassen werden als *Assoziationsenden* bezeichnet.
- ❖ Es gibt 3 Arten von Assoziationen:
 - **allgemeine Assoziation**
 - **Aggregation** (Enthaltenseins-Beziehung)
 - **Vererbung**
- ❖ Allgemeine Assoziationen und Aggregationen können (durch Angabe einer Pfeilspitze an einem Assoziationsende) *gerichtet* sein.

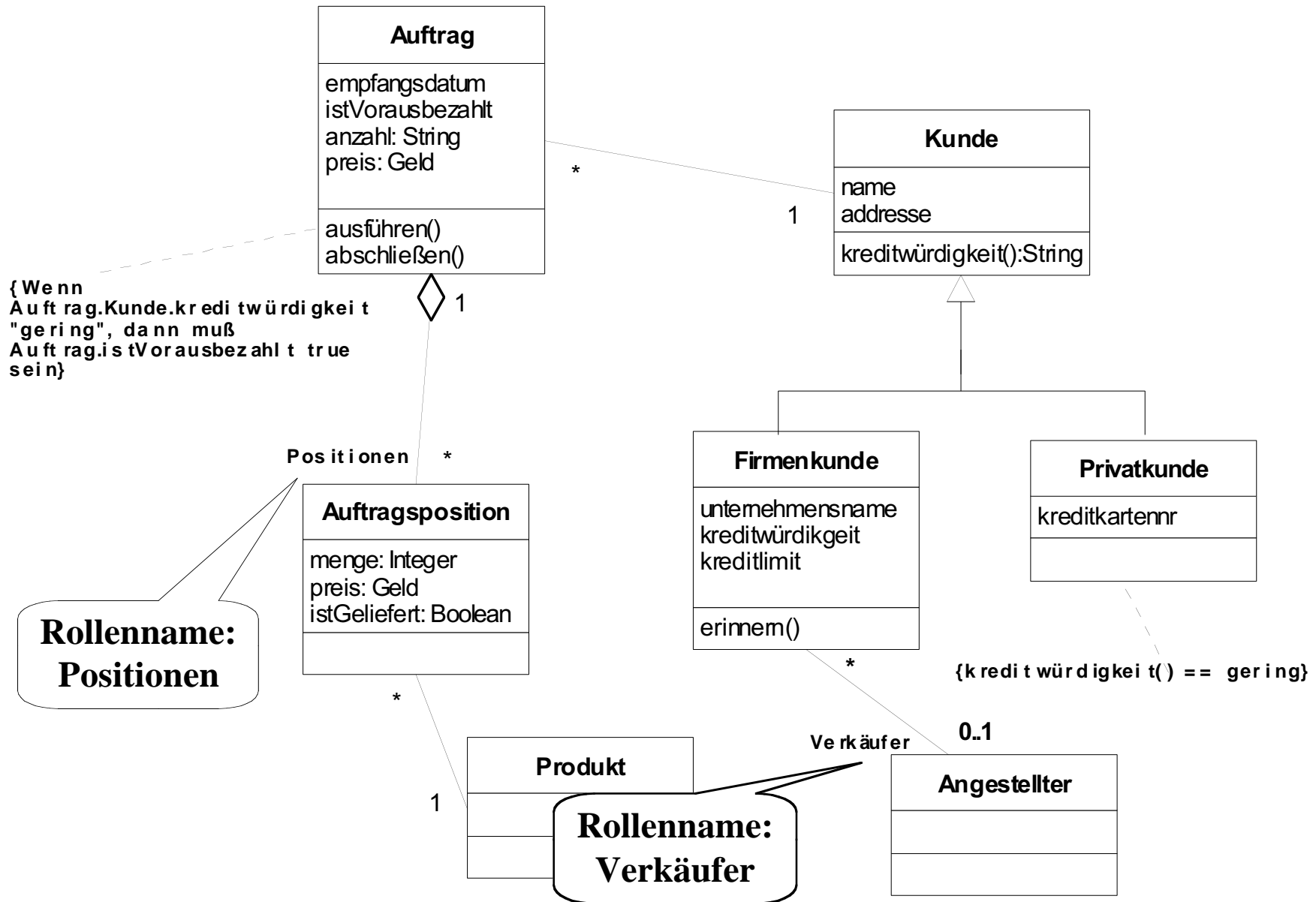
Assoziationen in UML (Beispiel)



Rollen

- ❖ **Definition:** Ein Assoziationsende kann explizit mit einer Beschriftung versehen werden. Diese Beschriftung heißt Rolle.
 - Wenn eine Assoziation keine Beschriftung hat, nennt man das Assoziationsende nach der Zielklasse
 - Beispiel (siehe nächste Folie): Die Rolle von Auftrag zu Kunde würde man Kunde nennen.

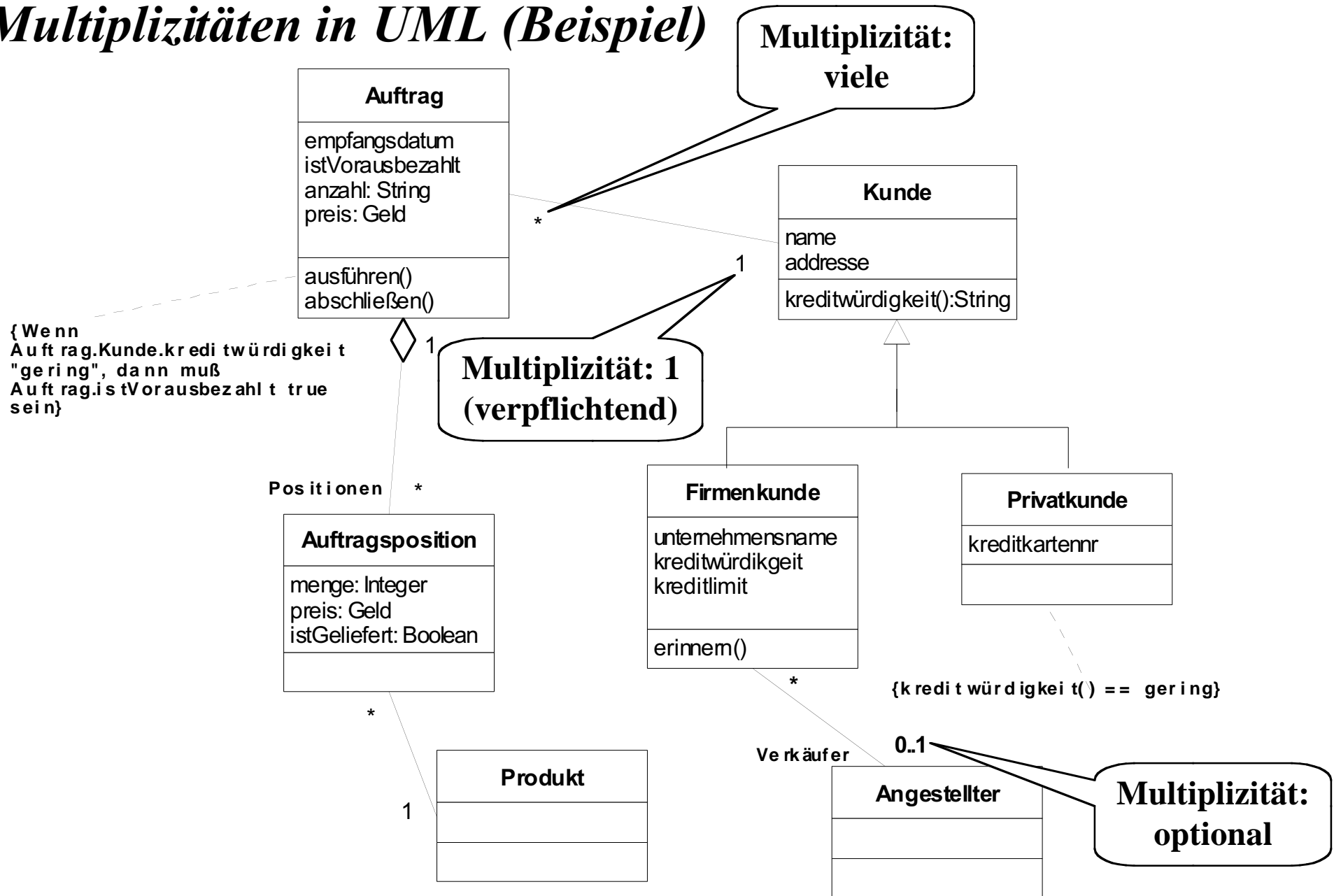
Rollen in UML (Beispiel)



Multiplizität

- ❖ **Definition:** Die Zahl an einem Assoziationsende, die besagt, wieviele Objekte an der Beziehung beteiligt sein können, heißt Multiplizität.
- ❖ Beispiele für Multiplizitäten:
 - * ("0..∞")
Beispiel: Ein Kunde kann keinen oder viele Aufträge haben
 - 1 ("genau 1")
Beispiel: Ein Auftrag ist genau einem Kunden zugeordnet
 - 0..1 ("Einer oder keiner", "optional")
Beispiel: Einige Firmenkunden werden durch einen speziellen Verkäufer betreut, andere Firmenkunden nicht
- ❖ Voreinstellung: Wenn ein Assoziationsende keine explizite Multiplizität hat, dann hat sie die Multiplizität 1.

Multiplizitäten in UML (Beispiel)



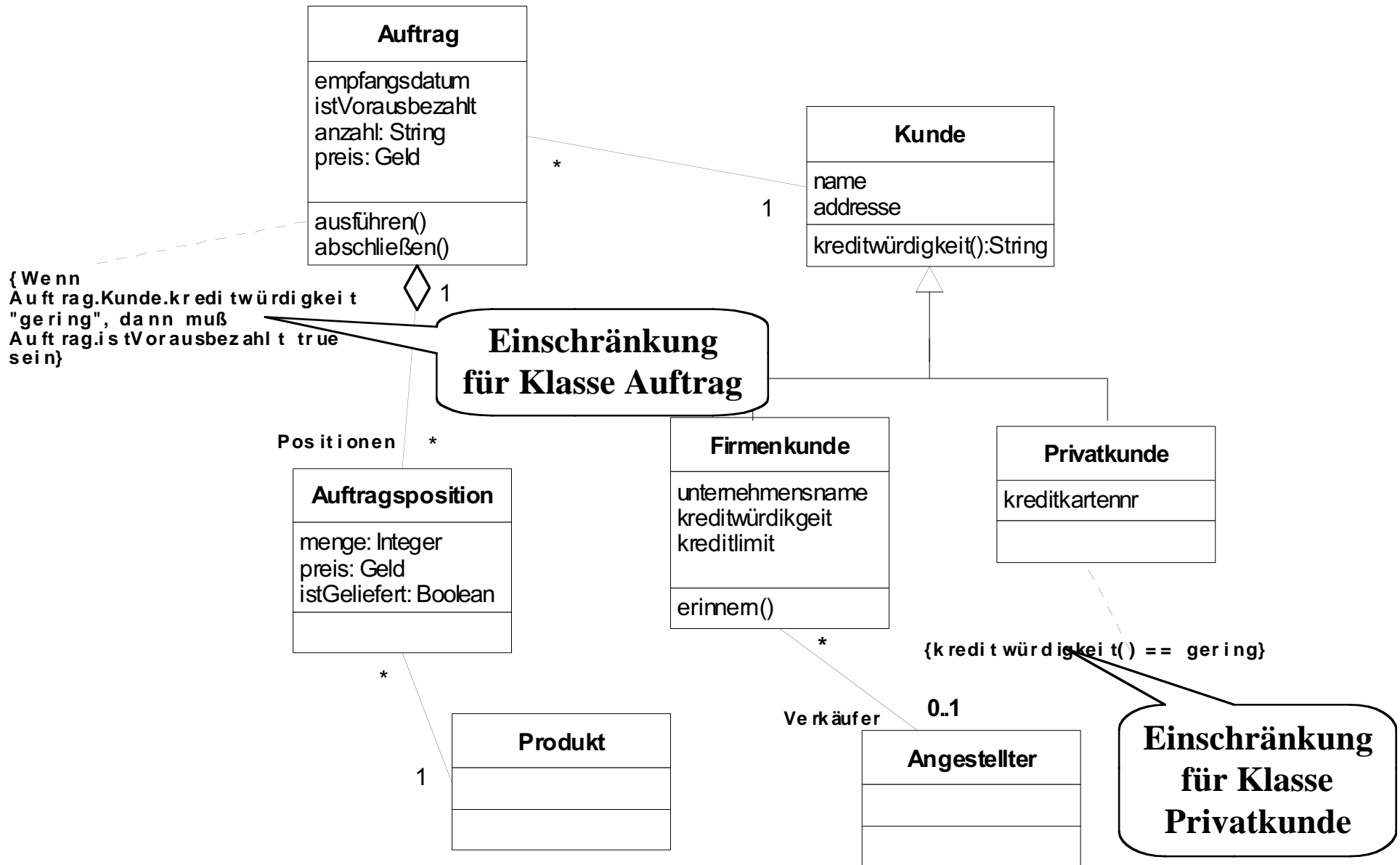
Die Vererbungsassoziation hat keine Multiplizität

- ❖ Die Angabe von Multiplizitäten ist nur bei allgemeinen Assoziationen und Aggregationen nötig.
- ❖ Bei Vererbung ist sie nicht sinnvoll. Warum?

Modellierung von Einschränkungen mit UML

- ❖ Ein wesentlicher Aspekt von Klassendiagrammen sind Beschreibungen von *Einschränkungen*.
- ❖ Mit den Konzepten "Assoziation", "Attribut" und "Generalisierung" kann man bereits viele Einschränkungen ausdrücken, z.B.:
 - Ein Auftrag kann nur von einem einzelnen Kunden kommen.
 - Ein Auftrag besteht aus separaten Auftragspositionen, d.h. man kann nicht 2 Auftragspositionen zusammen bestellen.
- ❖ Die folgende Einschränkung kann man allerdings so nicht ausdrücken:
 - Firmenkunden haben ein Kreditlimit, Privatkunden nicht.
- ❖ UML enthält die Möglichkeit zur Beschreibung von beliebigen Einschränkungen.
 - Die Einschränkung muss in geschweifte Klammern { } gesetzt werden.
- ❖ Wir unterscheiden informelle und formale Einschränkungen.

Informelle Einschränkungen in UML: Beliebiger Text



Formale Einschränkungen in UML: OCL

- ❖ UML stellt eine Sprache zur Verfügung, mit der Einschränkungen formal beschrieben werden können.
- ❖ Die Sprache heißt OCL (Object Constraint Language).
- ❖ OCL besprechen wir im Vorlesungsblock "Entwurf durch Verträge".

Wer benutzt Klassendiagramme?

- ❖ In Info II benutzen wir Klassendiagramme für 2 Beschreibungszwecke:
 - Die Beschreibung der statischen Eigenschaften eines Informatik-Systems. (Hauptzweck)
 - Beschreibung von Konzepten (Didaktische Gründe)
- ❖ Informatik-Systeme sind für zwei Gruppen von Personen interessant: **Kunde** und **Entwickler**.
 - Der **Kunde (Benutzer)** eines Informatik-Systems ist an Klassendiagrammen nicht so sehr interessiert, sondern an der Funktionalität des Systems, die in UML mit Anwendungsfällen modelliert werden.
 - Der **Entwickler** eines Informatik-Systems erzeugt und benutzt Klassendiagramme während der Entwicklungsphasen des Informatik-Systems, d.h. während der Analyse, dem System-Entwurf, dem detaillierten Entwurf und der Implementation.

Benutzer von Klassendiagrammen

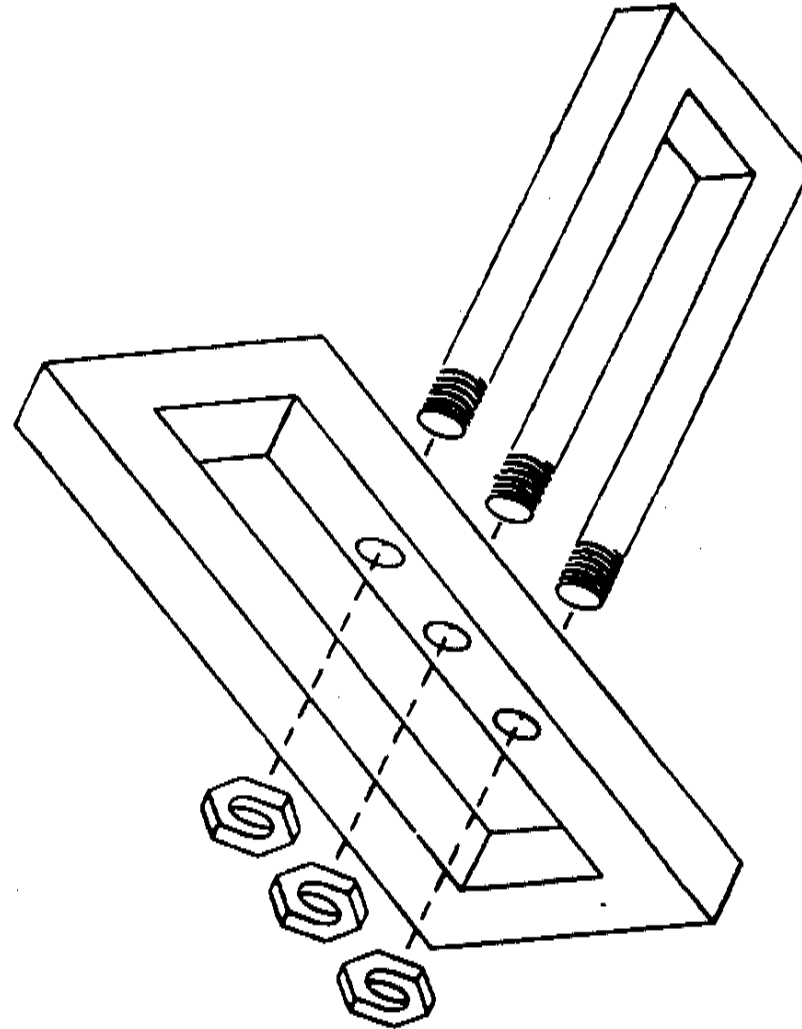
- ❖ Die **Entwickler** eines Informatik-Systems bezeichnen wir nach der Entwicklungsphase des Informatik-Systems als
 - **Analytiker**
 - **System-Entwerfer**
 - **Klassen-Entwerfer**
 - **Implementierer**
- ❖ Jeder dieser Entwickler hat unterschiedliche Sichtweisen von Modellen.
- ❖ Bevor wir diese Typen von Entwicklern beschreiben, machen wir noch eine Unterscheidung bezüglich der Art von Klassen, die in Klassendiagrammen auftauchen:
 - **Anwendungsklassen**
 - **Lösungsklassen**

Einschub: Anwendungsklassen vs Lösungsklassen

- ❖ **Definition Anwendungsdomäne** (application domain): Der Bereich, aus dem das Problem kommt (Finanzwelt, Meteorologie, Unfallverhütung, Architektur, ...).
- ❖ **Definition Anwendungsklasse:** Eine Abstraktion aus der Anwendungsdomäne. Bei Modellierungen in der Geschäftswelt nennt man diese Klassen auch oft Geschäftsobjekte (business objects).
 - Beispiele: Student, Studentenverzeichnis, Einkaufskorb
- ❖ **Definition Lösungsdomäne** (solution domain): Der Bereich, der Lösungen für Teilprobleme liefert (Telekommunikation, Datenbanken, Compilerbau, Betriebssysteme, Entwurfsmuster, ...).
- ❖ **Definition Lösungsklasse:** Eine Abstraktion, die aus technischen Gründen eingeführt wird, da sie bei der Lösung des Problems hilft.
 - Beispiele: sortierter Baum, verkettete Liste

Warum diese Unterscheidung?

Der Kunde kommt mit diesem Problem. Was machen Sie?



Analytiker

❖ Analytiker (analyst)

- Die Klassen, die den Analytiker interessieren, sind Applikationsklassen.
- Die Assoziationen zwischen Klassen bezeichnen Beziehungen zwischen Abstraktionen in der Anwendungsdomäne.
- Den Analytiker interessiert auch, ob Vererbungshierarchien im Modell die Taxonomie in der Anwendungsdomäne richtig widerspiegeln.
 - **Definition Taxonomie:** Eine Hierarchie von Begriffen.
- Lösungsklassen sowie die genaue Signatur von Operationen interessieren den Analytiker nicht.

Welche Anwendungsklassen sehen Sie hier?



Entwerfer

- ❖ Der Entwerfer konzentriert sich auf die Lösung des Problems, also auf die Lösungsdomäne.
- ❖ Entwurf besteht aus vielen Aufgaben (Subsystemauswahl, Auswahl der Datenbank, Auswahl der Zielplattform, usw.). In Info II konzentrieren wir uns auf die Spezifikation von Schnittstellen.
 - Ein Entwerfer, der sich nur auf Schnittstellen konzentriert, wird auch Spezifizierer genannt.
- ❖ Aufgaben des Spezifizierers:
 - Erstellt die Schnittstelle einer Klasse (Klassen-Entwurf) oder eines Subsystems (System-Entwurf).
 - Definiert die Schnittstelle so, dass sie von möglichst vielen anderen Klassen in demselben Informatik-System benutzbar ist.
 - Wenn möglich: Definiert die Schnittstelle so, dass sie in anderen Informatik-Systemen wiederverwendbar (reusable) ist.

Drei Arten von Implementierern

❖ **Klassen-Implementierer (class implementor):**

- Implementiert die Klasse. Der Implementierer wählt geeignete Datenstrukturen und Algorithmen, und realisiert die Schnittstelle der Klasse in einer Programmiersprache.

❖ **Klassen-Erweiterer (class extender):**

- Erweitert die Klasse durch eine Subklasse, die für ein neues Problem oder eine neue Anwendungsdomäne benötigt wird.

❖ **Klassen-Benutzer (client):**

- Verwendet eine existierende Klasse für eigene Programme (z.B. eine Klasse aus einer Klassenbibliothek, oder eine Klasse aus einem anderen Subsystem).
- Den Klassen-Benutzer interessiert die Signatur der Klasse und ihre Semantik, so dass er die Methoden aufrufen kann. Die Implementation der Klasse selbst interessiert ihn nicht.

Warum betonen wir diese unterschiedlichen Benutzer?

- ❖ Viele Modelle machen keinen Unterschied zwischen Applikationsklasse ("Verzeichnis") und Lösungsklasse ("Reihung", "Baum").
 - **Der Grund:** Modellierungssprachen erlauben es, beide Arten in einem Modell zu verwenden.
 - **Info II Dogma:** Lösungsklassen sind im Analysemodell nicht erlaubt.
- ❖ Viele Softwaresysteme machen keinen Unterschied zwischen der Spezifikation und Implementation einer Klasse.
 - **Der Grund:** Objekt-orientierte Programmiersprachen erlauben die Vermischung von Schnittstellenspezifikation und Implementation bei der Deklaration einer Klasse.
 - **Info II Dogma:** Implementationen sind im Spezifikationsmodell nicht erlaubt.
- ❖ *Heuristik:* Der Schlüssel für die Erstellung von Softwaresystemen mit hoher Qualität ist eine genaue Unterscheidung zwischen
 - Anwendungsklassen und Lösungsklassen
 - Schnittstellenspezifikation und Implementation

Erstellung von Informatik-Systemen: Beteiligte Personen

- ❖ Informatik-Systeme haben viele unterschiedliche *Interessenten*:

Endanwender

Kunde

Entwickler

- **Analytiker**

- **Entwerfer**

 - System-Entwerfer

 - Klassen-Entwerfer

- **Implementierer**

 - Benutzer von Klassen

 - Implementierer von Klassen

 - Erweiterer von Klassen.

- ❖ Wir sehen diese Interessenten als **Rollen**, die je nach Komplexität des Informatik-Systems auf eine oder mehrere Personen abgebildet werden.

Beispiele für die Abbildung von Rollen auf Personen

- ❖ **Studienverwaltungssystem in Info II:**
Kunde ist der Übungsleiter; alle Entwicklerrollen werden von derselben Person ausgeübt, nämlich von Ihnen.
 - Frage: Wer ist der Benutzer?
- ❖ **Campus Management System von SAP:**
Jede Rolle wird von einem Team ausgeübt, das aus mehreren Personen besteht.
 - Frage: Wer ist der Benutzer?
- ❖ Das Ziel von Info II ist, dass Sie die wichtigsten Rollen bei der Entwicklung eines Software-Systems kennen und einige davon sehr gut ausüben können.

Klassendiagramme sind immer Teile von Modellen

- ❖ Klassendiagramme werden in verschiedenen Modellen verwendet:
 - **Analysemodell**
 - **Spezifikationsmodell**
 - **Implementationsmodell**
- ❖ Je nach unserem Interesse betrachten wir diese Modelle sehr unterschiedlich, und oft interessiert uns in einer bestimmten Rolle nicht alles im Modell:
 - ⇒ 3 Arten von Schnittstellen im Spezifikationsmodell
- ❖ Je nach Rolle und Modell gibt es unterschiedliche Interpretationen für verschiedene UML-Konstrukte:
 - Unterschiedliche Interpretation von **Assoziationen**
 - Unterschiedliche Interpretation von **Attributen**
 - Unterschiedliche Interpretation von **Vererbung**
- ❖ Wir werden jetzt auf diese unterschiedlichen Interpretationen eingehen.

Analysemodell

- ❖ Das **Analysemodell** wird während der Analysephase erstellt.
 - Hauptinteressenten: Analytiker, Kunde, Benutzer.
 - Das Klassendiagramm enthält nur die Applikationsklassen.
- ❖ Das Analysemodell ist die Grundlage der Kommunikation zwischen den Analytikern, den Experten der Anwendungsdomäne und den Endbenutzern des Systems.

Spezifikationsmodell

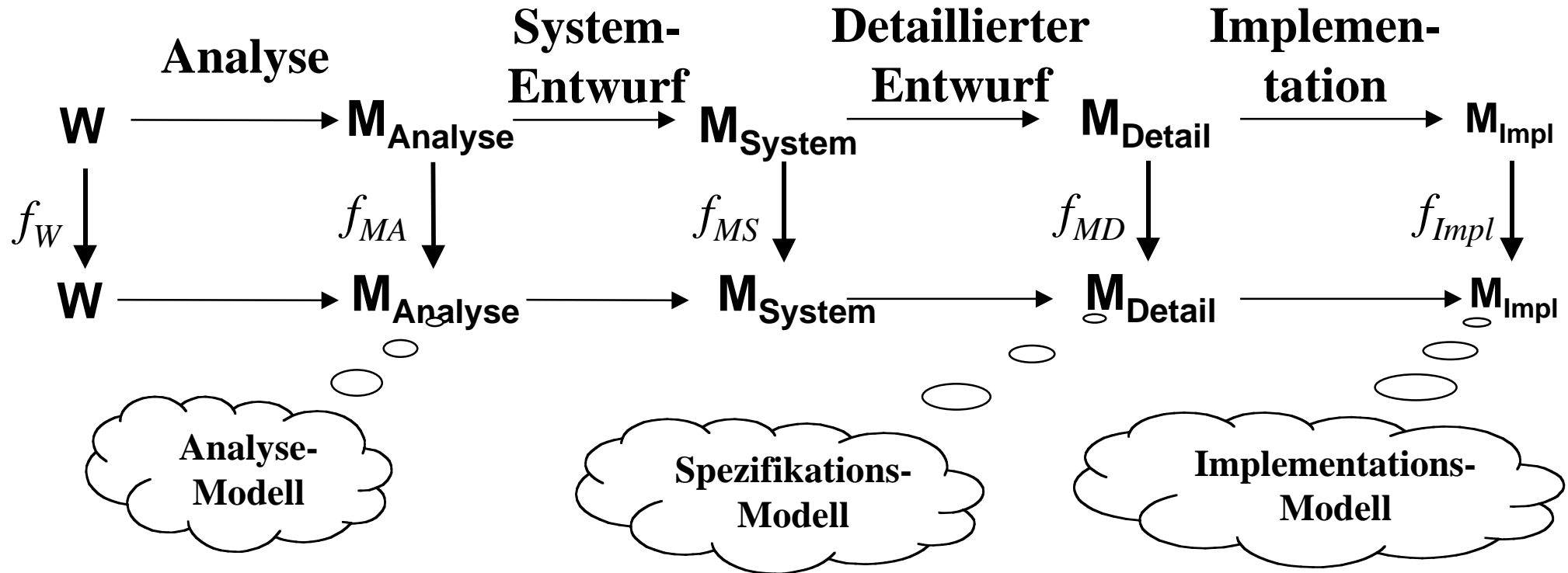
- ❖ Das **Spezifikationsmodell** wird während des detaillierten Entwurfs erstellt.
 - Hauptinteressenten sind Spezifizierer und Benutzer.
 - Das Klassendiagramm enthält Applikationsklassen und Schnittstellen.
- ❖ Das Spezifikationsmodell ist die Grundlage der Kommunikation zwischen dem Entwerfer und Implementierer.

Implementationsmodell

- ❖ Das **Implementationsmodell** wird während der Implementation erstellt.
 - Hauptinteressenten: Implementierer und Erweiterer.
 - Das Diagramm enthält Applikationsklassen, Schnittstellen und Lösungsklassen.
- ❖ Das Implementationsmodell ist die Grundlage für die Implementation in einer Programmiersprache, das heißt, es dient zur Kommunikation zwischen dem Programmierer und dem Rechner :-).

Entwicklungsaktivitäten sind Modelltransformationen

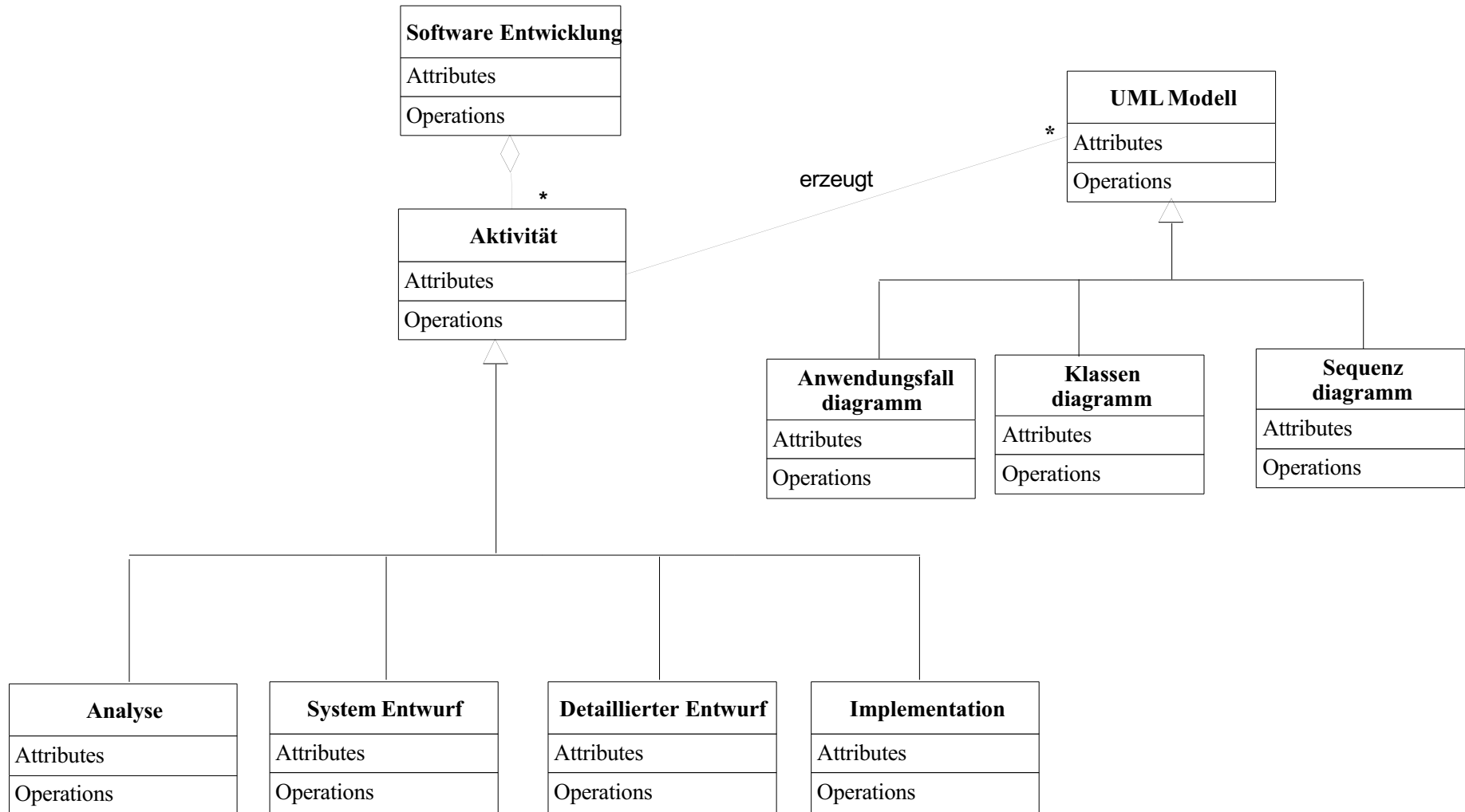
(siehe Info I - Vorlesung 2: Informatik-Systeme)



Ein Modell enthält ein oder mehrere Diagramme:

Klassendiagramme
Anwendungsfalldiagramme
Sequenzdiagramme

Modellierung der Entwicklungsaktivitäten in UML



Die Benutzer des Spezifikationsmodells

- ❖ Im Spezifikationsmodell gibt es für die verschiedenen Arten von Implementierern unterschiedliche Schnittstellen
- ❖ **Öffentliche Schnittstelle** (public interface):
 - Menge der Merkmale, die für den Klassen-Benutzer sichtbar sind.
- ❖ **Private Schnittstelle** (private interface):
 - Menge der Merkmale, die nur für den Klassen-Implementierer sichtbar sind.
- ❖ **Geschützte Schnittstelle** (protected interface):
 - Menge der Merkmale, die für den Klassen-Erweiterer sichtbar sind.
 -

Öffentliche Schnittstelle (public interface)

- ❖ Menge der Merkmale, die für den Klassen-Benutzer sichtbar sind.
- ❖ Ein **öffentliches Merkmal** (public member) ist überall im Modell sichtbar.
- ❖ Jede Klasse im System kann auf öffentliche Merkmale zugreifen

Private Schnittstelle (private interface)

- ❖ Menge der Merkmale, die nur für den Klassen-Implementierer sichtbar sind.
- ❖ Ein **privates Merkmal** (private member) darf nur in der definierenden Klasse verwendet werden.
- ❖ Der Klassen-Implementierer kann bei der Erstellung einer Klasse öffentliche, geschützte und private Merkmale dieser Klasse benutzen.

Geschützte Schnittstelle (protected interface)

- ❖ Menge der Merkmale, die für den Klassen-Erweiterer sichtbar sind.
- ❖ Ein **geschütztes Merkmal** (protected member) darf nur in der definierenden Klasse selbst und in Unterklassen dieser Klasse verwendet werden.
- ❖ Der Klassen-Erweiterer kann bei der Erstellung einer Unterklasse öffentliche und geschützte Merkmale ihrer Oberklasse(n) und öffentliche, geschützte und private Merkmale der Unterklasse benutzen.

Je nach Modell haben Assoziationen unterschiedliche Interpretationen

- ❖ Im **Analysemodell** (während der **Analyse**) interpretieren wir Assoziationen als *Beziehungen zwischen Klassen*:
 - Ein Auftrag stammt von einem einzelnen Kunden
 - Ein Kunde kann mehr als einen Auftrag haben
- ❖ Im **Spezifikationsmodell** (während des **Entwurfs**) interpretieren wir Assoziationen als *Verantwortlichkeiten*:
 - Die Klasse **Kunde** muss die Methode **kreditwürdigkeit()** bereitstellen, die von der Klasse **Auftrag** aufgerufen wird.
 - Die Klasse **Auftrag** muss eine Methode **getAuftragsposition()** bereitstellen, die für einen gegebenen Kunden die zum Auftrag gehörenden Auftragspositionen zurück gibt.
- ❖ Im **Implementationsmodell** (während der **Implementation**) interpretieren wir Assoziationen als *Variablen* (Java: Instanzvariablen):
 - Ein **Auftrag** hat eine Instanzvariable **Kunde**.
 - Ein **Auftrag** hat eine Reihung von Instanzvariablen, die jeweils auf eine **Auftragsposition** zeigen.

Je nach Modell haben Attribute unterschiedliche Interpretationen

- ❖ Im **Analysemodell** (während der **Analyse**) interpretieren wir Attribute als *Eigenschaften einer Klasse*:
 - Ein Kunde hat einen Namen
 - Ein Student hat eine Matrikelnummer
- ❖ Im **Spezifikationsmodell** (während des **Entwurfs**) interpretieren wir Attribute als Aufforderung, Methoden zu definieren, die das Attribut setzen oder lesen:
 - `public String getName()` teilt den Namen des Kunden mit.
 - `public void setName(String S)` setzt den Namen des Kunden.
- ❖ Im **Implementationsmodell** (während der **Implementation**) interpretieren wir Attribute als *Variablen* (Java: Instanzvariablen):
 - Ein **Kunde** hat ein Feld **Name** für seinen Namen.

Einschub: Was ist der Unterschied zwischen Assoziationen und Attributen?

- ❖ Attribute sind Assoziationen sehr ähnlich.
- ❖ Im Analysemodell gibt es gar keinen Unterschied. Attribute und Assoziationen sind lediglich alternative Schreibweisen. Multiplizitäten lassen sich leichter mit Assoziationen ausdrücken, aber man kann auch Reihungen verwenden:
 - **empfangsDatum[0..1] : Datum**
- ❖ Im Spezifikationsmodell bezeichnen Attribute lokale Klasseneigenschaften, während Assoziationen eine Navigierbarkeit im Klassenmodell implizieren:
 - Eine Assoziation zwischen **Auftrag** und **Kunde** sagt, dass ich von der Klasse **Auftrag** zu der Klasse **Kunde** kommen kann.
- ❖ Im Implementationsmodell werden Attribute und Assoziationen unterschiedlich realisiert:
 - Attribute entsprechen Variablen mit Wertsemantik
 - Assoziationen entsprechen Variablen mit Referenzsemantik

Je nach Modell hat Vererbung eine unterschiedliche Bedeutung

- ❖ Im **Analysemodell** gilt:
 - Alles, was für die Oberklasse gilt, trifft auch auf die Unterklasse zu.
- ❖ Im **Spezifikationsmodell** gilt:
 - Die Schnittstelle der Unterklasse erbt die **Schnittstelle** der Oberklasse. Hier interessiert uns die *Spezifikationsvererbung* (specification inheritance), auch *Schnittstellenvererbung* genannt.
- ❖ Die Vererbung im **Implementationsmodell** hängt stark davon ab, was die Programmiersprache an Vererbungskonzepten zu bieten hat, aber im allgemeinen gilt:
 - Die Unterklasse erbt alle Methoden und Felder der Oberklasse (incl. ihrer **Implementation**) und kann geerbte Methoden überschreiben. Hier interessiert uns die *Implementationsvererbung* (implementation inheritance).

Beispiele für Spezifikations- und Implementationsvererbung: Siehe Info I - Vorlesung 13.

Ansonsten → In den Vorlesungsblöcken "Polymorphismus" + "Entwurf durch Verträge"

Weitere UML-Diagrammarten

- ❖ Klassendiagramme beschreiben die **statische** Struktur eines Systems, aber nicht sein funktionelles oder dynamisches Verhalten. Dafür gibt es in UML andere Diagrammarten.
- ❖ Funktionelles Verhalten:
 - Ein **Anwendungsfalldiagramm** (use case diagram) beschreibt die *Funktionalität* des Systems (Beispiele für Funktionalität: Abschließen einen Auftrags, Ausführen des Auftrags)
- ❖ Dynamisches Verhalten:
 - Ein **Sequenzdiagramm** beschreibt das *dynamische Verhalten* zwischen mehreren Objekten, und zwar die Reihenfolge von Nachrichten, die zwischen den Objekten ausgetauscht werden (Beispiele für dynamisches Verhalten: Interaktion zwischen Web-Browser und Web-Server über HTTP, allgemein: jedes Protokoll)
 - Ein **Zustandsdiagramm** beschreibt das *dynamische Verhalten* eines Objekts (Beispiel: Verkaufsautomat, Stern).

Welche UML-Diagramme behandeln wir in Info II?

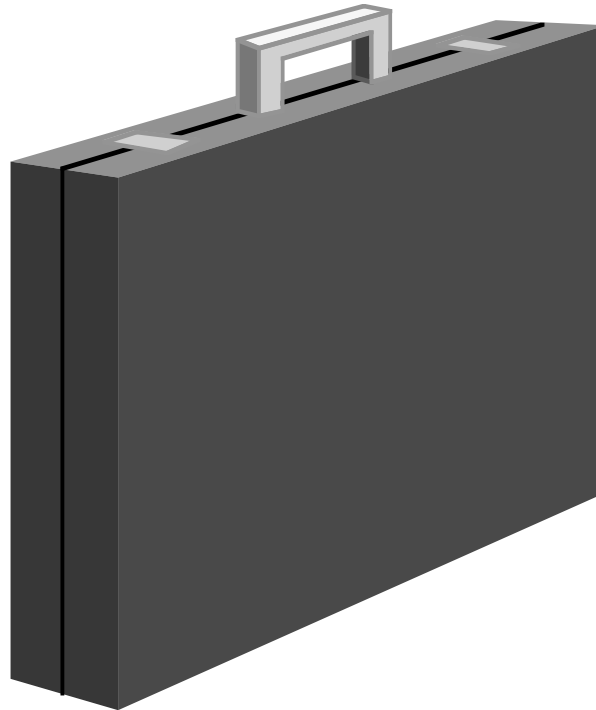
- ❖ UML kennt noch viele andere Diagrammarten, unter anderem Kollaborationsdiagramme, Aktivitätsdiagramme, ...
 - ⇒ Hauptstudium (Software Engineering)
- ❖ In Info II behandeln wir nur diese vier UML-Diagrammarten:
 - **Klassendiagramme**: bereits gemacht
 - **Anwendungsfalldiagramme**: heute
 - **Sequenzdiagramme** und **Zustandsdiagramme**
 - ⇒ im Vorlesungsblock "Ereignis-basierte Programmierung"
im Zusammenhang mit endlichen Automaten.

Anwendungsfall

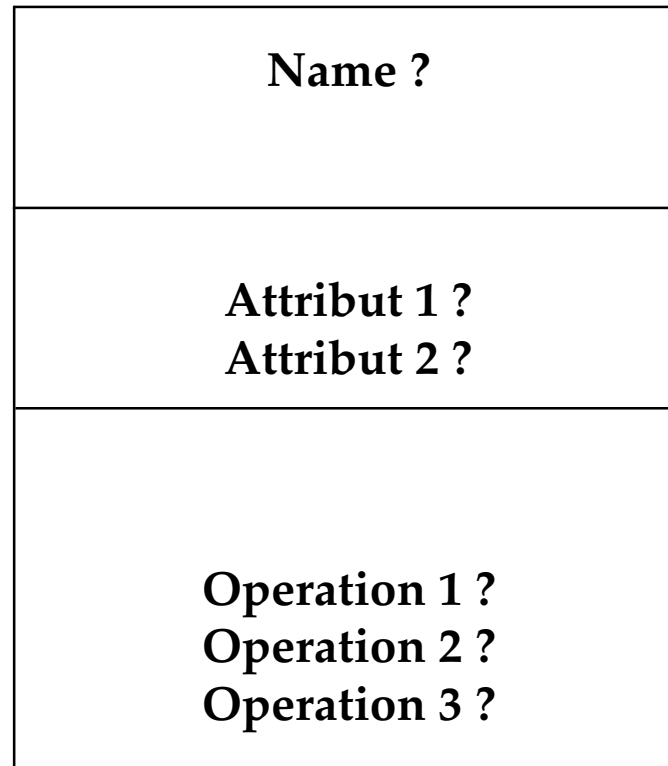
- ❖ Ein Anwendungsfall beschreibt eine Funktion des Systems als eine Folge von Ereignissen, mit einem sichtbaren Result für den Benutzer.

Warum brauchen wir Anwendungsfälle?

Was ist das hier?



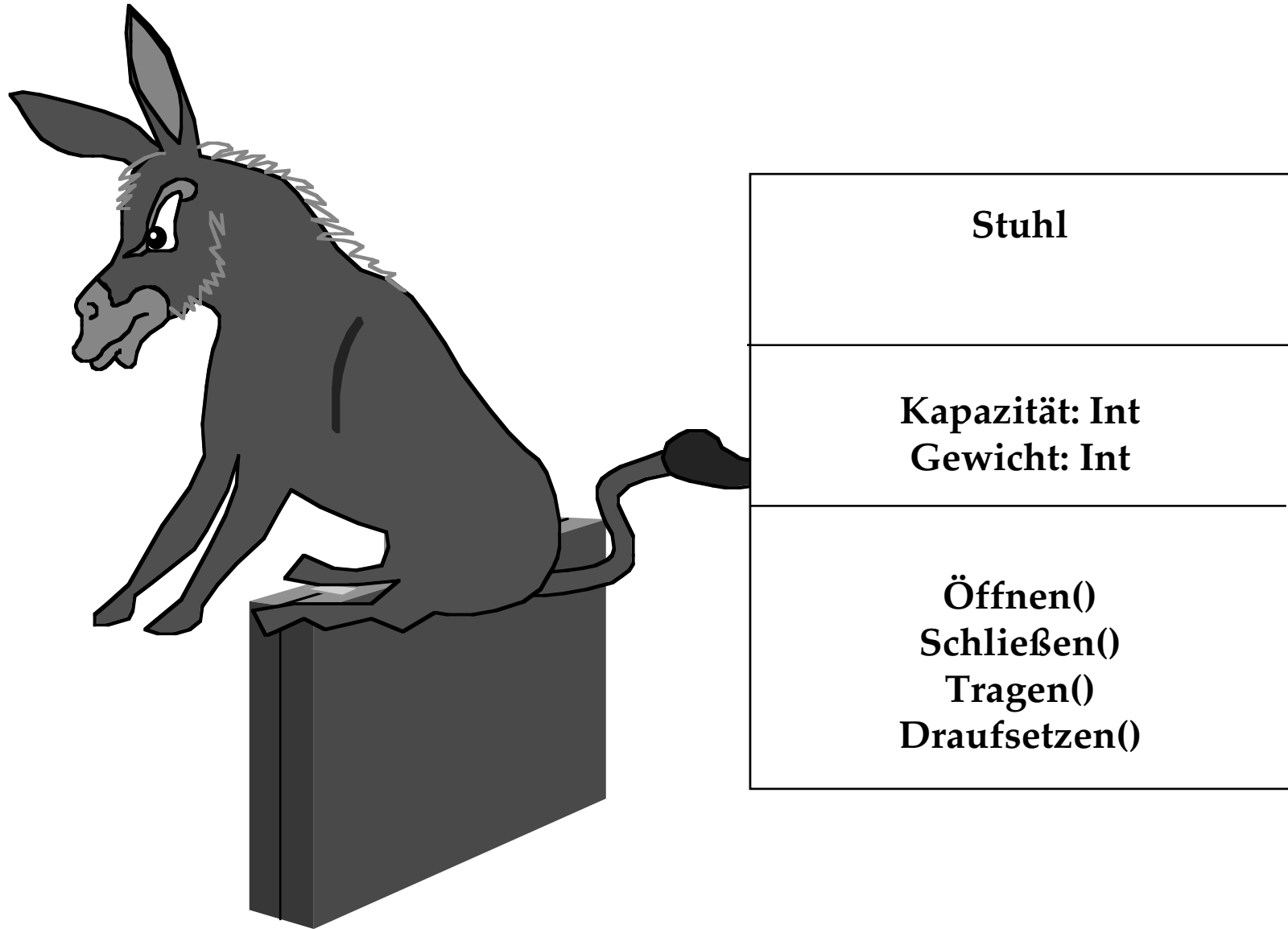
Modellieren wir doch mal das Problem als Klassendiagramm



Ergebnis unserer Analyse der Anwendungsdomäne

| |
|--|
| Aktentasche |
| Kapazität: Int Gewicht: Int |
| Öffnen() Schließen() Tragen() |

Leider benutzen wir die Klasse in diesem Fall ganz anders!



Fragen

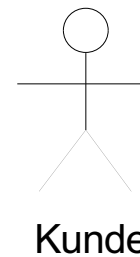
- ❖ Warum haben wir das Ding als “Aktentasche” modelliert?
- ❖ Warum haben wir es nicht als Stuhl modelliert?
- ❖ Was machen wir, wenn **Draufsetzen ()** die am meisten benutzte Operation ist?
- ❖ Wie können wir solche Modellierungsfehler verhindern?
⇒ Wir fangen mit Anwendungsfällen an

Anwendungsfall

- ❖ Ein Anwendungsfall beschreibt eine Funktion des Systems als eine Folge von Ereignissen, mit einem sichtbaren Result für den Benutzer.
- ❖ **Definition Anwendungsfall (use case):** Eine Folge von Schritten (sequence of events), die die Interaktion eines Benutzers mit einem System beschreibt.
- ❖ Ein Anwendungsfall besteht aus mindestens 3 Teilen:
 - **Name**
 - **Akteure**
 - **Ereignisfolge**
 - Ein vollständig formulierter Anwendungsfall hat noch 3 weitere Teile:
Eingangsbedingung, Ausgangsbedingung, Spezielle Anforderungen. (→Vorlesungsblock "Entwurf durch Verträge")
- ❖ Wir beschreiben Anwendungsfälle anhand eines Beispiels

Akteur

- ❖ **Definition Akteur (actor):** Die von einem Benutzer in Bezug auf das System eingenommene Rolle.
- ❖ **Beispiel:** Kunde, Kassierer.
- ❖ Akteure werden in UML-Anwendungsfalldiagrammen als Strichfigur gezeichnet.



Modellierungsbeispiel mit einem Anwendungsfall: Erstellung eines abstrakten Szenarios

- ❖ Zunächst beschreiben wir in natürlicher Sprache in einem **konkreten Szenario**, wie das System benutzt wird:

Beispiel: Joe durchstöbert den Quelle-Katalog und legt zwei DVDs in seinen Einkaufskorb. Zum Zahlen gibt er seine Mastercard-Kreditkartennummer an. Er wählt Normalversand aus und klickt **OK**. Das System checkt die Kreditkarte, dann bestätigt es den Verkauf mit einer E-Mail an joe@mac.com.

- ❖ Aus dem konkreten Szenario machen wir ein **abstraktes Szenario**, in dem wir konkrete Namen (Joe, DVD,...) durch Klassen (Kunde, Artikel, ...) ersetzen:

Beispiel: Der Kunde durchstöbert einen Katalog und legt die ausgewählten Artikel in seinen Einkaufskorb. Zum Zahlen gibt er seine Versand- und Kreditkarteninformation an und bestätigt seinen Kauf. Das System autorisiert den Verkauf über eine Kreditkarte und bestätigt den Verkauf mit einer Quittung und einer E-Mail an die E-Mail-Adresse aus der Versandinformation.

Vom abstrakten Szenario zum Anwendungsfall

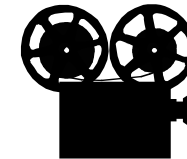
Wir machen aus dem abstrakten Szenario einen Anwendungsfall "Einkauf eines Artikels", indem wir die Komponenten auflisten:

1. Name: Einkauf eines Artikels

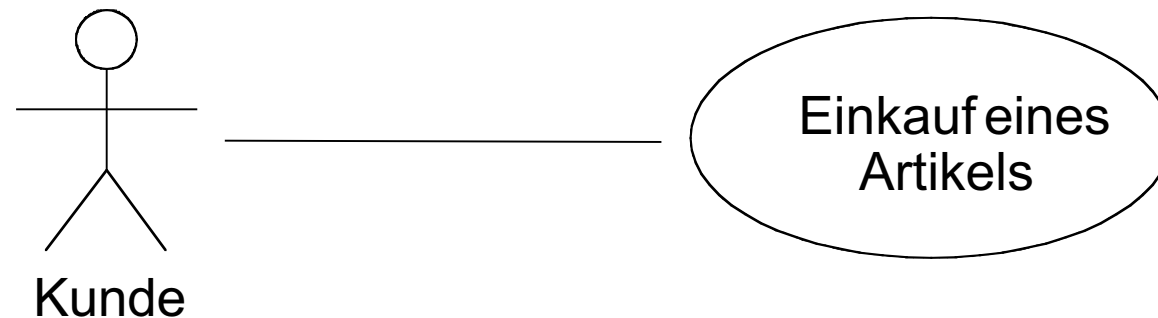
2. Akteur: Kunde.

3. Ereignisfolge:

1. Der Kunde stöbert durch den Katalog
2. Der Kunde wählt einen Artikel zum Kauf aus.
3. Der Kunde geht zur Kasse.
4. Der Kunde gibt seine Versand-Information an
(Adresse, Express- oder Normalversand)
5. Der Kunde gibt Kreditkartendaten an.
6. Das System autorisiert den Verkauf
7. Das System bestätigt den Verkauf sofort.
8. Das System sendet eine Bestätigung per E-Mail an den Kunden

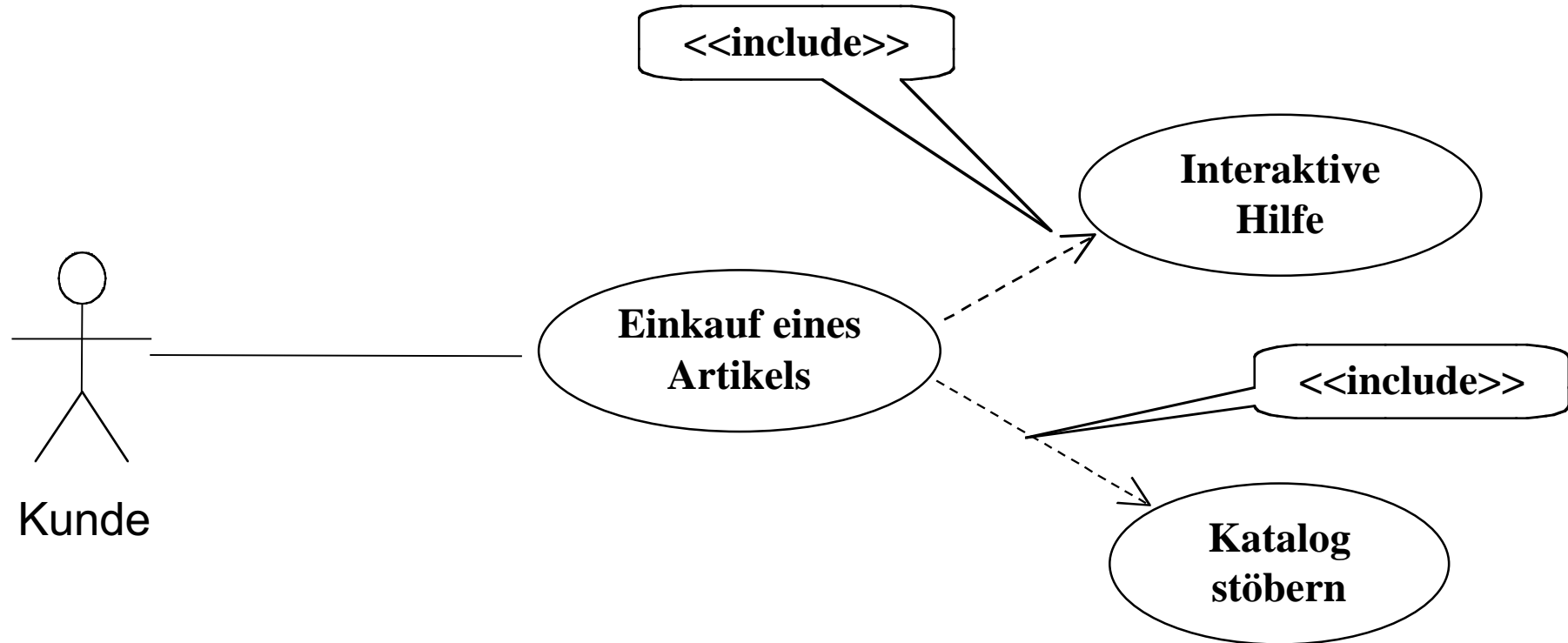


Darstellung des Anwendungsfalles in UML



- ❖ Eine **Beziehung** zwischen dem **Akteur** "Kunde" und dem **Anwendungsfall** "Einkauf eines Artikels" wird durch eine allgemeine Assoziation beschrieben.
- ❖ Anwendungsfälle können auch Beziehungen zu anderen Anwendungsfällen haben.

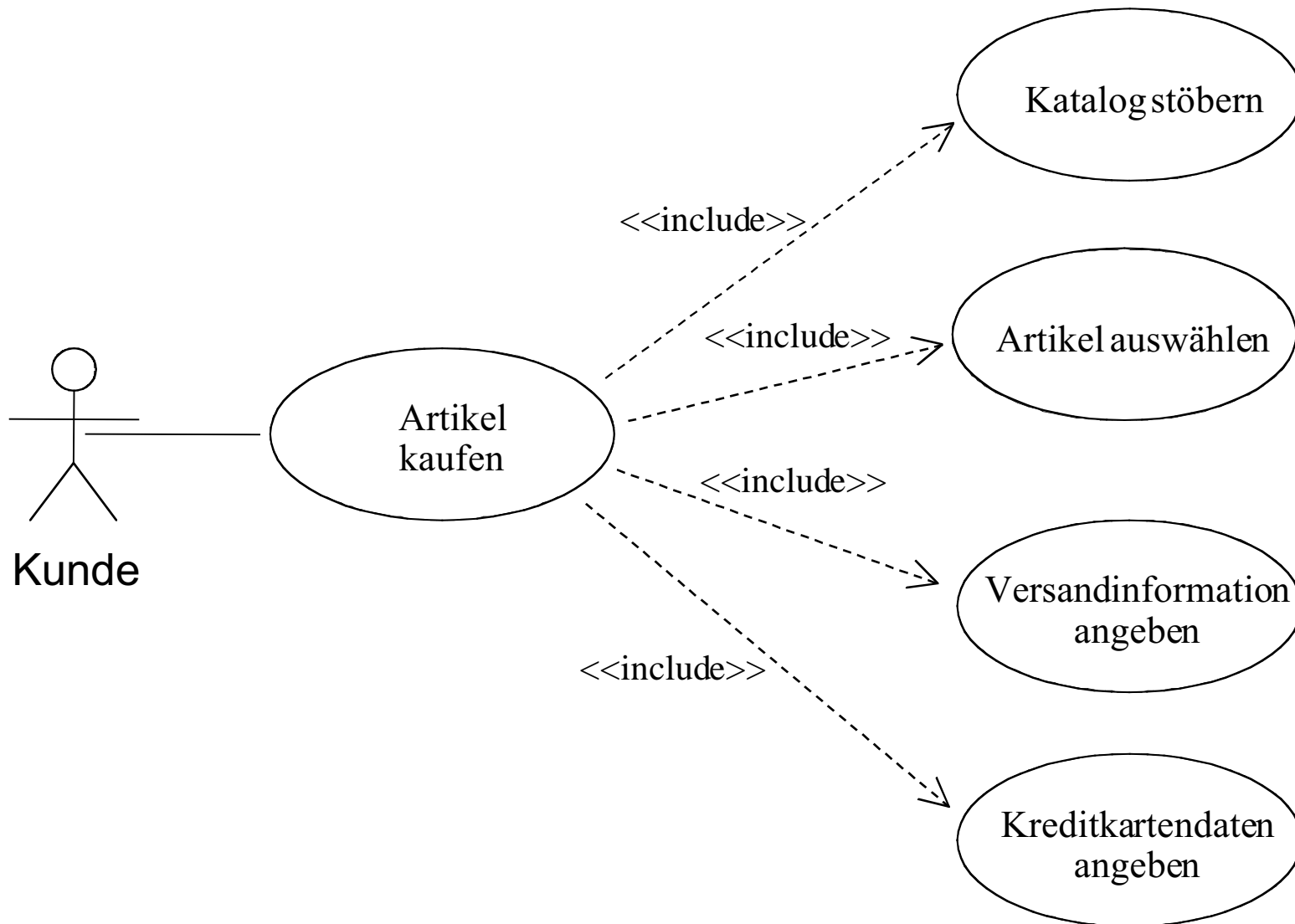
Es gibt auch Beziehungen zwischen Anwendungsfällen



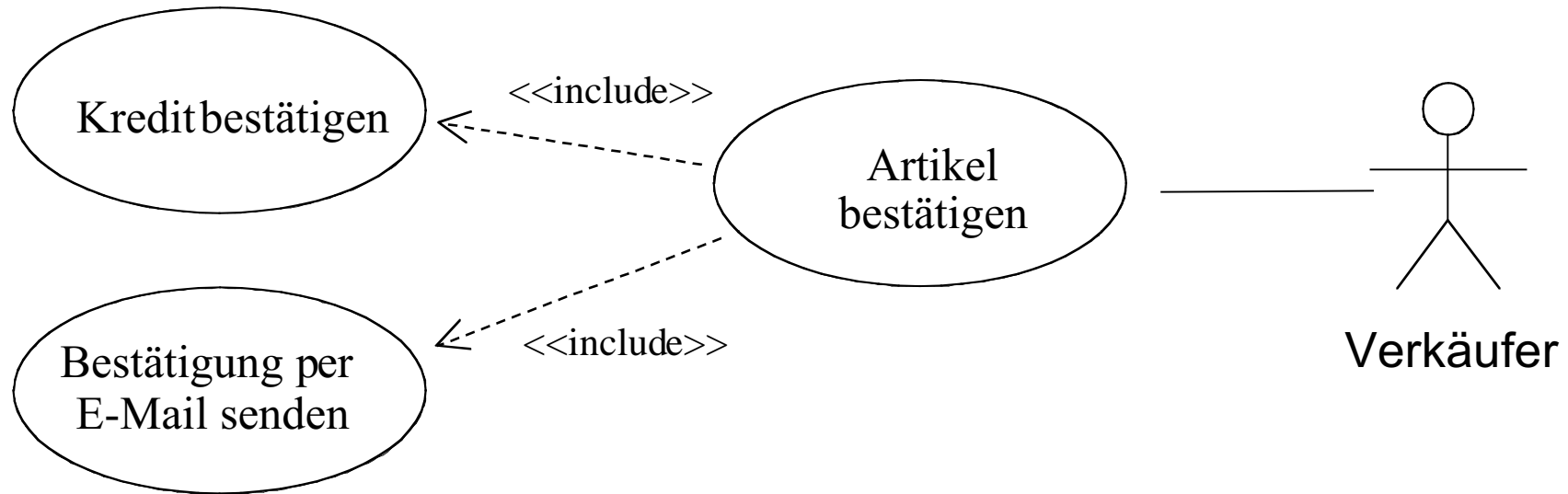
Beziehungen zwischen Anwendungsfällen

- ❖ Neben der Beziehung zwischen Akteuren und Anwendungsfällen gibt es auch Beziehungen zwischen Anwendungsfällen:
- ❖ **Enthält-Beziehung (include):**
 - Eine bestimmte Funktionalität tritt in mehreren Anwendungsfällen auf.
Dann kann man sie als separaten Anwendungsfall "herausfaktorisieren" (funktionale Dekomposition).

Beispiel für Funktionale Dekomposition

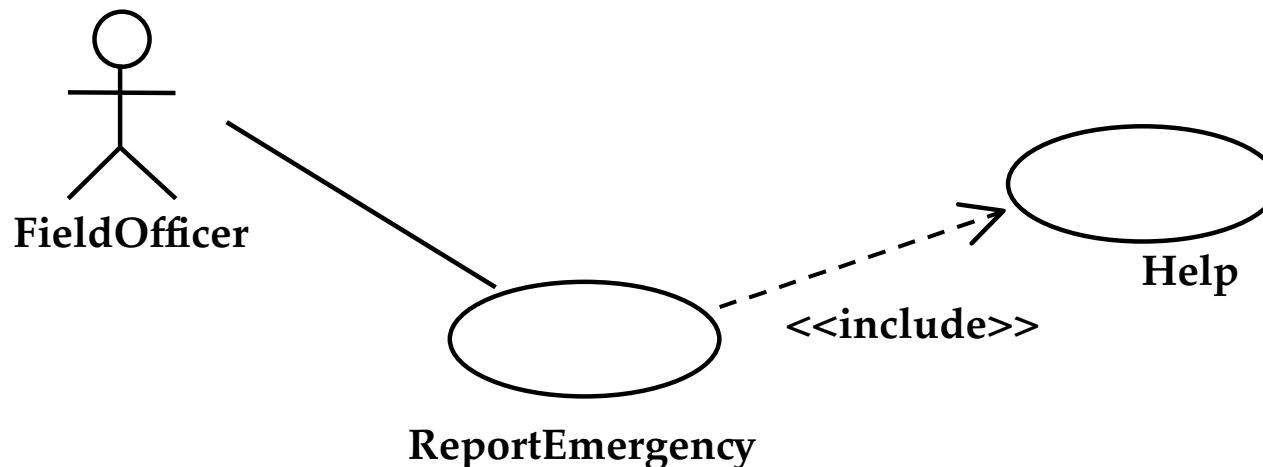


Funktionale Dekomposition von "Artikel bestätigen"



Includes-Beziehung

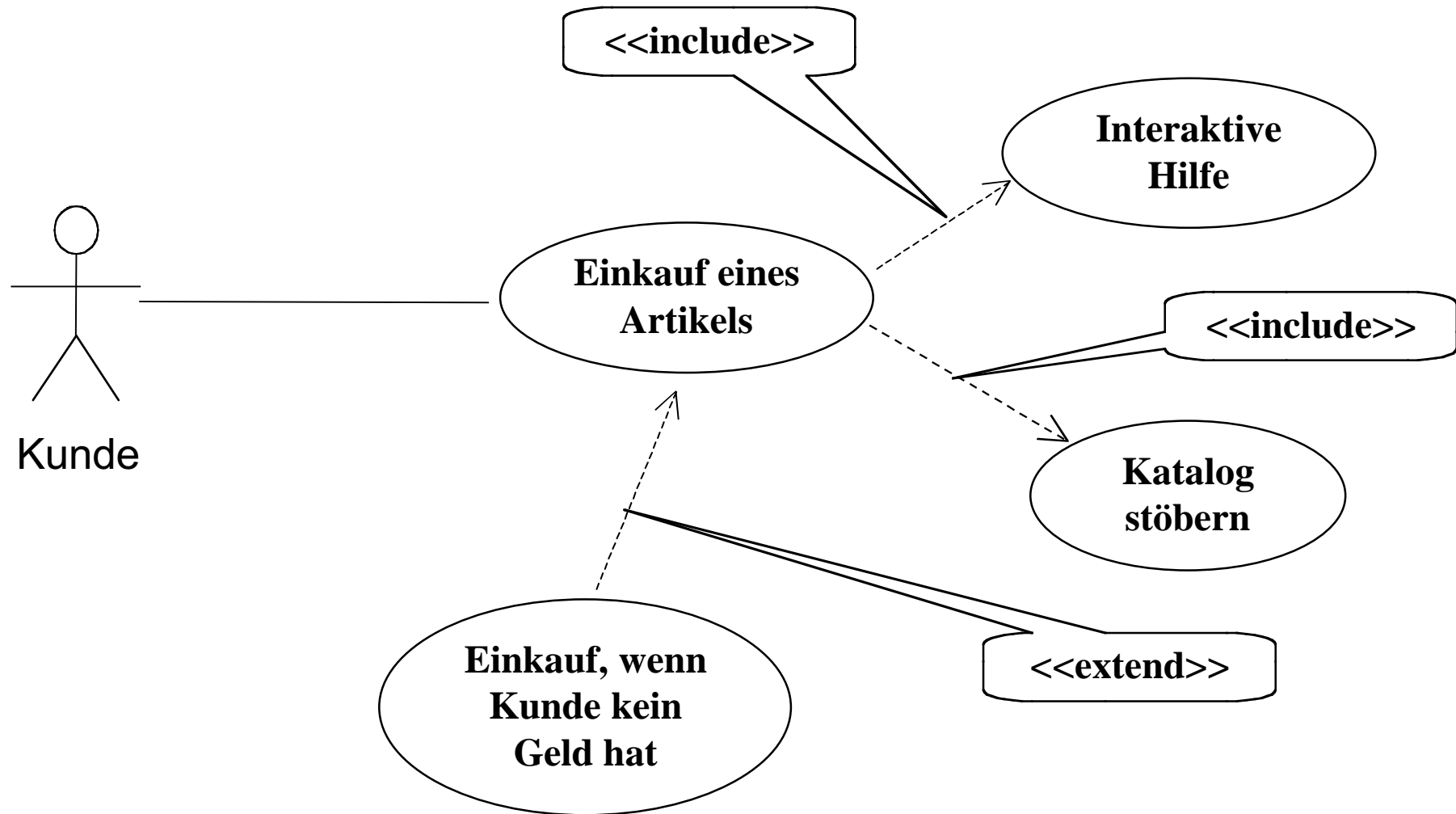
- ❖ **Problem:** Die Funktionalität A in der ursprünglichen Problembeschreibung soll um eine andere, bereits vorhandene Funktionalität B erweitert werden. Die Funktionalität B kann in mehr als einem Anwendungsfall auftreten.
- ❖ **Lösung:** Eine `<<include>>`-Beziehung von Anwendungsfall A zu Anwendungsfall B.
- ❖ **Beispiel:** Der Anwendungsfall "ReportEmergency" beschreibt die Folge von Schritten bei der Abwicklung eines Notfalls. Für ein bestimmtes Szenario ("Der Benutzer ist ein Anfänger") kann der "Help"-Anwendungsfall benutzt werden.



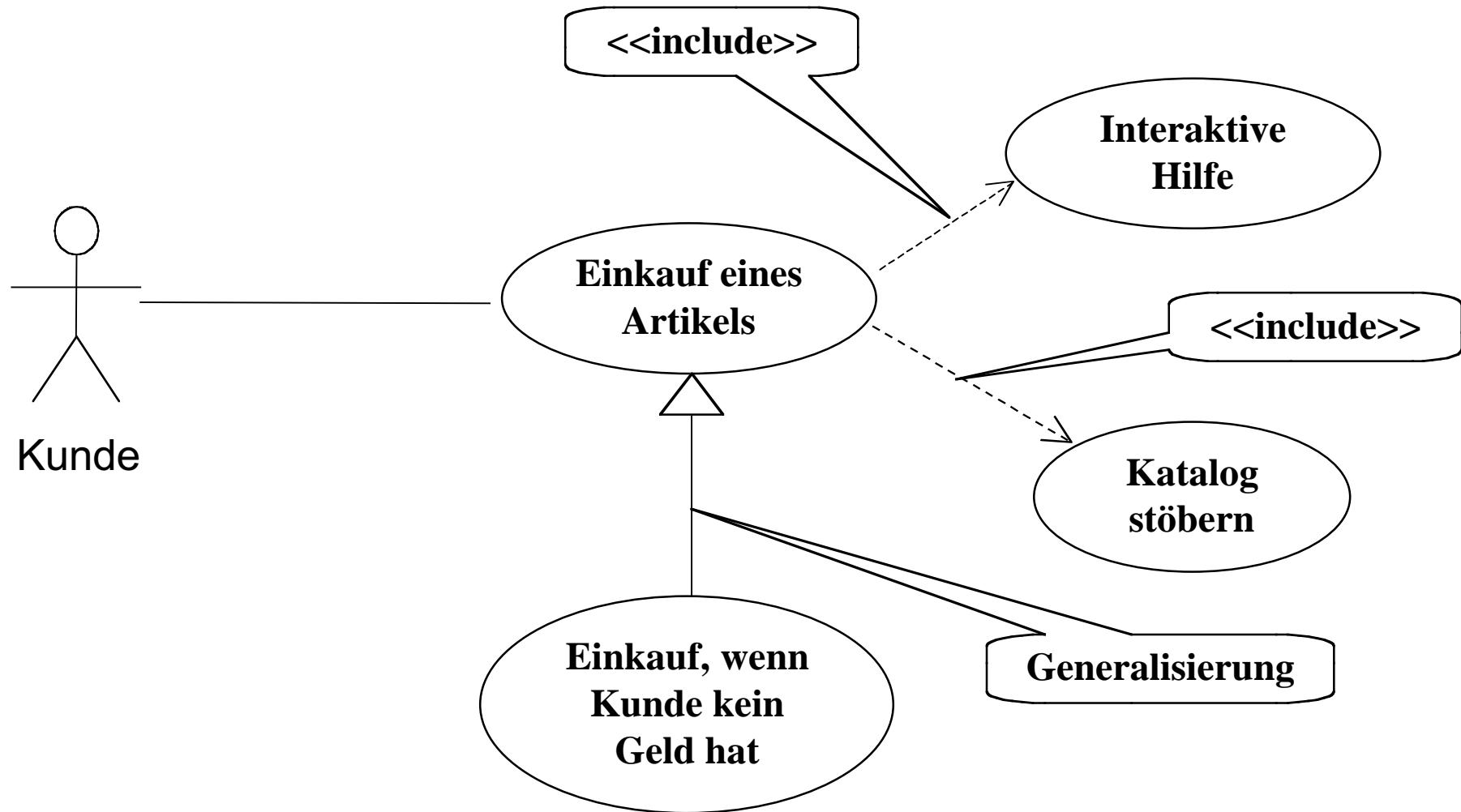
Heuristiken für die Modellierung mit Anwendungsfällen

- ❖ Während der Ausarbeitung von Anwendungsfällen ist es oft sinnvoll, komplizierte Anwendungsfälle in kleinere Anwendungsfälle aufzuteilen ("Divide and impera").
 - Die Aufteilung eines komplexen Anwendungsfalles in kleinere Anwendungsfälle (mit Hilfe der <<include>>-Beziehung) nennen wir **funktionale Dekomposition**.
- ❖ Wie findet man die kleineren Anwendungsfälle?
 - Ein guter Ausgangspunkt für die kleineren Anwendungsfälle ist die Ereignisfolge des komplexen Anwendungsfalles.
- ❖ Was ist, wenn ein Anwendungsfall viele Varianten hat?
 - Unterteile ihn in einen normalen Anwendungsfall und einen oder mehrere erweiternde Anwendungsfälle (mit <<extend>>).
 - In der ersten Entwicklungsphase (Projektphase 1) implementiere den normalen Anwendungsfall.
 - Implementiere die erweiternden Anwendungsfälle in Phase 2, Phase 3,....

Noch eine Beziehung zwischen Anwendungsfällen: Erweiterung von Anwendungsfällen mit <<extend>>



Noch eine Beziehung zwischen Anwendungsfällen: Verallgemeinerung (Generalisierung)



Verallgemeinerung und Erweiterung von Anwendungsfällen

❖ **Verallgemeinerungs-Beziehung (generalization):**

- Für einen speziellen Anwendungsfall wird ein allgemeinerer Anwendungsfall mit derselben Grundfunktionalität angegeben. Wir modellieren dann, ausgehend vom **speziellen Anwendungsfall**, einen **Basis-Anwendungsfall**.

❖ **Erweiterungs-Beziehung:**

- Ähnelt der Generalisierungs-Beziehung, erlaubt aber die Spezifikation von **Erweiterungsstellen** im Basis-Anwendungsfall. Ausgehend vom Basis-Anwendungsfall können dann spezielle Anwendungsfälle modelliert werden.

- ❖ Verallgemeinerung und Erweiterung unterscheiden sich in der Reihenfolge, in der ein Basis-Anwendungsfall und die zugehörigen speziellen Anwendungsfälle modelliert werden.

⇒ Hauptstudium

Wie finde ich die "richtigen" Objekte?

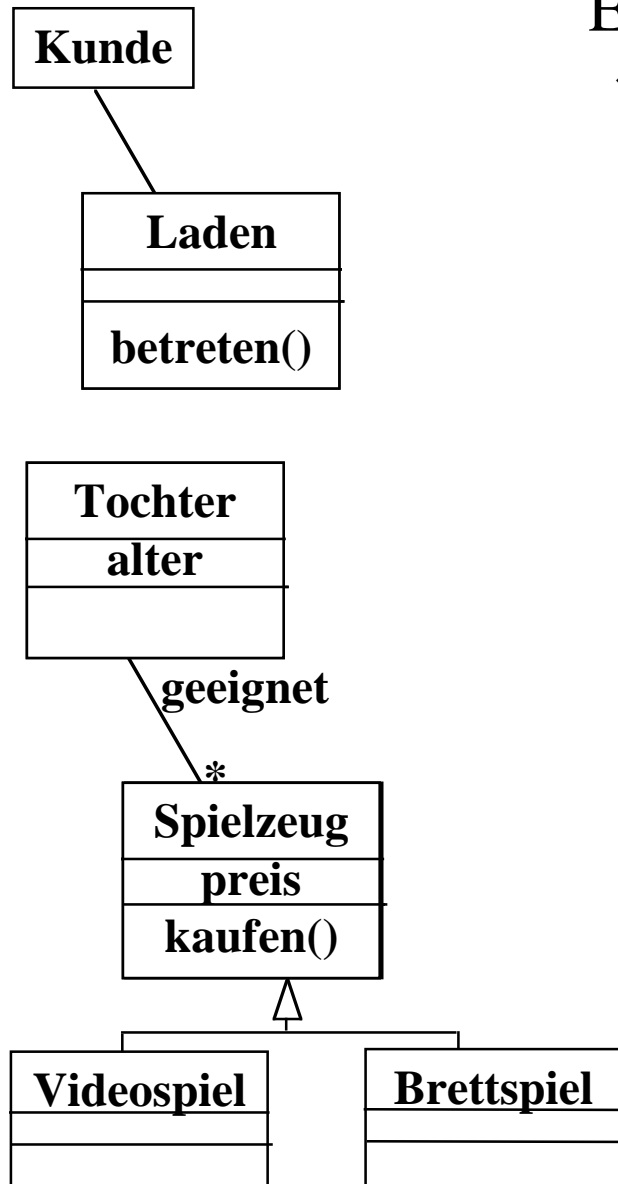
- ❖ Wir sagen: Objekt-Orientierung ist gut
- ❖ Wir haben gesehen:
 - Wenn man die Analyse naiv mit der Objekt-Identifizierung beginnt, kann man leicht die falschen Objekte finden, was zu einem falschen System führt ("GIGO: garbage in, garbage out")
- ❖ Wir haben deshalb auch gesagt:
 - Eine gute Modellierung fängt mit der Identifizierung der Funktionen an, die das Informatik-System bieten soll.
- ❖ Wenn wir mit der Modellierung der Funktionalität anfangen sollen, wie finden wir dann Objekte und Klassen?
- ❖ **Definition Teilnehmende Objekte (participating objects):**
Objekte oder Klassen, die an einem Anwendungsfall teilnehmen.
 - Teilnehmende Objekte findet man, indem man sich Ereignisfolgen genauer anschaut.

Wie finde ich teilnehmende Objekte/Klassen?

- ❖ **Verwendung von vielen verschiedenen Wissensquellen:**
 - Interviews mit *Kunden* und *Experten*, um die Abstraktionen der Anwendungsdomäne zu identifizieren
 - *Entwurfsmuster*, um Abstraktionen in der Lösungsdomäne zu identifizieren
 - *Allgemeines Wissen* und *Intuition*
- ❖ **Formulierung von Szenarios** (in natürlicher Sprache):
 - Beschreibung der konkreten Benutzung des Systems.
- ❖ **Formulierung von Anwendungsfällen** (natürliche Sprache und UML):
 - Beschreibung der Funktionen mit Akteuren und Ereignisfolgen
- ❖ **Syntaktische Untersuchung** von Substantiven, Verben und Attributen:
 - In der Problembeschreibung (oft nicht machbar, da zu umfassend)
 - In den Ereignisfolgen der Anwendungsfälle (sehr gut machbar).

⇒ textuelle Analyse nach Abbot

Beispiel: Erzeugung eines Klassendiagramm aus einem Anwendungsfall



Ereignisfolge eines Anwendungsfalls:

- ❖ Der **Kunde** Joe Smith **betritt** den **Laden** um ein **Spielzeug** zu **kaufen**. Es soll ein Spielzeug sein, das seiner **Tochter** gefallen muss und **weniger als 50 DM** kostet. Er probiert ein **Videospiel** aus, das aus einem Handschuh und einem Kopfbildschirm besteht. Es gefällt ihm.
- ❖ Eine Hilfskraft berät ihn. Die Eignung des Spiels **hängt davon ab**, wie **alt** das Kind ist. Joe's Tochter ist erst 3 Jahre alt. Die Hilfskraft empfiehlt eine andere **Art von Spielzeug**, nämlich das **Brettspiel** "Mensch ärgere Dich nicht".

Textuelle Analyse nach Abbot

In [Goos II] auch "Substantiv-Verb-Analyse" genannt

| <i>Beispiel</i> | <i>Satz-Konstrukt</i> | <i>UML-Komponente</i> |
|---|------------------------|-----------------------|
| "Joe Smith" | konkrete Person, Ding | Objekt |
| "Spielzeug" | Substantiv | Klasse |
| "3 Jahre alt" | Adjektiv | Attribut |
| "betreten", "Hilfe suchen" | allgemeines Verb | Operation |
| "Hängt ab von...." | intransitives Verb | Operation (Ereignis) |
| "ist ein..." , "entweder...oder", "Art von..." | Klassifizierung | Vererbung |
| "hat ein", "gehört zu" | besitzanzeigendes Verb | Aggregation |
| "muss...sein", "ist weniger ...als" | Bedingung | Einschränkung |

Reihenfolge bei der Modellierung

1. Zunächst einmal **formulieren wir einige Szenarien**, möglichst mit Hilfe des Kunden.
2. Dann **extrahieren wir die Anwendungsfälle** aus den Szenarien, möglichst mit Experten aus der Anwendungsdomäne
3. Dann machen wir eine **Analyse der Ereignisfolgen** der Anwendungsfälle, z.B. mit Hilfe von Abbot's Textueller Analyse.
4. Dann erst **erzeugen wir Klassendiagramme**. Dies beinhaltet die folgenden Schritte:
 - 4.1. *Klassen* finden (textuelle Analyse, Domänenexperten).
 - 4.2. *Attribute und Operationen* finden (manchmal allerdings auch, bevor die Klassen identifiziert sind!)
 - 4.3. *Assoziationen* zwischen Klassen finden
 - 4.4. *Multiplizitäten* bestimmen
 - 4.5. *Rollen* bestimmen
 - 4.6. *Einschränkungen* bestimmen

Zusammenfassung der wichtigsten Konzepte

- ❖ Zentraler Bestandteil eines Informatik-Systems ist dessen **Beschreibung**
- ❖ Wir verwenden zwei verschiedene Beschreibungsformen:
 - Analyse- und Entwurfsebene: **Modellierungssprachen** (UML)
 - UML-Notationen in Info II: Klassendiagramme, Anwendungsfälle, Sequenzdiagramme, Zustandsdiagramme.
 - Implementationsebene: **Programmiersprachen** (in Info II: Java)
- ❖ **Modelle**: Analysemodell, Spezifikationsmodell, Implementationsmodell
 - Modelle in UML sind einfacher zu kommunizieren als Code
- ❖ **Unterschiedliche Benutzer von Modellen**: Kunde, Experte, Analytiker, Entwerfer (System-Entwerfer, Klassen-Entwerfer), Implementierer (Klassen-Benutzer, -Implementierer, -Erweiterer)
- ❖ Eine gute Modellierung beginnt mit Anwendungsfällen, und identifiziert dann die **teilnehmenden Klassen/Objekte**.
 - Wenn man mit Klassendiagrammen bei der Modellierung anfängt, findet man oft die falschen Klassen.