

# Software Engineering

## SS 2005

**Prof. Dr. Barbara Paech, Jürgen Rückert**



Institut für Informatik  
Im Neuenheimer Feld 326  
69120 Heidelberg  
<http://www-swe.informatik.uni-heidelberg.de>  
[paech@informatik.uni-heidelberg.de](mailto:paech@informatik.uni-heidelberg.de)



RUPRECHT-KARLS-UNIVERSITÄT HEIDELBERG

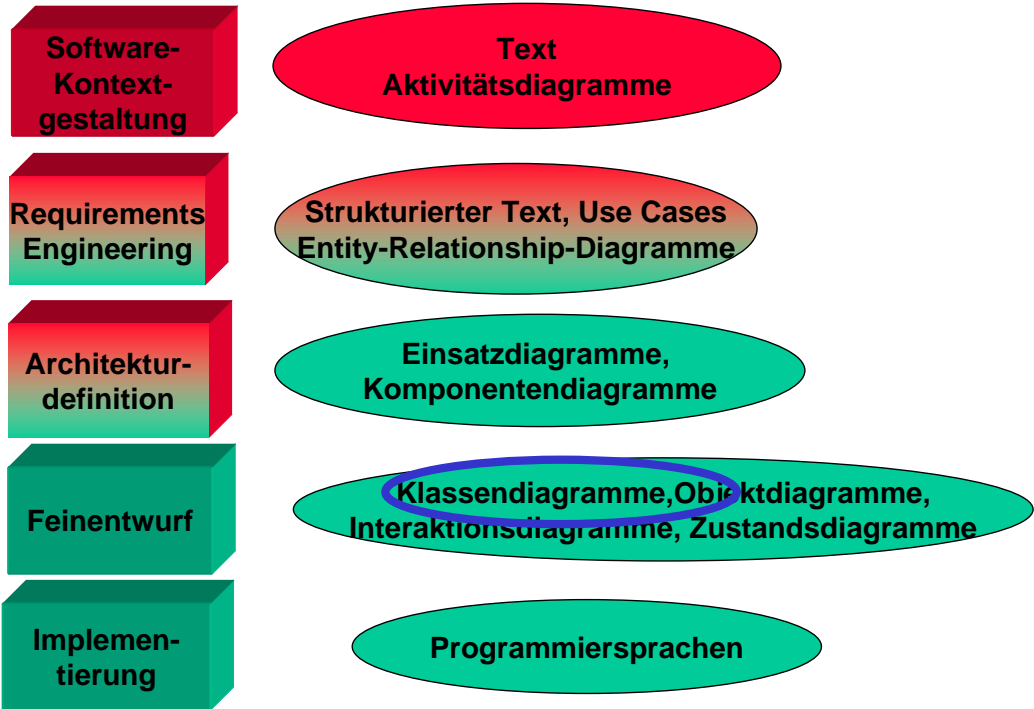


- 2. Beschreibungstechniken
  - 2.1. Modellierung
  - 2.2. Objektorientierung
  - 2.3. Klassendiagramme



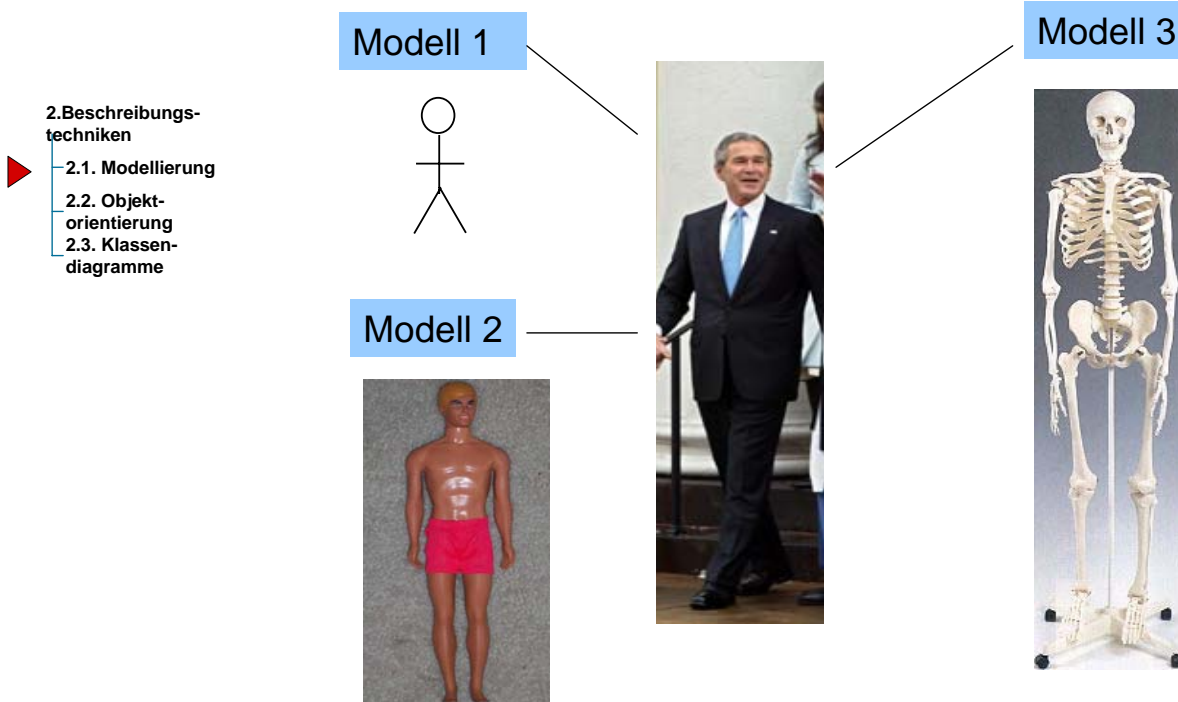
## 1.3.5. Beschreibungstechniken

2. Beschreibungstechniken
- 2.1. Modellierung
  - 2.2. Objektorientierung
  - 2.3. Klassendiagramme



## Modellierung

2. Beschreibungstechniken
- 2.1. Modellierung
  - 2.2. Objektorientierung
  - 2.3. Klassendiagramme



## 2.1. Was ist ein Modell?

- ◆ Ein Modell ist seinem Wesen nach **eine in Maßstab, Detailliertheit und/oder Funktionalität verkürzte beziehungsweise abstrahierte Darstellung** des originalen Systems [Stachowiak 73]
- ◆ Ein Modell ist eine **Abstraktion eines Systems mit der Zielsetzung das Nachdenken** über ein System zu vereinfachen, indem irrelevante Details ausgelassen werden [Brügge 00].
- ◆ **Beispiele** für Modelle:
- ◆ Zur Darstellung von Modellen ist eine **Notation** notwendig.
  - In SWE typisch: Text, Diagramm, Tabelle, Formeln, Programmtext, Strukturierter Text (z.B. eingeschränkte Satzform), Pseudocode

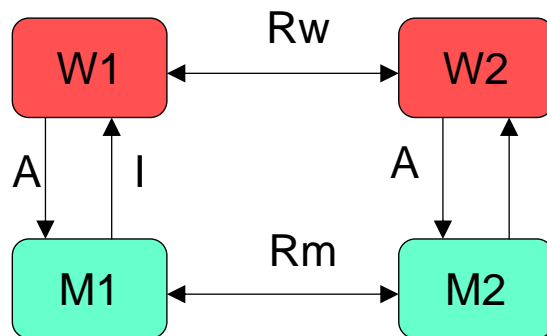
## 2.1. Bestandteile einer Notation

- ◆ **Syntax**
  - Konkretes Aussehen
  - Abstrakte Syntax
  - Kontextbedingungen zur Wohlgeformtheit
- ◆ **Semantik**
  - Bedeutungsbeschreibung
  - Interpretation in der realen Welt (bzw. der beschriebenen Domäne)
- ◆ **Pragmatik (Methodik des Gebrauchs)**
  - Analysetechniken (Typprüfung, Konsistenzchecks)
  - Simulationstechniken
  - Transformationstechniken (Z.B. Refactoring)
  - Generierungsmöglichkeiten
- ◆ Bilden eine „**Theorie**“ zur Bearbeitung der in einer Notation getroffenen Aussagen.

## 2.1. Modell und Wirklichkeit

2. Beschreibungs-  
techniken

- ▶ 2.1. Modellierung
- 2.2. Objekt-orientierung
- 2.3. Klassen-diagramme

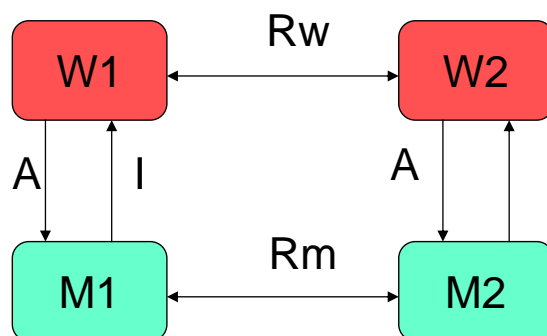


- W1, W2: Elemente der realen Welt
- Rw: Relation in der realen Welt
- M1, M2: Elemente im Modell
- Rm: Relation im Modell
- A: **Abstraktion** (Abbildung reale Welt -> Modell)
- I: **Interpretation** (Abbildung Modell -> reale Welt)

## 2.1. Was ist ein gutes Modell?

2. Beschreibungs-  
techniken

- ▶ 2.1. Modellierung
- 2.2. Objekt-orientierung
- 2.3. Klassen-diagramme

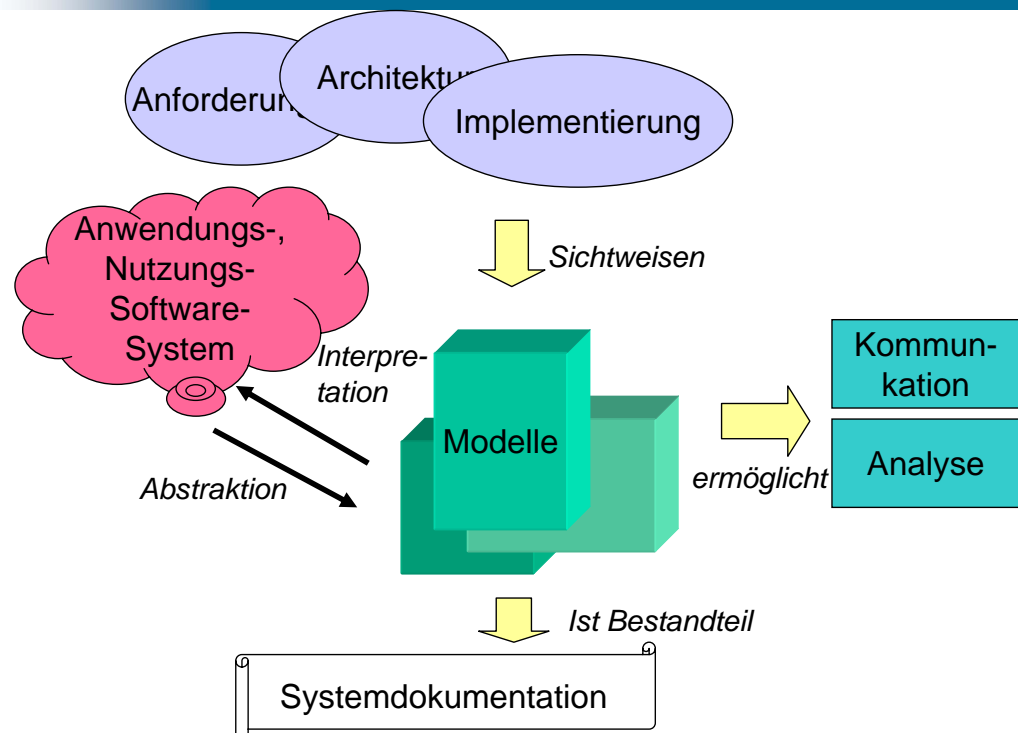


- Ein Modell ist **gut**, wenn das Diagramm kommutativ ist, d.h. Beziehungen im Modell gelten genau dann, wenn sie auch in der Wirklichkeit gelten

## 2.1. Wozu braucht man Modelle?

- ◆ Modelle helfen **komplexe Zusammenhänge zu verstehen**
  - Bildhafte Darstellung und Abstraktion unterstützen **Kommunikation** zwischen verschiedenen Beteiligten
    - Skizze zur **Diskussion**
    - **Vorgabe** zur Entwicklung (**VORBILD**)
  - Exakte Notation und Abstraktion ermöglicht Verwendung von Werkzeugen zur **Analyse** von Eigenschaften (des Originals) (**ABBILD**)
    - Größe
    - Komplexität
    - Erreichbarkeit von Graphen
    - .....

## Modelle in Systementwicklung



## 2.1. Unified Modeling Language (UML)

2. Beschreibungs-  
techniken

▶ 2.1. Modellierung

2.2. Objekt-  
orientierung

2.3. Klassen-  
diagramme

- ◆ Seit 1997 De-Facto-Industrie-Standard, der die bisherigen Einzelnotationen integriert (aktuell Version 2.0, [www.uml.org](http://www.uml.org))
- ◆ Vor allem ausgerichtet auf **Objektorientierte Entwicklungsmethoden**
- ◆ definiert
  - Klassen/Objekt/Paketdiagramme
  - Kompositionsstruktur/Komponenten/Verteilungsdiagramme
  - Use/Case/Aktivitätsdiagramme
  - Zustands/Sequenz/Kommunikation/Timingdiagramme

## 2.2. Konzepte der Objektorientierung: Objekte

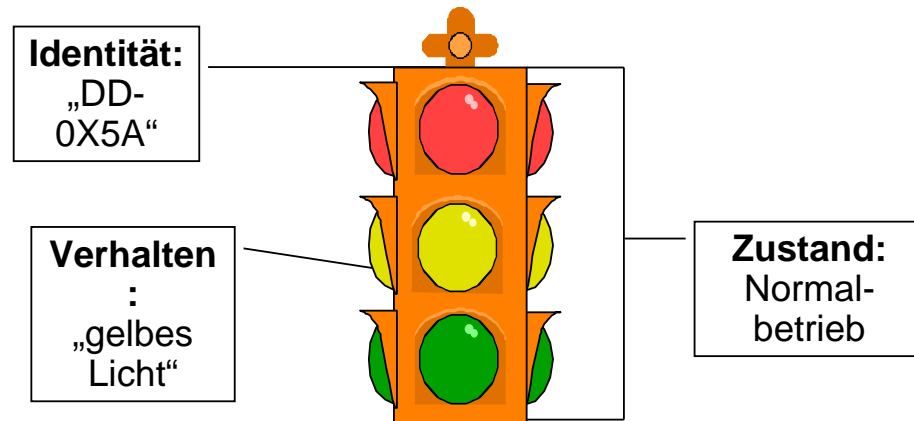
2. Beschreibungs-  
techniken

▶ 2.1. Modellierung

2.2. Objekt-  
orientierung

2.3. Klassen-  
diagramme

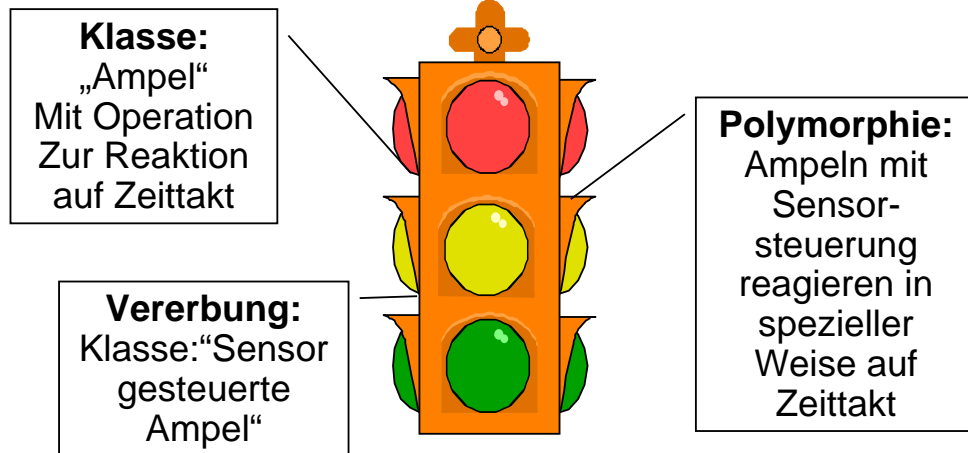
- ◆ Ein System besteht aus vielen Objekten.
- ◆ Ein Objekt hat ein definiertes **Verhalten**
  - Definierte **Methoden**
  - Methode wird beim Empfang einer **Nachricht** ausgeführt.
- ◆ Ein Objekt hat einen inneren **Zustand**
  - Zustand des Objekts ist **Privatsache**.
  - Resultat einer Methode hängt vom **aktuellen Zustand** ab.
- ◆ Ein Objekt hat eine eindeutige **Identität**
  - Identität ist **unabhängig von anderen Eigenschaften**.
  - Es können mehrere verschiedene Objekte mit identischem Verhalten und identischem inneren Zustand im gleichen System existieren.



- ◆ Wie ist das Objekt bezeichnet?
- ◆ Wie verhält es sich zu seiner Umgebung?
- ◆ Welche Informationen sind „Privatsache“ des Objekts?

- ◆ Ein **Objekt** gehört zu einer **Klasse**.
  - Die Klasse schreibt das Verhaltensschema und die innere Struktur ihrer Objekte vor.
- ◆ Klassen besitzen einen ‘Stammbaum’, in der Verhaltensschema und innere Struktur durch **Vererbung** weitergegeben werden.
  - Vererbung bedeutet **Generalisierung** einer Klasse zu einer Oberklasse.
- ◆ **Polymorphie:** Eine Nachricht kann verschiedene Reaktionen auslösen, je nachdem zu welcher Unterklasse einer Oberklasse das empfangende Objekt gehört

## 2.2. Beispiel: Klasse und Objekt

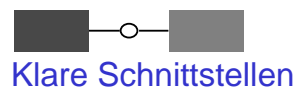


- ◆ Welcher Begriff beschreibt das Objekt?
- ◆ Welche Begriffshierarchie wird verwendet?
- ◆ Wie hängt das Verhalten des Objektes von der Hierarchie ab?

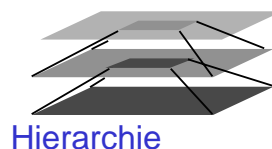
## 2.2. Vorteile der OO



Lokale Kombination von  
Daten und Operationen,  
gekapselter Zustand



Definiertes Objektverhalten,  
Nachrichten zwischen Objekten



Vererbung und Polymorphie  
(Spezialisierung),  
Klassenschichtung



Benutzung vorgefertigter  
Klassenbibliotheken,  
Anpassung durch Spezialisierung  
(mittels Vererbung)

## 2.3. Klassen/Objektdiagramme

2. Beschreibungs-  
techniken

- 2.1. Modellierung
- 2.2. Objekt-orientierung
- 2.3. Klassen-  
diagramme

- ◆ 2.3.1. Klassen (Objekte)
- ◆ 2.3.2. Assoziationen (Aggregation, Komposition)
- ◆ 2.3.3. Attribute
- ◆ 2.3.4. Operationen
- ◆ 2.3.5. Generalisierungsbeziehungen (Vererbung)
- ◆ 2.3.6. Schnittstellen

## 2.3.1. Klasse und Objekt

2. Beschreibungs-  
techniken

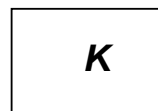
- 2.1. Modellierung
- 2.2. Objekt-orientierung
- 2.3. Klassen-  
diagramme

- ◆ **Definition:** Ein **Objekt** ist ein elementarer Bestandteil des betrachteten Fachgebiets.

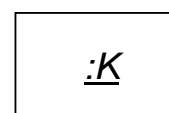
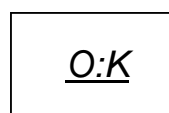
**Definition:** Eine **Klasse** ist eine Beschreibung gleichartiger Objekte.

Jedes Objekt gehört zu (ist **Instanz** von) genau einer Klasse.

- ◆ **Notation:**



Klasse



Objekt

## 2.3.1. Klasse und Objekt

◆ Beispiele:

2. Beschreibungs-  
techniken
- 2.1. Modellierung
  - 2.2. Objekt-  
orientierung
  - 2.3. Klassen-  
diagramme

Ampel

class Ampel {...}

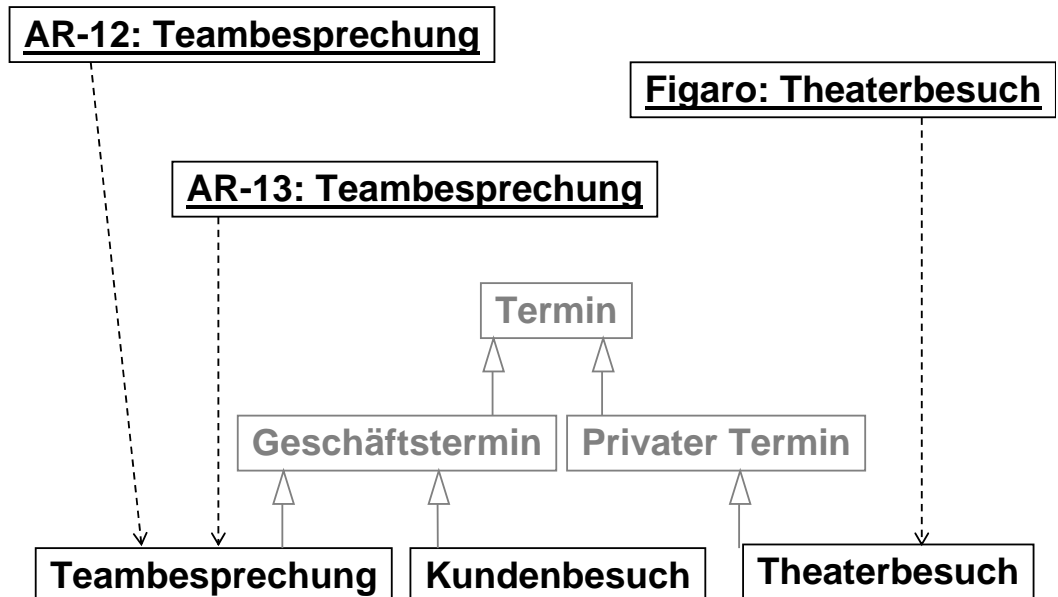
Sensorampel: Ampel

Ampel Sensorampel = ....

: Ampel

## 2.3.1. Beispiel: Termin-Klasse und Termin-Objekte

2. Beschreibungs-  
techniken
- 2.1. Modellierung
  - 2.2. Objekt-  
orientierung
  - 2.3. Klassen-  
diagramme



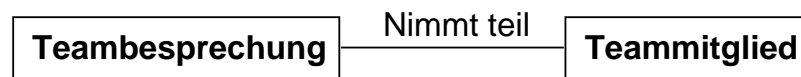
- > Instanz einer Klasse (hier redundante Information)
- > Generalisierung / Vererbung

- ◆ **Definition:** Eine (binäre) **Assoziation** *AS* zwischen zwei Klassen *K1* und *K2* beschreibt, dass die Instanzen der beiden Klassen in einer fachlich wesentlichen Beziehung zueinander stehen.
- ◆ **Semantik:** Für jedes Objekt *O1* der Klasse *K1* gibt es eine individuelle, veränderbare und endliche Menge *AS* von Objekten der Klasse *K2*, mit dem die Assoziation *AS* besteht. Analoges gilt für Objekte von *K2*.

◆ **Notation:**



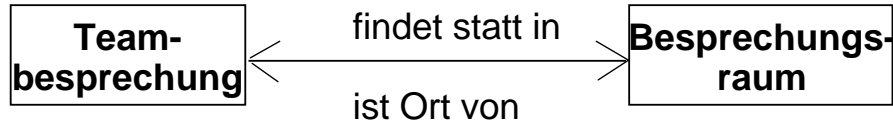
◆ **Beispiele**



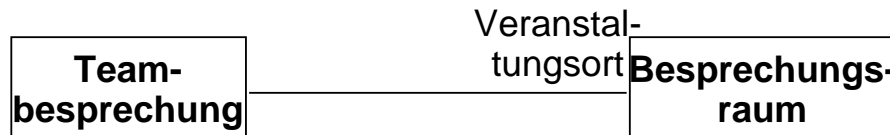
**Assoziationen entsprechen  
Aufrufbeziehungen,  
Attributen,  
Objekterzeugung**

## 2.3.2. Navigationsrichtung und Assoziationsenden

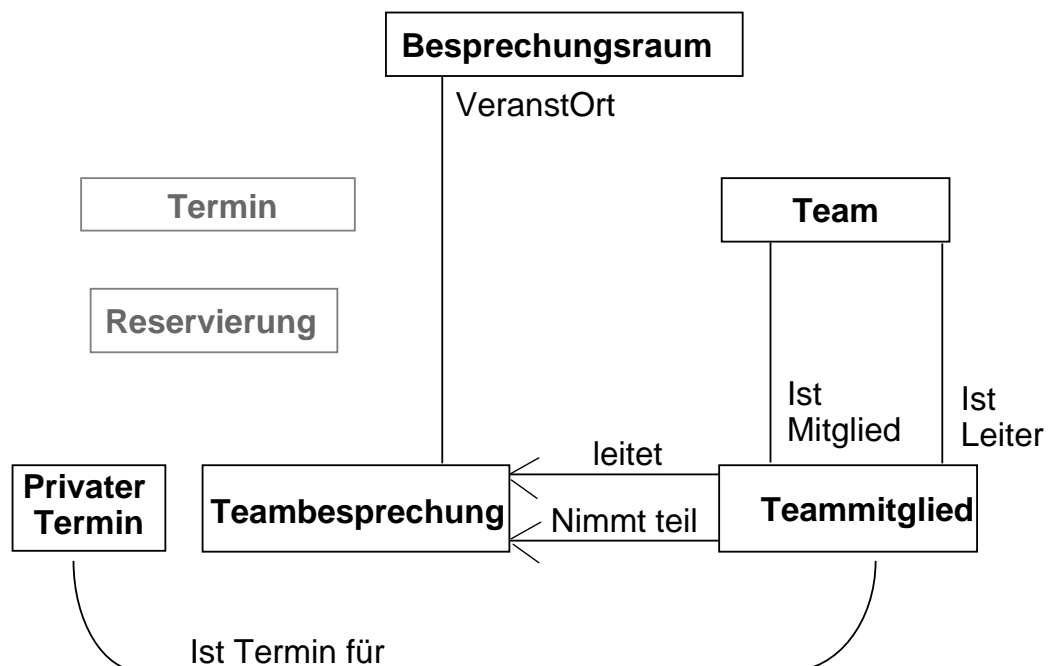
- Für Assoziationsnamen kann die **Navigationsrichtung** angegeben werden (C: Pointer). Es ist möglich, mehrere Namen für eine Assoziation anzugeben. Ein X schließt Navigierbarkeit aus.



- Ein Name für ein **Assoziationsende** bezeichnet die Assoziation (optional) aus der Sicht einer der teilnehmenden Klassen.

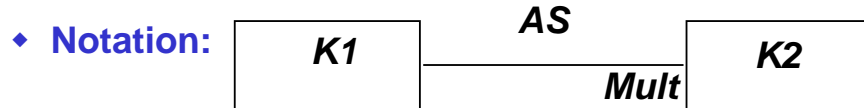


## 2.3.2. Beispiel: Assoziationen



## 2.3.2. Multiplizität bei Assoziationen

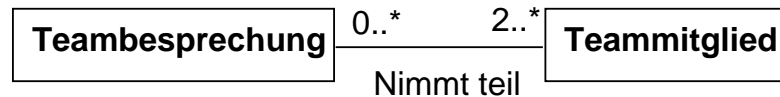
- ♦ **Definition** Die **Multiplizität** einer Klasse  $K1$  in einer Assoziation  $AS$  mit einer Klasse  $K2$  begrenzt die Anzahl der Objekte der Klasse  $K2$ , mit denen ein Objekt von  $K1$  in der Assoziation  $AS$  stehen muss und darf.



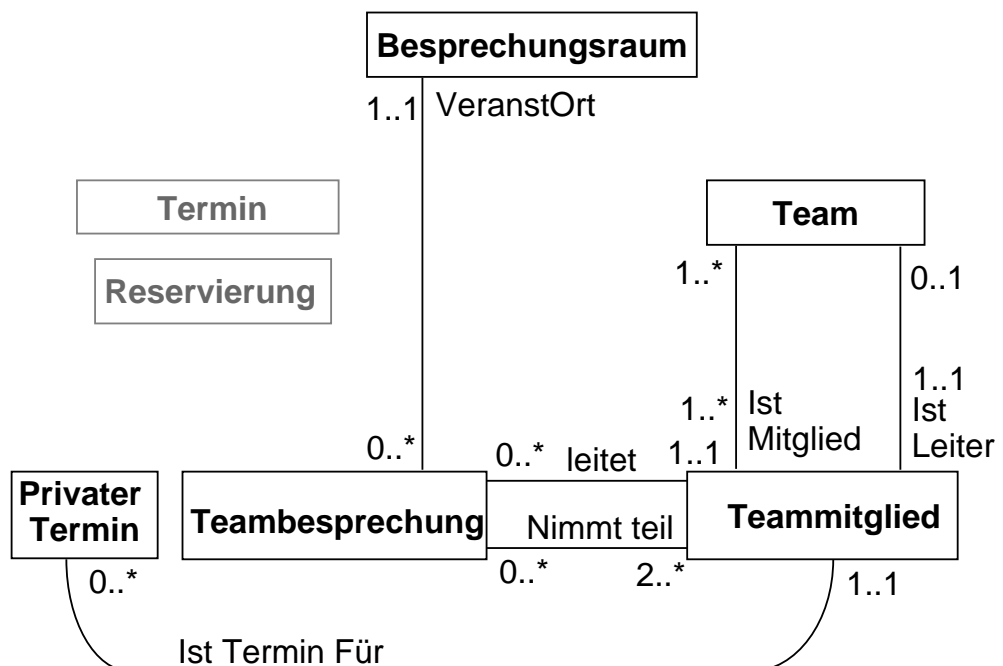
**Multiplizität *Mult*:**

$n..m$  ( $n$  bis  $m$  Objekte der Klasse  $K2$ )  
 Zulässig für  $n$  und  $m$  Zahlenwerte (auch 0)  
 \* (d.h. beliebiger Wert, einschließlich 0)

- ♦ **Beispiel:**

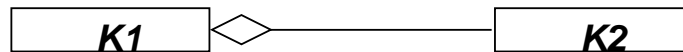


## 2.3.2. Beispiel: Multiplizitäten

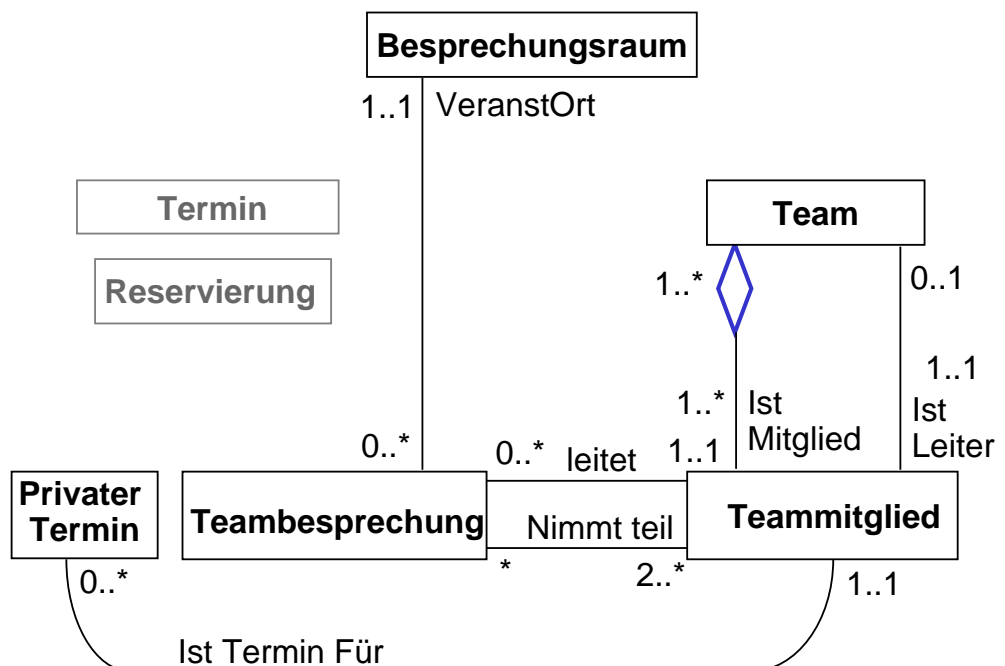
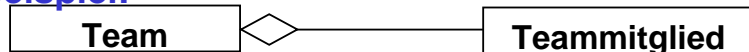


- ◆ **Definition:** Ein Spezialfall der Assoziation ist die **Aggregation**.
- ◆ **Regel:** Wenn die Assoziation den Namen "besteht aus" tragen könnte, handelt es sich um eine Aggregation.
  - Eine Aggregation besteht zwischen einem **Aggregat** und seinen **Teilen**.
  - Die auftretenden Aggregationen bilden auf den Objekten immer eine **transitive, antisymmetrische Relation** (einen gerichteten zyklensfreien Graphen).
  - Mit der Aggregation sind oft **gemeinsame Lebensdauer** der Teile mit dem Aggregat impliziert.

◆ **Notation:**

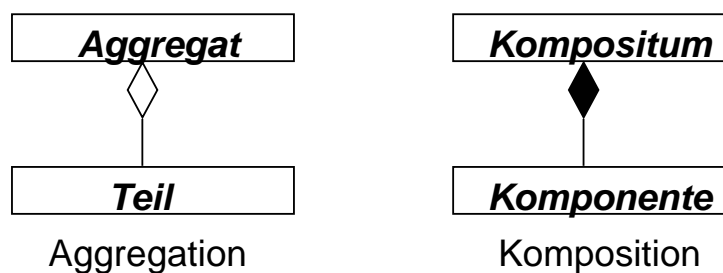


◆ **Beispiel:**



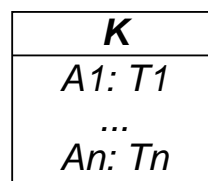
- ◆ **Definition:** Ein Spezialfall der Aggregation ist die **Komposition**. Eine Komposition besteht zwischen einem **Kompositum** und seinen **Komponenten**.
  - Ein Objekt kann Komponente **höchstens eines** Kompositums sein.
  - Das Kompositum hat die **alleinige Verantwortung für Erzeugung und Löschung** seiner Komponenten.
  - Wenn ein Kompositum gelöscht wird, **werden alle seine Komponenten** gelöscht.

◆ **Notation:**



- ◆ **Definition** Ein **Attribut**  $A$  einer Klasse  $K$  beschreibt ein Datenelement, das in jedem Objekt der Klasse vorhanden ist.  
Jedes Objekt der Klasse  $K$  trägt für jedes Attribut  $A$  von  $K$  einen individuellen und veränderbaren Attributwert eines festen Typs  $T$ .

◆ **Notation:**



```
class K {
    T1 A1;
    Public T2 A2;
    Private T3 A3;
    ...
}
```

- ◆ `[Sichtbarkeit] name [: Typ] [Multiplizität] [=Vorgabewert] [ {Eigenschaft = Wert} ]`
- ◆ **Sichtbarkeit des Attributs:**
  - **public (+):** uneingeschränkter Zugriff (JAVA: public)
  - **private (-):** für alle Objekte der Klasse (JAVA: private)
  - **protected (#):** für alle Objekte der Klasse und ihrer Unterklassen (JAVA: protected)
  - **package (~):** für alle Objekte der Klassen eines Packages (JAVA: kein access modifier, d.h. default)
- ◆ **Typisches Beispiel:**
  - `private nachname : String = „paech“`

- ◆ `[Sichtbarkeit] [/] name [: Typ] [Multiplizität] [=Vorgabewert] [ {Eigenschaftswert} ]`
- ◆ Evtl. zusätzlich **/:** **abgeleitetes Attribut** (muss nicht gespeichert werden)
  - `Public / alter : int`
- ◆ **Typangabe:**
  - Evtl. zusätzlich Multiplizität n..m: Mengenwertiges Attribut
    - `Protected numbers : int[1..4]`
- ◆ **Eigenschaft:**
  - **readonly:** darf nicht verändert werden (**JAVA: final**)
  - **redefines:** z.B. `geburtsname {redefines name}`
  - **ordered, bag, sequence:** schränkt Belegungen für mengenwertige Attribute ein



◆ Falsch oder richtig

2.Beschreibungstechniken

2.1. Modellierung

2.2. Objektorientierung

2.3. Klassendiagramme

- String **X**
- public adressen : String[1..\*] 
- # bruder : Person 
- public / int **X**
- numbers : int[1..\*] ordered **X**

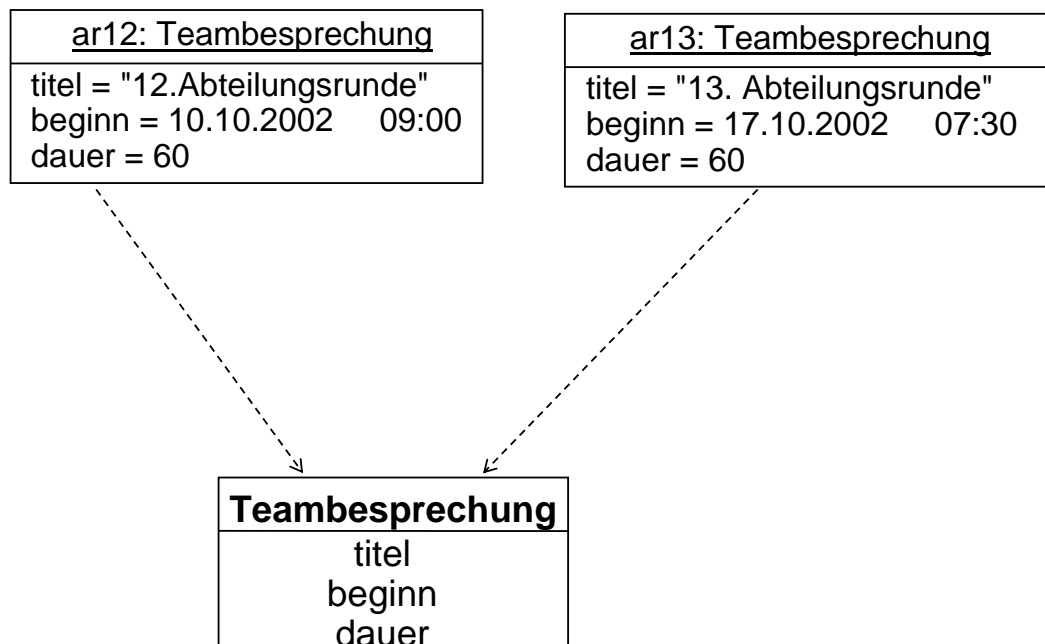
## 2.3.3. Beispiel: Attributwerte von Objekten

2.Beschreibungstechniken

2.1. Modellierung

2.2. Objektorientierung

2.3. Klassendiagramme

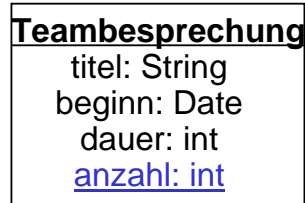


## 2.3.3. Klassenattribute

- ◆ **Definition:** Ein **Klassenattribut**  $A$  beschreibt ein Datenelement, das genau einen Wert für die gesamte Klasse annehmen kann.  
(Gewöhnliche Attribute heißen auch **Instanzattribute**, weil sie für jede Instanz individuelle Werte annehmen.)
- ◆ **Notation:** Unterstreichung

$A : Typ$

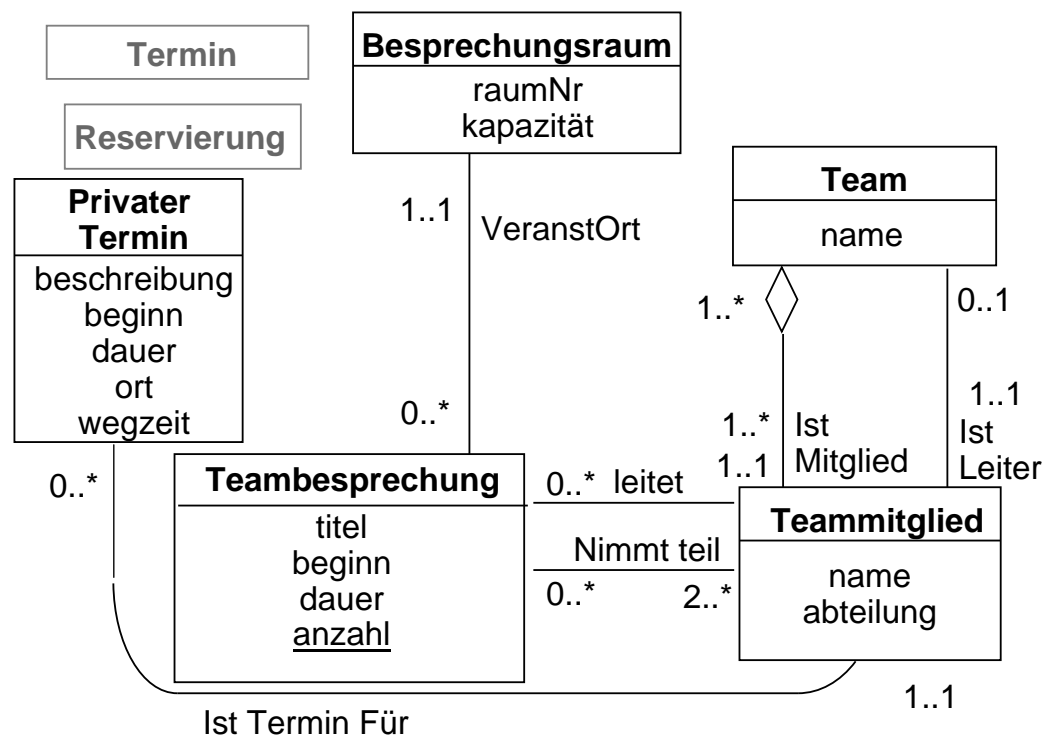
- ◆ **Beispiel:**



static int anzahl

- ◆ **Aber: Klassenattribute verursachen viel Test- und Wartungsprobleme. Deshalb soweit wie möglich vermeiden!**

## 2.3.3. Beispiel: Attribute



- ◆ **Definition** Eine **Operation** einer Klasse  $K$  ist die Beschreibung einer Aufgabe, die jede Instanz der Klasse  $K$  ausführen kann. In der Beschreibung der Klasse wird der Name der Operation angegeben.

- ◆ **Notation:**

<b>Klasse</b>
$A_1$
...
$A_n$
<b>Operation_1</b>
...
<b>Operation_m</b>





- **Beispiel:**

<b>Teambesprechung</b>
titel
beginn
dauer
raumFestlegen()
einladen()
absagen()

Zur besseren Unterscheidung von Attributnamen werden meist **Klammern hinter Operationsnamen** gesetzt, auch wenn über die Parameterliste noch keine Aussagen gemacht werden sollen.

- ◆ [Sichtbarkeit] *operationsname* ([  
[Übergaberichtung] *parameter* [: ParamTyp]  
[Multiplizität] [=DefWert], ...]) [: ResTyp ]
- ◆ DefWert legt einen **Default-Parameterwert** fest, der bei Weglassen des Parameters im Aufruf gilt.
- ◆ **Beispiele (Klasse Teambesprechung):**  
public raumFestlegen (wunschRaum:  
Besprechungsraum): boolean  
  
absagen (grund: String);

- ◆ [Sichtbarkeit] *operationsname* ([**Übergaberichtung**] *parameter* [: ParamTyp] [Multiplizität] [=DefWert], ...) [: ResTyp]
- ◆ Übergaberichtung
  - **In**: Parameter wird nur ausgelesen
  - **Out**: Inhalt des Parameters wird nicht gelesen, nur verändert
  - **Inout**: Inhalt des Parameters wird gelesen und dann verändert
  - **Return**: Rückgabeparameter (*benannter Rückgabetyt*)

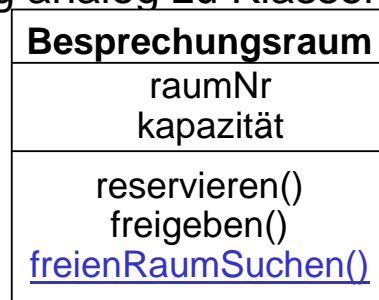
- ◆ Falsch oder richtig
  - +add(summand1, summand2) : int 
  - add ( in*X*i, int j)
  - Clear (foo) : vo*X*i
  - mult (summand : Matrix[2..\*]{ordered}) : Matrix 
  - setLstKl (lohnsteuerklasse : int = 1) 
  - sub (in minuend : double, in subtrahend : double, return resultat : double) 

## 2.3.4. Klassenoperation

- Definition: Eine **Klassenoperation** A einer Klasse K ist die Beschreibung einer Aufgabe, die nur unter Kenntnis der aktuellen Gesamtheit der Instanzen der Klasse ausgeführt werden kann. Gewöhnliche Operationen heißen auch **Instanzoperationen**.

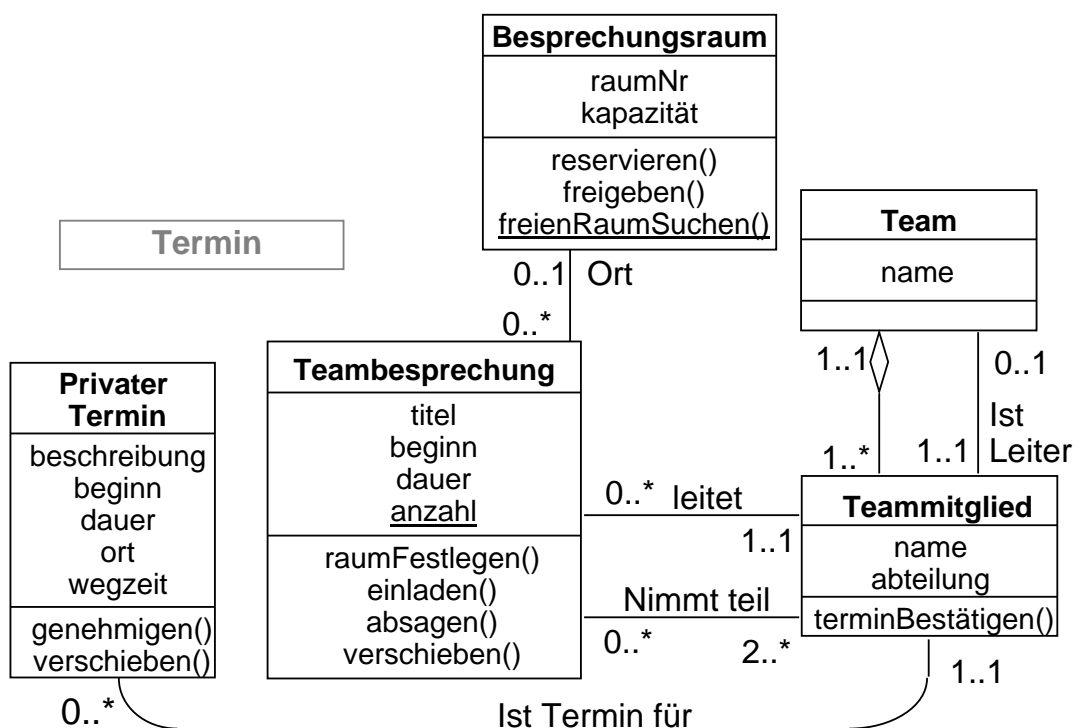
- Notation:  
Unterstreichung analog zu Klassenattributen.

- Beispiel:



**static void  
freienRaumSuchen**

## 2.3.4. Beispiel: Operationen

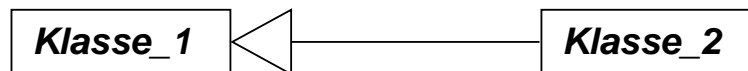


## 2.3.5. Vererbung (Generalisierung)

- ◆ Definition: Eine **Vererbungsbeziehung** von einer Klasse K1 zu einer Klasse K2 ist eine Beschreibung der Tatsache, dass alle Objekte der Klasse K2 zusätzlich zu den in der Klasse K2 beschriebenen Eigenschaften auch alle Eigenschaften der Klasse K1 haben.
- ◆ Eigenschaften sind hier
  - die Liste der Attribute
  - die Teilnahme an Assoziationen, Aggregationen und Kompositionen
  - die Liste der Operationen.

## 2.3.5. Vererbung (Generalisierung)

- ◆ Notation:



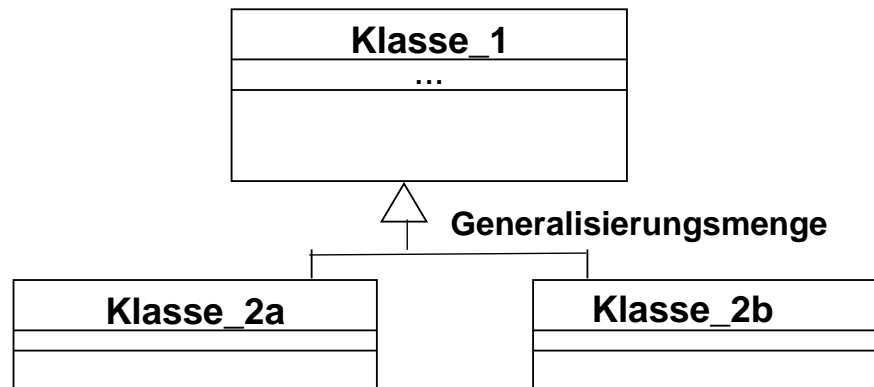
```

class Klasse_1 {..}
class Klasse_2 extends Klasse_1
  
```

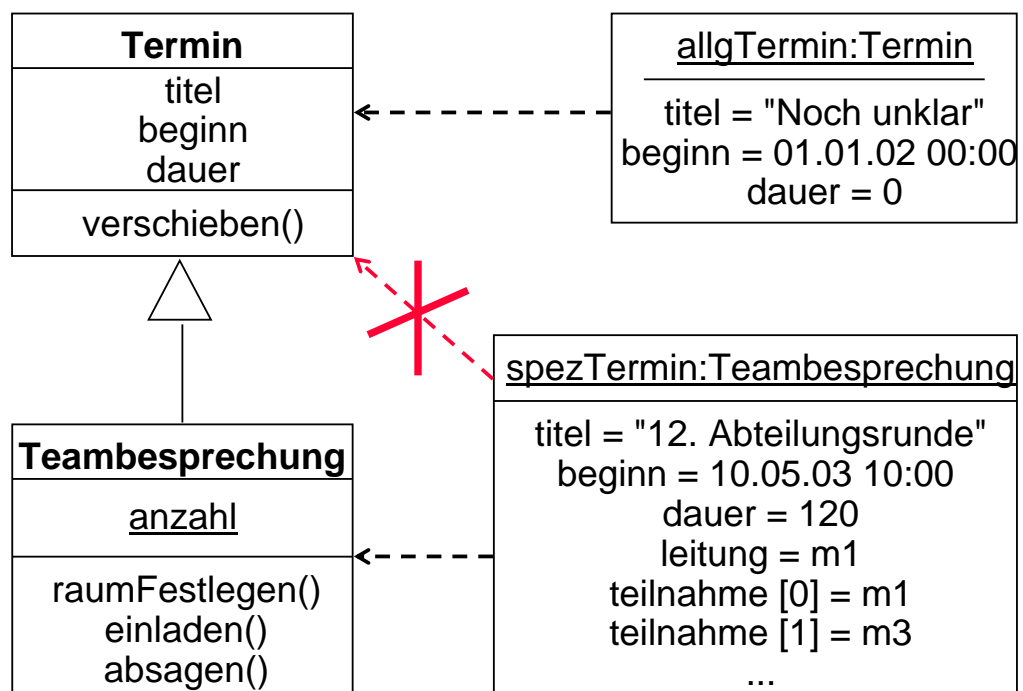
- ◆ Umgangssprachlich: Jede Klasse\_2 ist ein Klasse\_1. Klasse\_2 ist **Spezialfall** von Klasse\_1.

## 2.3.5. Generalisierungsmenge

- Generalisierungseigenschaften beschreiben Zerlegung der Oberklassenobjektmenge durch die Unterklassenobjektmengen:  
*complete, incomplete, disjoint, overlapping*



## 2.3.5. Vererbung und Instanzen



## 2.3.5. Abstrakte und konkrete Klassen

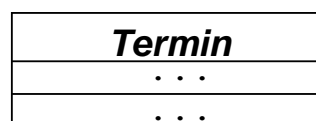
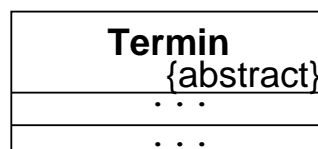
- ◆ Definition: Eine Klasse kann als **abstrakt** deklariert werden. In diesem Fall ist es nicht zulässig, Instanzen der Klasse zu bilden.
- ◆ Abstrakte Klassen dienen als "**Schema**" in der Vererbung und als Gruppierungsmittel, zum Beispiel für gemeinsame Funktionalität.
- ◆ Eine Klasse, von der Instanzen gebildet werden können, heißt **konkret**.

## 2.3.5. Abstrakte und konkrete Klassen

- ◆ Notation:

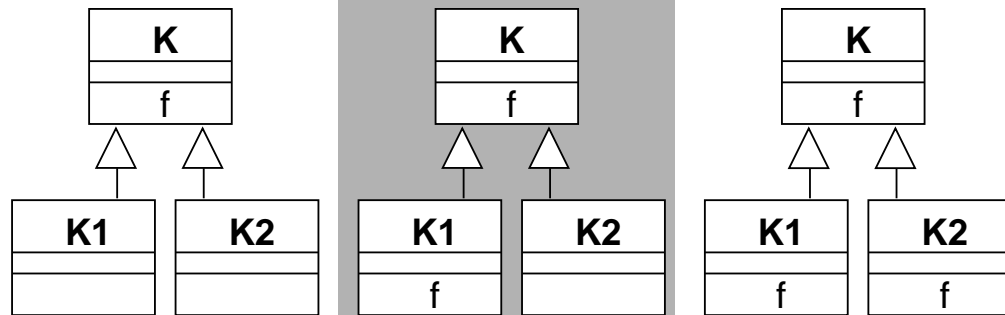


- ◆ Beispiel



abstract class Termin {..}

## 2.3.5. Redefinieren von Operationen



Das in K definierte Verhalten f gilt für alle Objekte der Klassen K, K1 und K2.

Das in K definierte Verhalten f gilt nur für die Objekte der Klassen K und K2.  
K1 definiert ein anderes Verhalten (*Redefinition, override*).

Das in K definierte Verhalten f gilt nur für die Objekte der Klasse K (kann in K1 und K2 bei der Neudefinition von f verwendet werden, sonst nutzlos).

## 2.3.5. Abstrakte und konkrete Operationen

- ◆ **Definition** Eine Operation *OP* kann als **abstrakt** deklariert werden, wenn sie in einer abstrakten Oberklasse *K* definiert ist. In diesem Fall legt *K* kein Verhalten für *OP* fest. Das Verhalten von *OP* muss dann in den Unterklassen von *K* definiert werden.

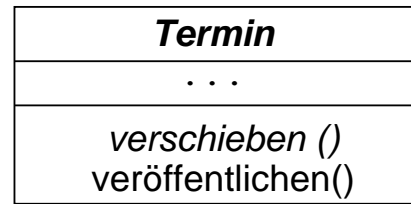
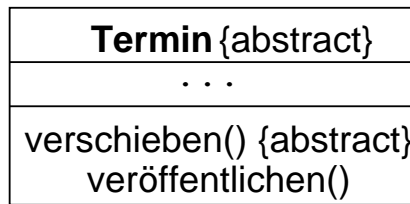
- ◆ **Notation:**



## 2.3.5. Abstrakte und konkrete Operationen

### ◆ Beispiel

2. Beschreibungs-  
techniken
- 2.1. Modellierung
  - 2.2. Objekt-  
orientierung
  - 2.3. Klassen-  
diagramme

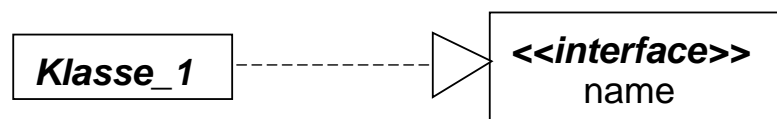
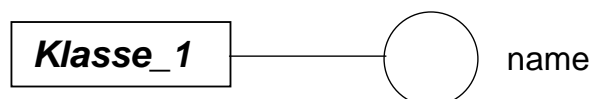
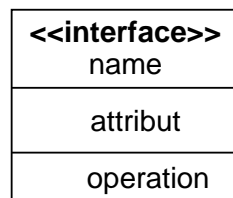


```
abstract class termin{
..
    abstract void verschieben()
}
```

## 2.3.6. Schnittstelle (provided)

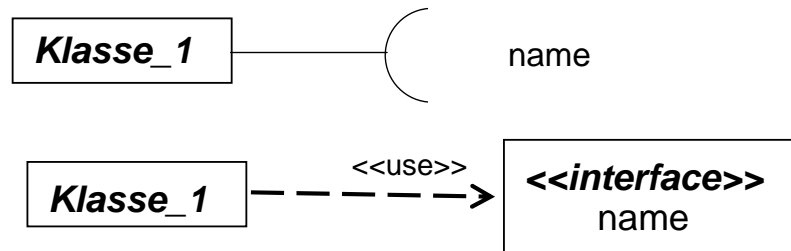
- ### ◆ Festlegung, dass eine Klasse eine bestimmte Mengen von Attribute und Operationen umsetzt

2. Beschreibungs-  
techniken
- 2.1. Modellierung
  - 2.2. Objekt-  
orientierung
  - 2.3. Klassen-  
diagramme

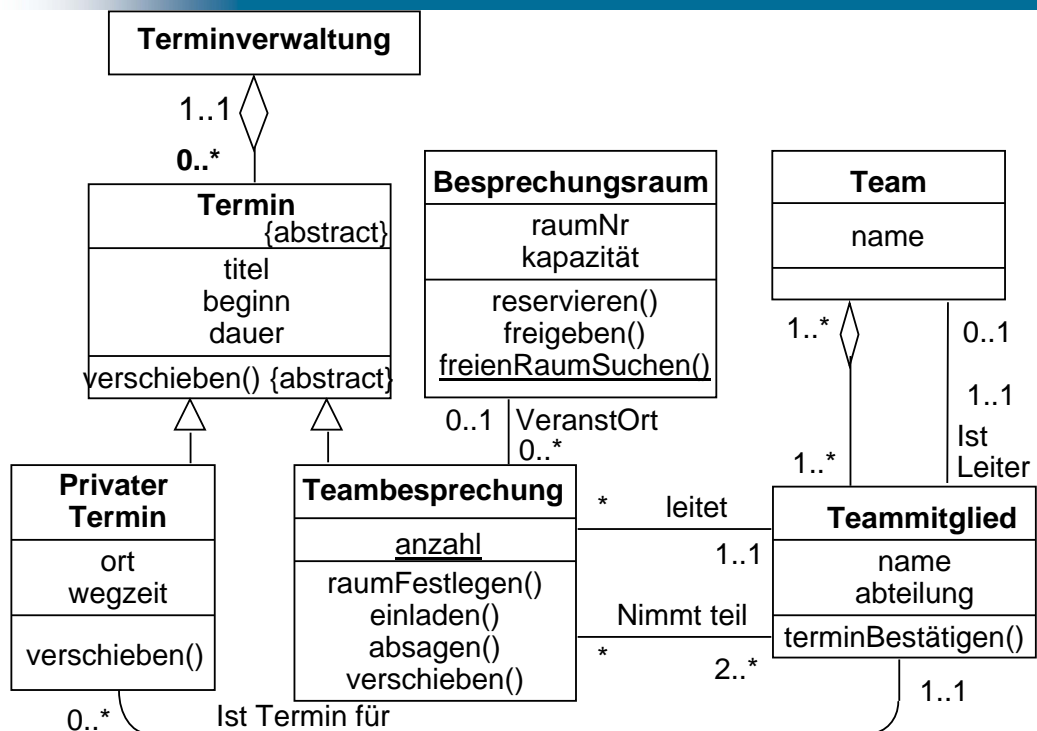


## 2.3.6. Schnittstellen (requested)

- ◆ Festlegung, dass eine Klasse eine andere Klasse mit einer bestimmten Schnittstelle benötigt.



## 2.3.5. Beispiel: Klassendiagramm



## 2.1-2.3. Zusammenfassung

2. Beschreibungs-  
techniken

2.1. Modellierung

2.2. Objekt-  
orientierung

2.3. Klassen-  
diagramme

- ◆ Modellierung unterstützt **Kommunikation und Verständnis** durch Abstraktion
- ◆ Objektorientierung unterstützt **Kapselung** (Objekte kapseln Datenzustand und Verhalten)
- ◆ Klassendiagramme beschreiben **komplexe Strukturen von Objekten**

## 2. Literatur

2. Beschreibungs-  
techniken

2.1. Modellierung

2.2. Objekt-  
orientierung

2.3. Klassen-  
diagramme

- ◆ M. Jeckle, Ch. Rupp, J. Hahn, B. Zengler, S. Queins, „UML 2 glasklar“, Hanser Verlag 2004