

Kapitel 2.1 - Vertiefung der Konzepte der Objektorientierung und UML Klassendiagramme

SWT I – Sommersemester 2010

Walter F. Tichy, Andreas Höfer, Korbinian Molitorisz

IPD Tichy, Fakultät für Informatik



Definition von Objekt und Klasse

■ Wikipedia (deutsch)

- Ein Objekt bezeichnet [...] ein **Exemplar** oder eine **Instanz** einer bestimmten Klasse. [...]

(http://de.wikipedia.org/wiki/Objekt_%28Programmierung%29)

- Klasse ist [...] ein abstrakter Oberbegriff für die Beschreibung der gemeinsamen **Struktur** und des gemeinsamen **Verhaltens** von Objekten [...]

(http://de.wikipedia.org/wiki/Klasse_%28objektorientierte_Programmierung%29)

Definition einer Grundmenge \mathcal{G}

- Def. \mathcal{G} : Die Menge G ist die Vereinigung aus allem vergangenen, gegenwärtigen und zukünftigen Substanziellem und Konzeptuellem.
 - Beispielsweise enthält \mathcal{G} :
 - Personen, Prof. Tichy, Andreas Höfer, Tom Gelhausen, ... (substanziell)
 - Luft (substanziell)
 - Mein Lebensversicherungsvertrag (konzeptuell)
 - Demokratie (konzeptuell)
 - Fußball WM 2010 (Ereignis → konzeptuell)
 - Schwimmen (Tätigkeit → konzeptuell)
 - Schwimmen (Fähigkeit → konzeptuell)
 - Blau (Farbe/Eigenschaft → konzeptuell)
 - etc.

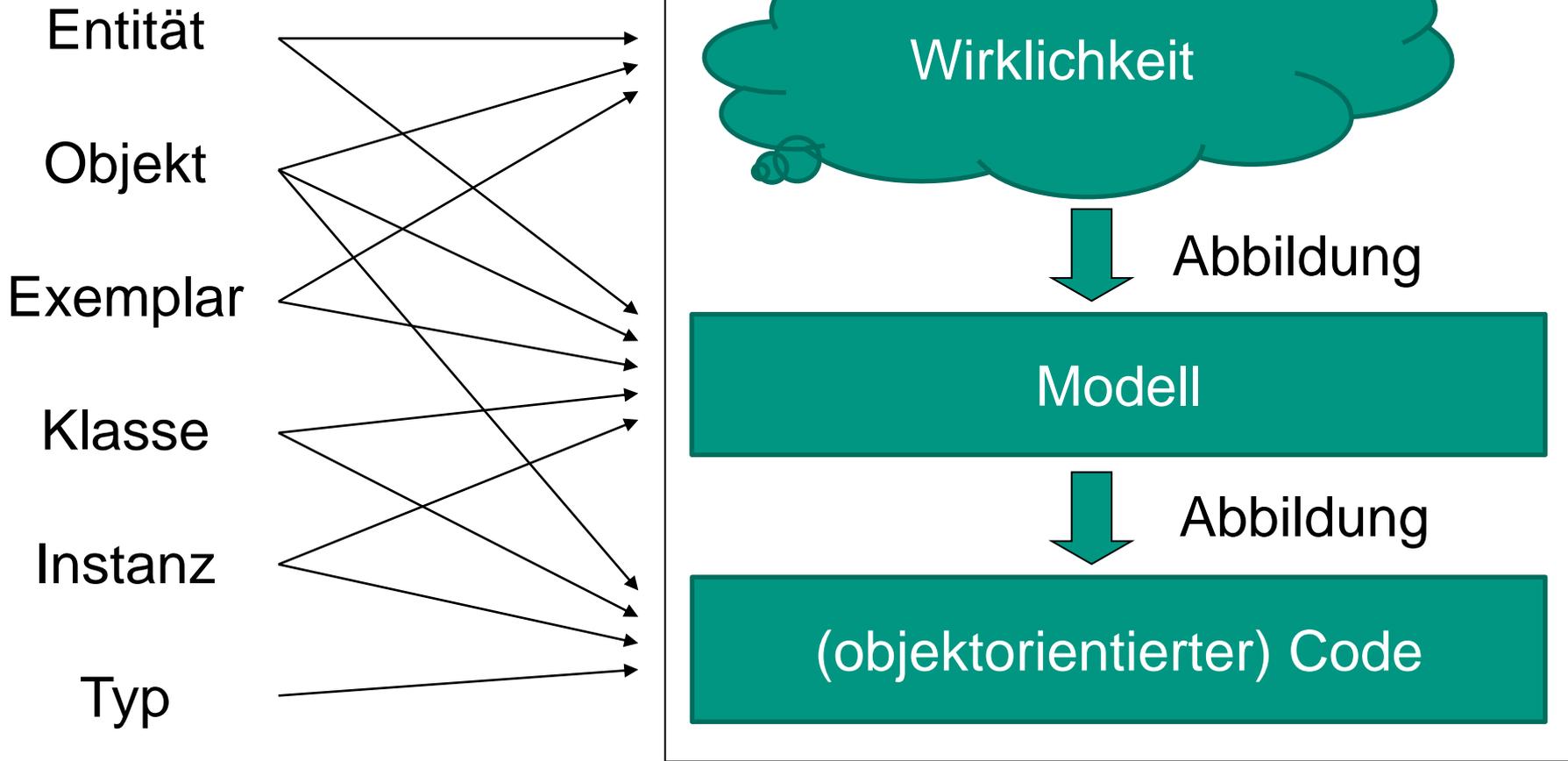
Objekt

- Def. **Objekt** (engl. *object*): Ein für min. ein Individuum erkennbares, eindeutig von anderen Objekten unterscheidbares, also *bestimmbares* Element aus der Menge \mathcal{O} .
- Def. Ω : Die Menge aller Objekte.
- Übung: Denken Sie über folgende alternative Definitionen nach:
 - Alles was man mit einem Nomen oder Namen bezeichnen kann.
 - Charles S. Peirce: „By an object, I mean anything that we can think, i.e. anything we can talk about.“

Klasse und Exemplar

- Def. **Klasse** (engl. *class*): Eine (prinzipiell willkürliche) Kategorie über der Menge aller Objekte Ω .
 - Hinweise
 - In der Regel wird man der Kategorisierung irgend eine Art von „Gleichartigkeit“ der darin enthaltenen Objekte zugrunde legen.
 - Die Kategorie kann auch leer sein: „Klasse“ bezeichnet die grundsätzliche Idee/das Konzept der Dinge und existiert unabhängig davon, ob eine Ausprägung existiert, oder nicht.
- Def. **Exemplar** (engl. *exemplar*): Ein konkretes Element aus einer bestimmten Klasse.
 - Hinweise
 - Ein Exemplar gehört zu (min.) einer bestimmten Klasse.
 - Man spricht auch oft von „Ausprägung“ oder „Instanz“.

Domänen in denen die Begriffe üblicherweise verwendet werden



Einordnung der Begriffe

■ Hinweise

- Der Begriff „Objekt“ wird vor allem in der **Analysephase** häufig für den Repräsentant einer Klasse (und damit als Stellvertreter für diese) verwendet; vergl. Moduloarithmetik:

$$\mathbb{Z}_4 = \{\tilde{0}, \tilde{1}, \tilde{2}, \tilde{3}\}$$

- Verwende Begriffe „Klasse“ und „Instanz“, wenn die Struktur der Klasse identifiziert und festgelegt ist und ein Element aus genau dieser Klasse gemeint ist – also auf der **Realisierungsebene**.

Attribute

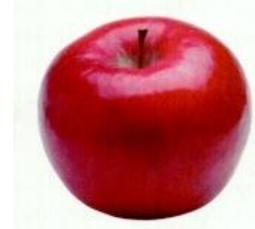
- Def. **Attribut** (engl. *attribute*): Eine für alle Exemplare einer Klasse definierte und vorhandene Eigenschaft, die
 - für jedes einzelne Exemplar unabhängig von den anderen angegeben werden kann und
 - einen klar definierten Wert
 - aus einer bestimmten, für alle gleichen Domäne hat.
 - Notation: **Attributname: Typ [=wert];**



Exemplar 1
Farbe: rot;
Angebissen: ja;



Exemplar 2
Farbe: grün;
Angebissen: nein;



Exemplar 3
Farbe: rot;
Angebissen: nein;



Exemplar 4
Farbe: grün;
Angebissen: ja;

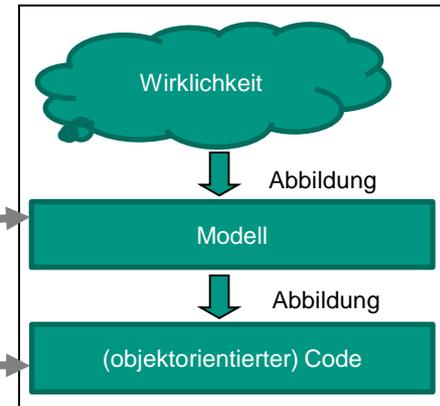
Domäne: Farben

Domäne: ja/nein

Hinweis für Java-Programmierer

■ Unterscheide **Attribut** und **Instanzvariable** eines Objektes!

- Häufig 1:1-Abbildung von Attributen auf eine Instanzvariable möglich



- Die Umkehrung gilt i. A. nicht, da auch der **Zustand** und die **Assoziationen** eines Objektes in den Instanzvariablen gespeichert werden müssen (siehe später)
- Attribute könnten zusätzliche **Einschränkungen** oder **Zusicherungen** enthalten, die sich nicht alleine durch eine Instanzvariable realisieren lassen, sondern noch zusätzlichen Code benötigen (→ Object Constraint Language, siehe später)

Objektidentität

- Def. **Objektidentität** (engl. *object identity*): Die Existenz eines Objektes ist unabhängig von seinen Attributwerten. Zwei Objekte sind auch dann unterscheidbar, wenn sie die gleichen Attributwerte besitzen.
 - Hinweise
 - Die Objektidentität ist eigentlich schon durch unsere Definition von Objekt und Attribut gegeben
 - Zwei (oder mehr) Instanzen können gleich sein, ohne das selbe sein zu müssen → Gleichheit?



≠



≠



Farbe: rot
Angebissen: nein

=

Farbe: rot
Angebissen: nein

=

Farbe: rot
Angebissen: nein

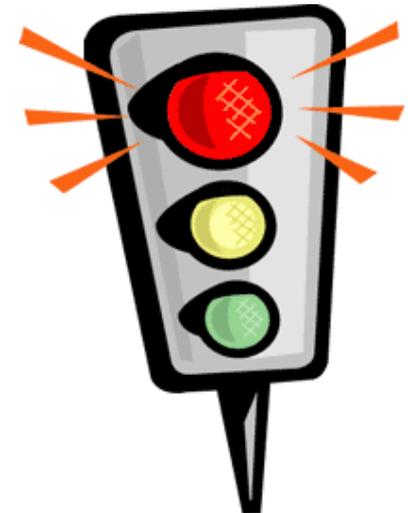
Vergleich zweier Objekte

- Def. **Gleichheit x -ter Stufe** (engl. *equality of order x*)
 - **Gleichheit 0. Stufe:** es handelt sich um das selbe Objekt, die Objekte sind identisch
 - **Gleichheit 1. Stufe:** es handelt sich entweder um das selbe Objekt oder zwei verschiedene Objekte, die aber *in allen Attributen* identische Werte* besitzen (Gleichheit 0. Stufe oder paarweise Gleichheit 0. Stufe in allen Attributen*)
 - **Gleichheit 2. Stufe:** es handelt sich entweder um das selbe Objekt oder es handelt sich um zwei verschiedene Objekte, die aber *in allen Attributen* gleiche oder identische Werte* besitzen (Gleichheit 1. Stufe oder paarweise Gleichheit 0. oder 1. Stufe in allen Attributen*)
 - **Gleichheit 3. Stufe:** Gleichheit 2. Stufe oder paarweise Gleichheit 0., 1. oder 2. Stufe in allen Attributen*
 - etc.

* natürlich müssen auch Assoziationen und Zustand (siehe später) Berücksichtigung finden

Definition des Zustands eines Objektes

- Häufig wird der Zustand als „Verkettung der Werte aller Attribute“ definiert, oder einfach die Folge aller Bits, die das Objekt speichert.
- Diese Definition ist jedoch problematisch: Ist z.B. der Wert eines Attributes „Betriebsstunden“ wirklich wichtig für den Zustand eines Ampel-Objektes?



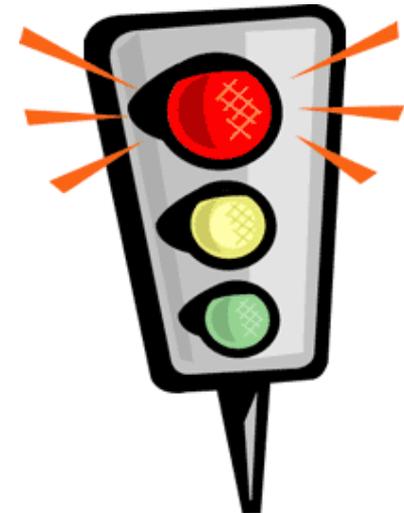
Zustand eines Objektes

- Def. **Zustand** (engl. *state*): Solange sich ein Objekt in einem Zustand befindet, reagiert es im gleichen (Aufruf-/Verwendungs-) Kontext immer **gleich** auf seine Umwelt. Ändert sich der Zustand, reagiert das Objekt in mindestens einem Kontext **anders** als zuvor. (Außensicht)

Übung: Vergleiche diese Definition mit:

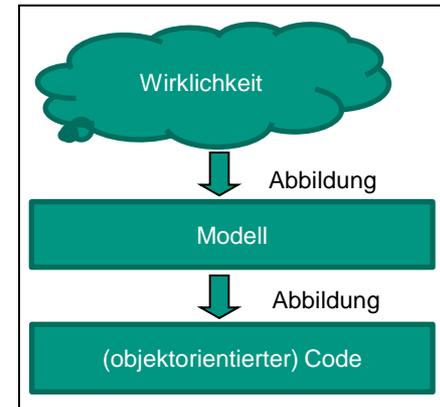
„**state** - A condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event.“

OMG, UML 2.0 Infrastructure Specification, Ch. 4, Terms and Definitions



Hinweis für Java-Programmierer

- Der Zustand eines Objektes muss in den **Instanzvariablen** gespeichert werden (so wie die Werte der Attribute).
- Hierfür können
 - entweder dedizierte Variablen verwendet werden (**expliziter** Zustand)
 - oder der Zustand kann aus dem Inhalt anderer Instanzvariablen *abgeleitet* werden (**impliziter** Zustand)
 - Beispiel: Wahlberechtigung aus Geburtsdatum

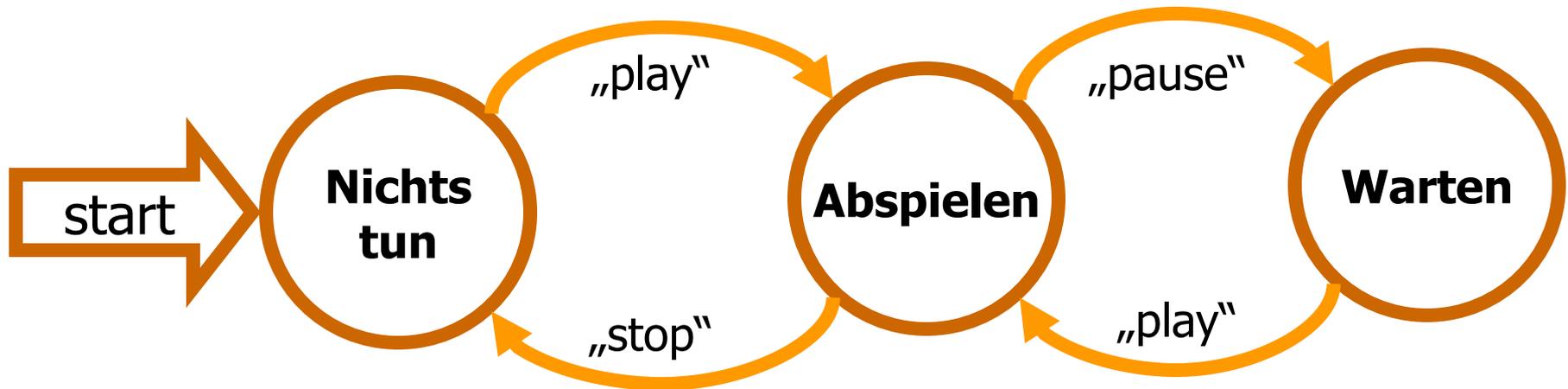


Kapselungsprinzip

- Def. **Kapselungsprinzip** (engl. *encapsulation*): Der Zustand ist zwar nach außen sichtbar, er wird aber im Inneren des Objektes verwaltet (und also nur kontrolliert geändert).
 - Hinweis: Die Änderung eines Attributwertes könnte die Änderung des Zustands bedingen.
 - Beispiel: Wenn sich der Wert des Attributes „Restzeit“ eines Objektes „Eieruhr“ auf 0 ändert, wechselt die „Eieruhr“ in den Zustand „Klingeln“.

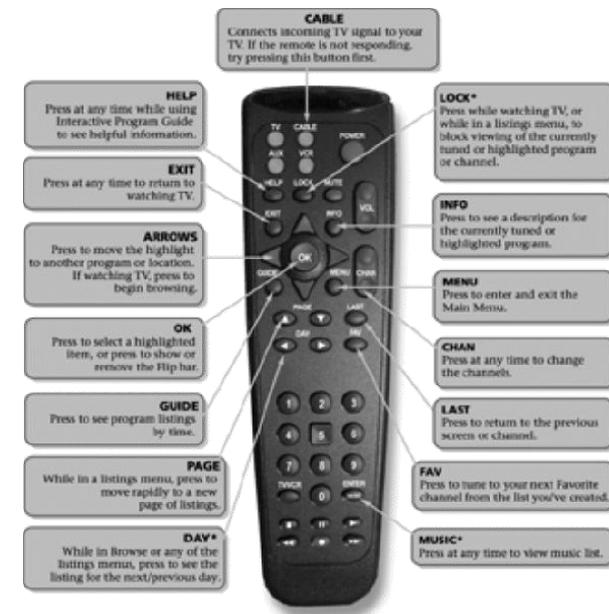
Zustandsänderung

- Beispiel **Kapselungsprinzip**:
Niemand käme auf die Idee, im Inneren eines Videorekorders rumzufummeln, um ihn in den Zustand „Abspielen“ zu versetzen



Nachricht an ein Objekt

- „Nachrichtenaustausch“ – man möchte betonen, dass ein bestimmtes Objekt (= Nachrichteneempfänger) aufgefordert wird, einen Zustandsübergang vorzunehmen
- Daher: **Nachrichtenaustausch = Methodenaufruf** bei einem bestimmten Objekt
- Beispiel: Mit dem Play-Knopf sendet man die Nachricht „*Wechsle in den Zustand ‚Wiedergabe‘*“ an das Gerät, zu dem der Knopf gehört

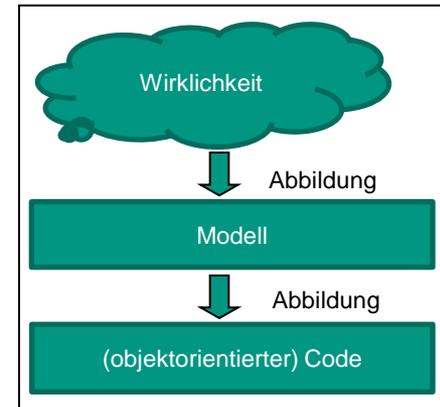


Methoden

- Methoden können den Zustand eines Objektes verändern
 - Die verfügbaren Methoden definieren die zulässigen Botschaften, die man einem Objekt senden kann (**Außenansicht**)
 - Problem: Wann darf ich welche Botschaft an ein Objekt senden?
 - Antwort: Das spezifiziert man mit einem **Zustandsübergangdiagramm** (siehe später)

Hinweis für Java-Programmierer

- Das Zustandsübergangdiagramm definiert einen endlichen Automaten.
- Ist der Automat **vollständig**?
 - Was passiert, wenn ich im Zustand X die Nachricht Y sende, was da eigentlich gar nicht vorgesehen ist?
 - Möglichkeiten:
 - Ausnahmeobjekt erzeugen
 - Java.lang.Error-Instanz erzeugen
 - „garbage in – garbage out“
 - Nichts tun
 - etc.



Methodensignatur

- Def. **Methodensignatur**: Besteht aus
 - der Methodenname und
 - dem Rückgabetyp und
 - der Parameterliste
 - Die Parameter sind die „Nutzdaten“ der Nachricht
 - Das Empfängerobjekt kann auch als „nullter Parameter“ angesehen werden, als „Adresse“ der Nachricht
- Notation:
Methodenname(Parameterliste) : Rückgabety;p;
- Notation (Parameterliste):
Attributname : Typ [, Attributname : Typ]*

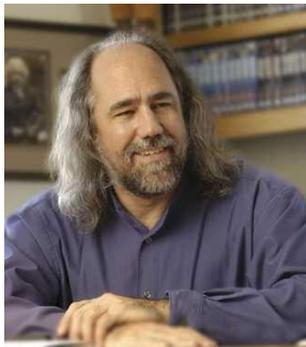
UML Klassendiagramme

IPD Tichy, Fakultät für Informatik



Was ist UML?

- UML ist die Vereinigung dreier Notation aus der objektorientierten Modellierung:
 - Booch (Grady Booch)
 - OOSE (Ivar Jacobson)
 - OMT (James Rumbaugh)



Grady Booch

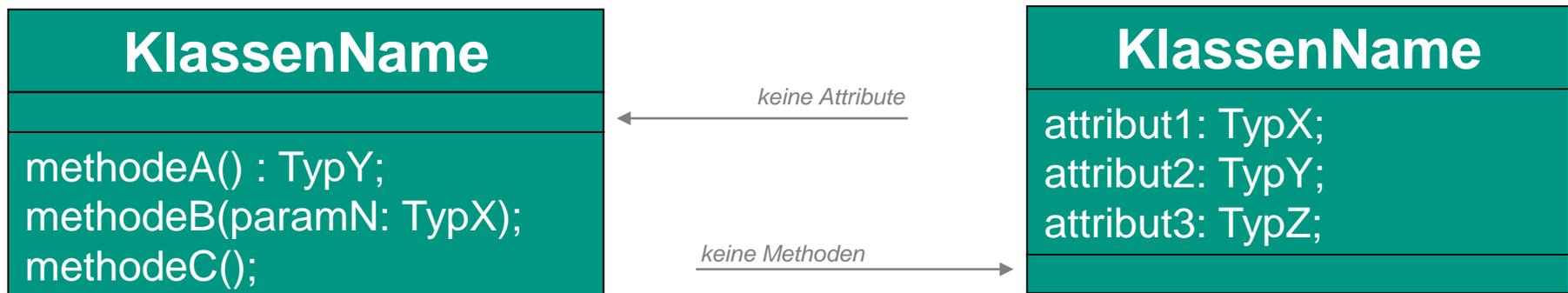
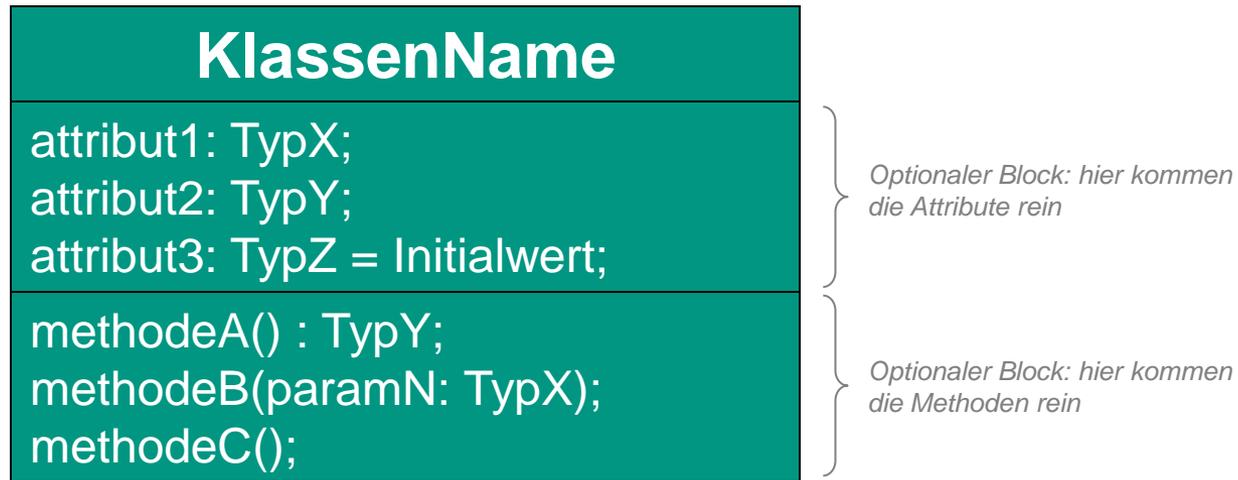


Ivar Jacobson

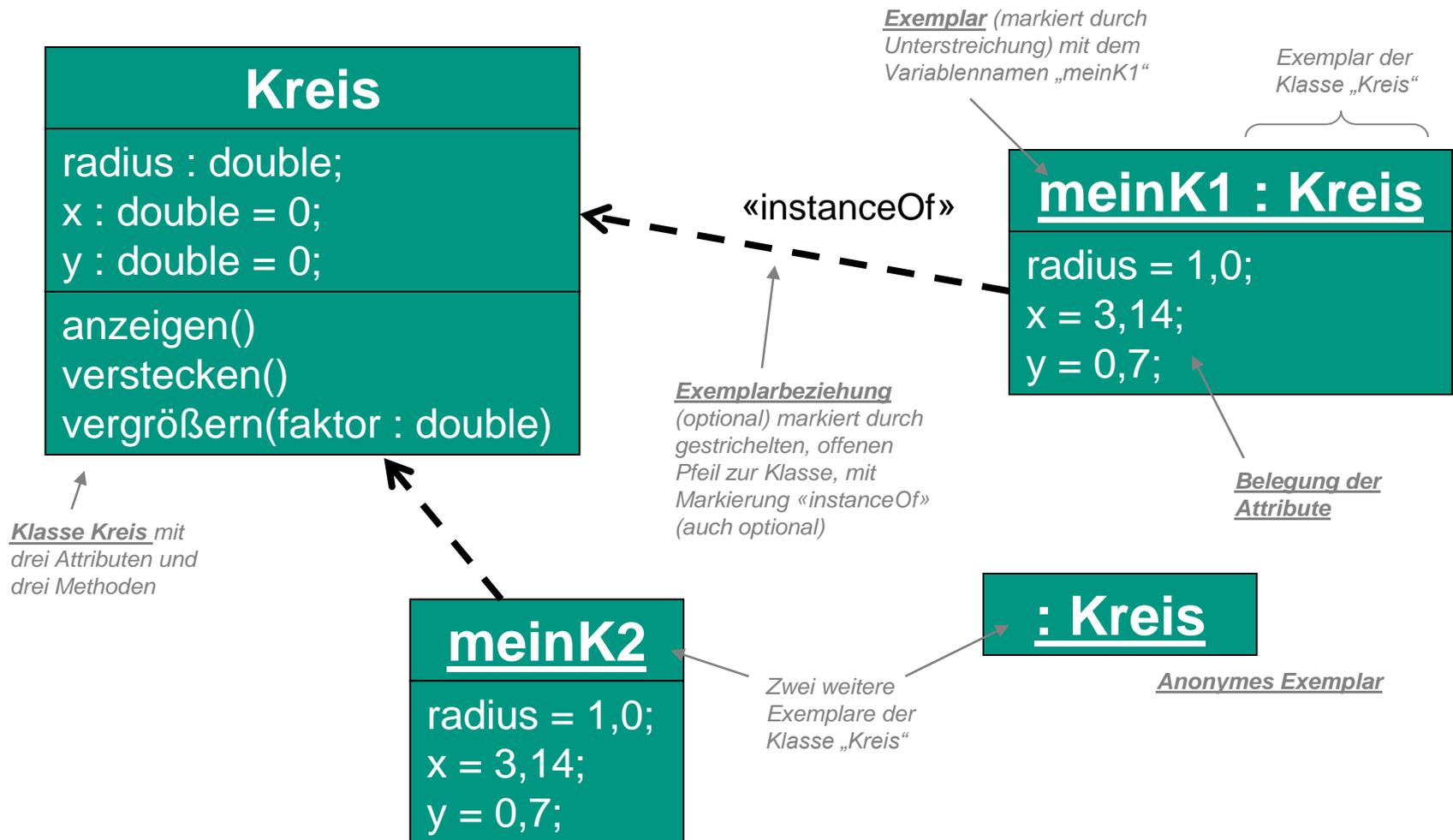


James Rumbaugh

Notation einer Klasse in UML



Objekt-/Instanzdiagramm

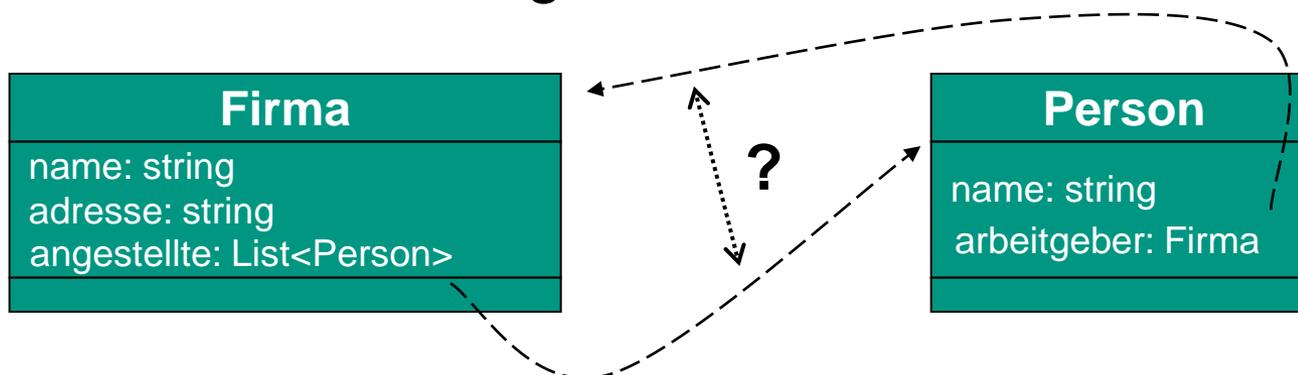


Beziehungen zwischen Objekten

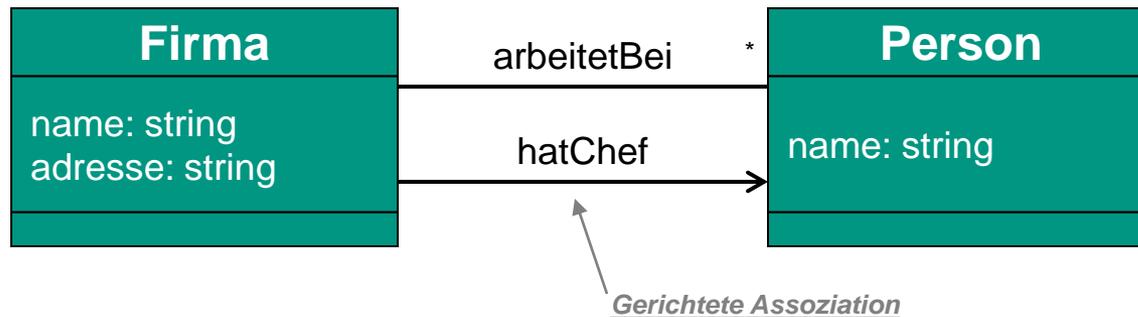
■ Motivierendes Beispiel:

- Wenn wir ausdrücken wollen, dass eine Person bei einer bestimmten Firma arbeitet, dann können wir bei der Klasse „Person“ ein Attribut „Firma“ anlegen
- Wenn wir aber auch wissen wollen, welche Personen bei einer Firma arbeiten, brauchen wir dort eine Liste von Personen (`List<Person>`)

■ Problem: Wie drückt man aus, dass diese Referenzen „irgendwie zusammengehören“?

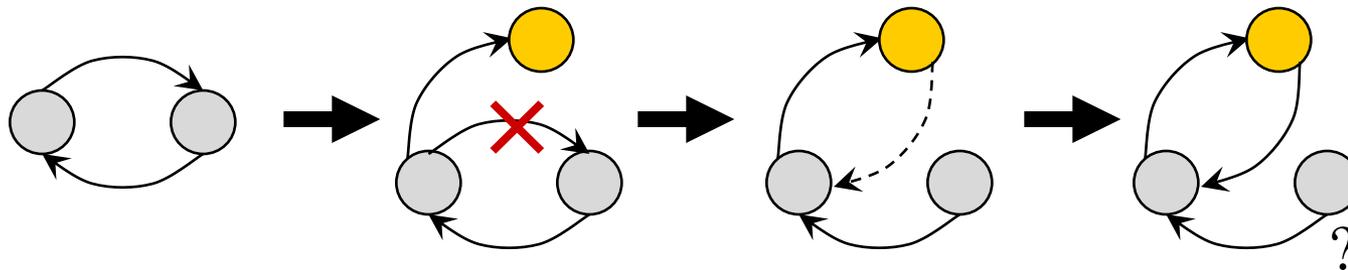


Beispiel: Assoziationen



Hinweis: Klassendiagramme sind Multigraphen, d.h. mehrere Kanten können zw. den gleichen Knoten bestehen.

Semantischer Mehrwert von Assoziationen

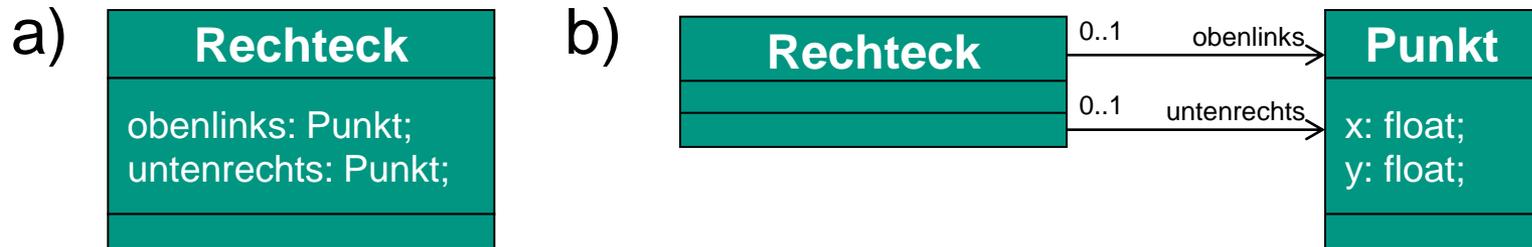


Forderung: **Transaktionalität**, ACID-Prinzip (ISO/IEC 10026-1:1992)

- **Atomicity** – **Unteilbarkeit** der Änderung, „Ganz oder gar nicht“
- **Consistency** – Änderungsvorgänge hinterlassen einen **konsistenten** Zustand (Zielzustand oder im Problemfall auch den Ausgangszustand)
- **Isolation** – Änderungen in nebenläufigen Programmen laufen **ohne Beeinflussung** durch andere Ausführungsfäden ab
- **Durability** – Änderung nach Abschluss für alle Programmteile und Ausführungsfäden **dauerhaft** zu sehen

Wann soll ich ein Attribut, wann eine Assoziation nehmen?

■ Zwei Alternativen:*



- Brauche ich die Transaktionalität? → Assoziation
- Brauche ich mehrere Gegenüber? → Assoziation
- Muss ich vom Gegenüber zurück navigieren können? → Assoziation
- Sonst: freie Wahl
 - Guter Stil: nur Attribute primitiver Typen (int, float, string, bool, ...)

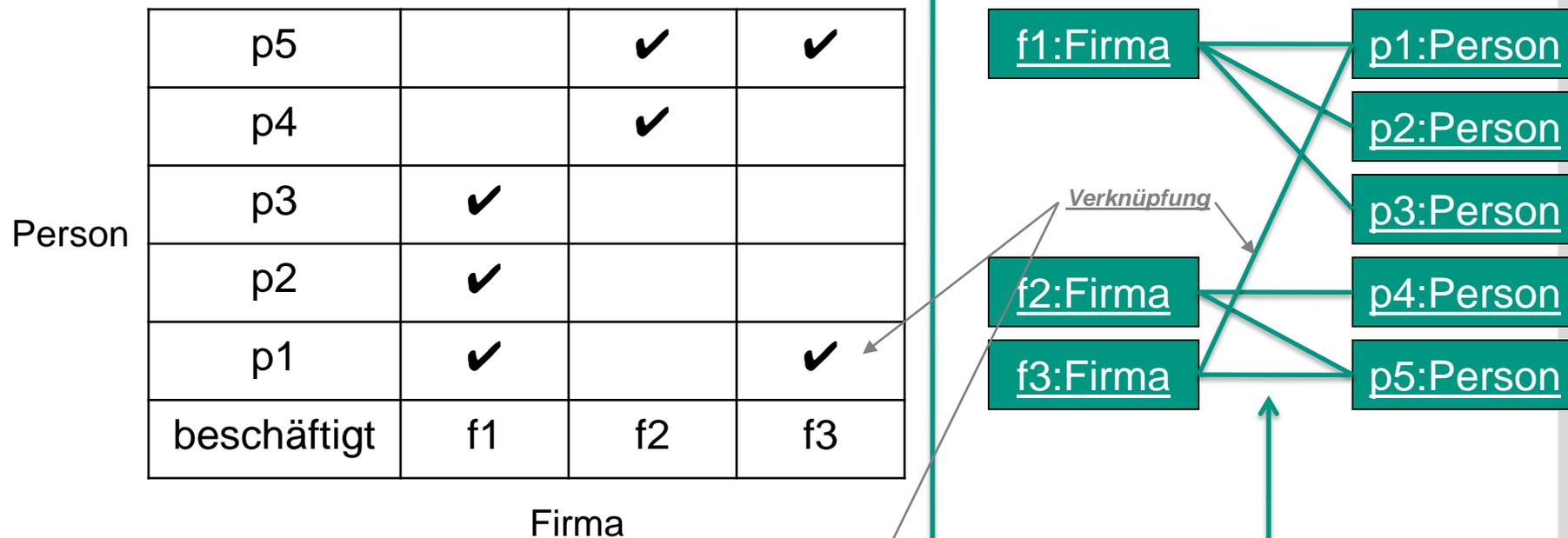
* Zwei Punkte reichen nur für ein Rechteck in achsenparalleler Lage

Assoziationen und Relationen

- Zur Erinnerung: Eine binäre Relation ist eine Teilmenge des kartesischen Produkts (oder Kreuzprodukts) zweier (nicht notwendigerweise unterschiedlicher) Mengen.
- Ternäre, quaternäre, n-äre Relationen sind Untermengen des Kreuzproduktes von drei, vier, n Mengen.
- Relationen lassen sich als n-dimensionale Felder, Tupelmengen oder Graphen darstellen.
- Ein einzelnes Element des Kreuzproduktes, ein einzelnes Tupel, oder eine Kante (mit mehr als zwei Enden bei n-ären Relationen mit $n > 2$).

Beispiel einer binären Relation

Es sei Firma = {f1, f2, f3} und Personen = {p1, p2, p3, p4, p5}.
 Eine mögliche Relation über diese beiden Mengen ist:

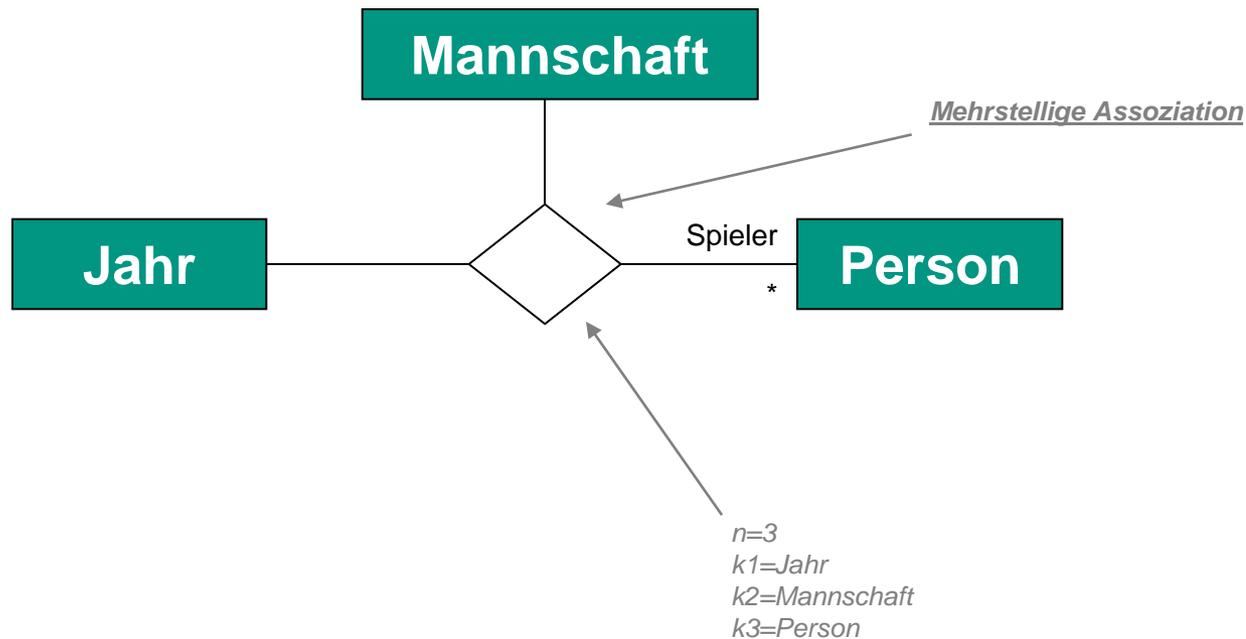


Diese Relation könnte auch als Tupelmengenge angegeben werden:
 {(f1,p1), (f1,p2), (f1,p3), (f2,p4), (f2,p5), (f3,p1), (f3,p5)}, oder als Graph.
 Diese Relation ist linkstotal, rechtstotal (surjektiv), weder rechts- noch
 linkseindeutig, nicht injektiv, nicht bijektiv, nicht funktional.
 Ein einzelnes Tupel oder eine Kante nennt man eine **Verknüpfung**.

Assoziation

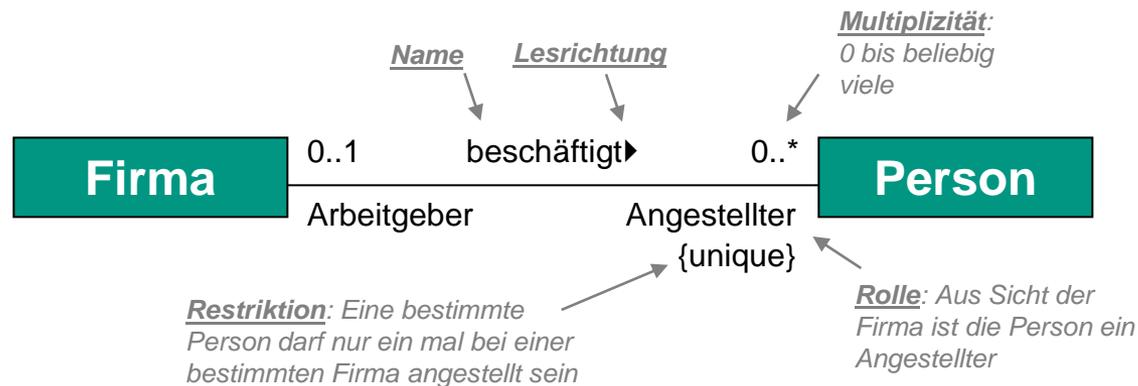
- Def. Eine **Assoziation** (engl. *association*) definiert Eigenschaften von n-ären Relationen:
 - Die Mengen werden als **Klassen** angegeben.
 - **Multiplizitäten** oder Vielfachheiten geben an, in wievielen Tupeln der Relation Elemente einer geg. Klasse erscheinen dürfen (z.B. eindeutig 1:1, mehrdeutig: 1:n, m:n, sowie Totalität)
 - Die Klassen brauchen **nicht disjunkt** zu sein (Selbstreferenz ist erlaubt).
 - Ein Tupel einer Relation heißt **Verknüpfung**.
 - Duplikate von Tupeln sind erlaubt, d.h. die Relationen sind eigentlich **Mehrfachmengen** oder auch Multigraphen.

Beispiel für mehrstellige Assoziation (Hypergraph)



Standardattribute von Assoziationen und Assoziationsenden

- Die durch eine Assoziation charakterisierte Relation kann genauer beschrieben werden:



- Name, Leserichtung und Rolle sind „nur“ Etikette, die der Interpretation dienen. Sie haben aber keine genauer definierte Semantik.

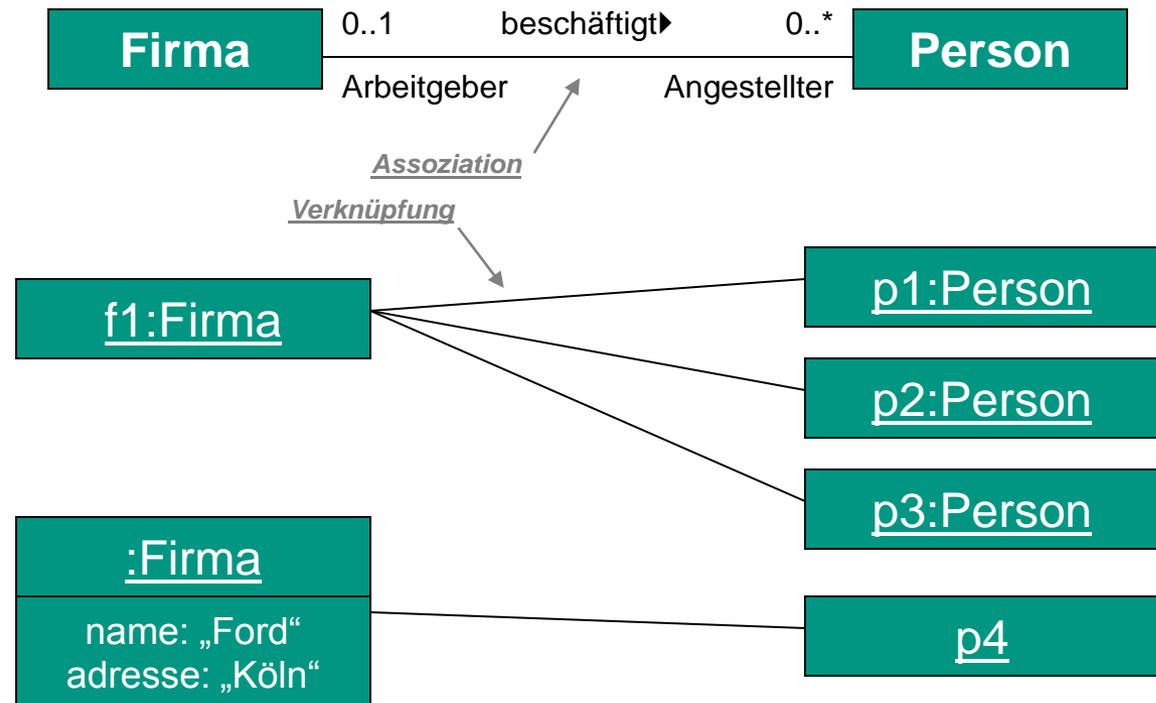
Assoziation vs. Verknüpfung

Assoziation

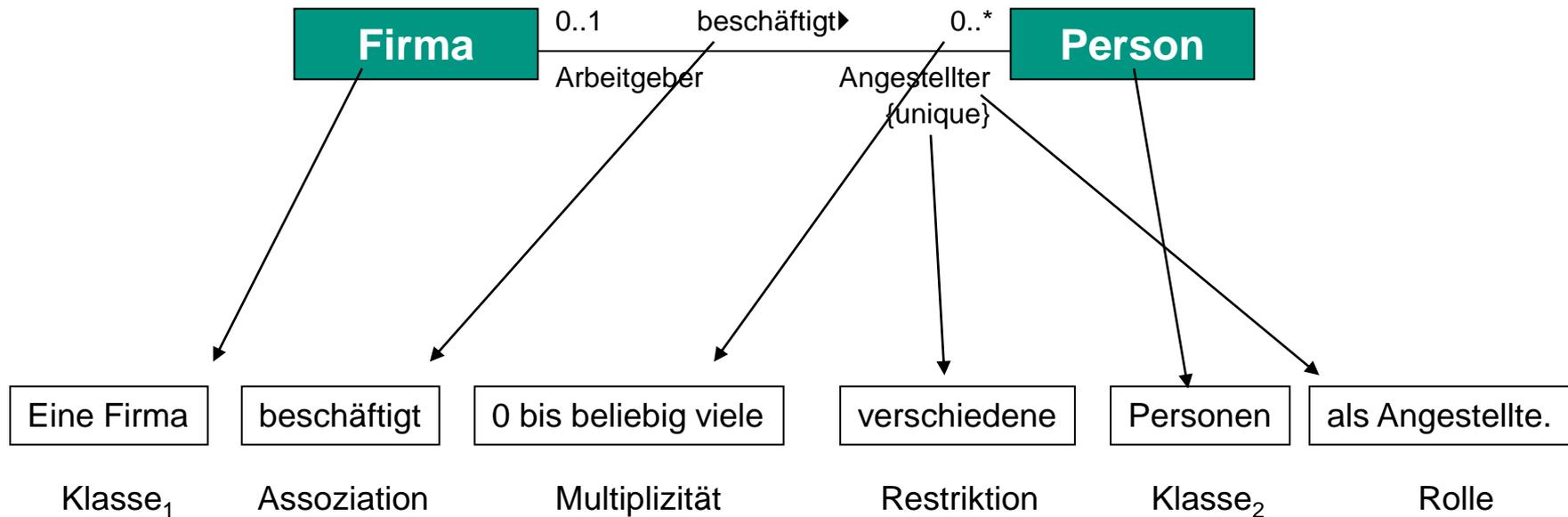
wird zwischen Klassen angegeben und beschreibt **mögliche Beziehungen** zwischen Exemplaren

Verknüpfung

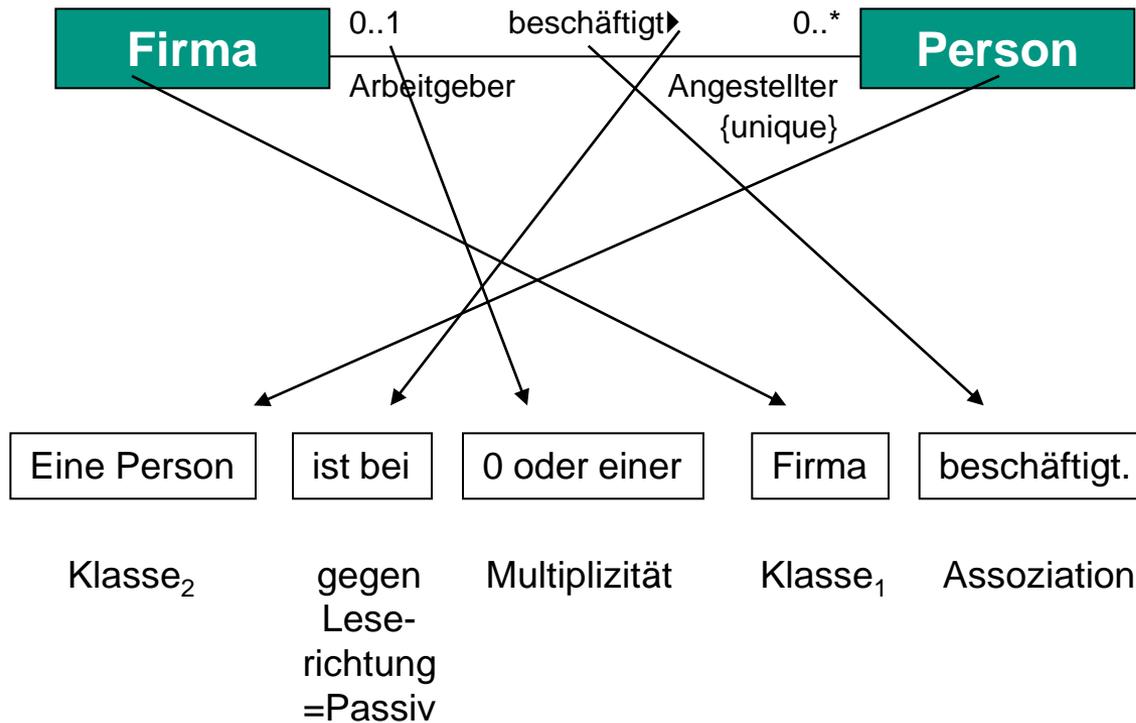
wird zwischen Exemplaren angegeben, drücken eine **tatsächliche Beziehung** zwischen Exemplaren aus



Standardattribute von Assoziationen und Assoziationsenden



Standardattribute von Assoziationen und Assoziationsenden



Standardattribute von Assoziationsenden: Multiplizität

- Def. **Kardinalität**: die Anzahl der Elemente einer Menge (\rightarrow Ganzzahl ≥ 0)
- Def. **Multiplizität**: ein geschlossenes Intervall der zulässigen Kardinalitäten
 - „0..1“ heißt „0 oder 1“
 - „0..*“ = „*“ heißt „beliebig viele, auch 0“
 - „1“ heißt „immer genau 1“
 - „1..*“ heißt „beliebig viele, aber mindestens 1“
 - „17“ heißt „immer genau 17“
 - Achtung: keine Angabe heißt „nicht spezifiziert“ und das ist spezifikationsgemäß „1“, also „immer genau 1“

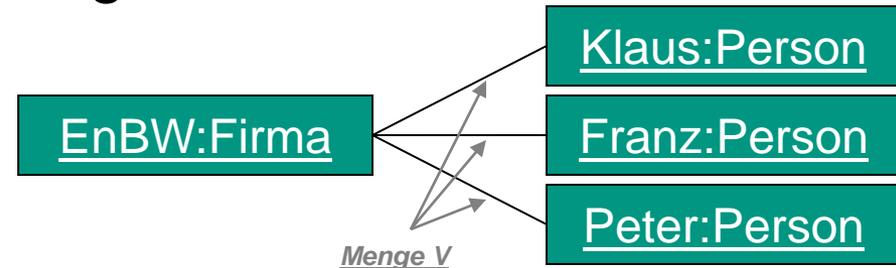
Interpretation der Multiplizität

- Gegeben: eine bel. a. f. n-äre Assoziation
 - Sei E ein bel. a. f. Ende der geg. Assoziation und K die Klasse, die am Ende E hängt
 - Wähle für die anderen $n-1$ Enden bel. a. f. Instanzen ihrer entsprechenden Klassen (mehr als $n-1$ möglich!)
 - Sei V die Menge der Verknüpfungen (aus der geg. Assoziation), die von dem gewählten Satz von Instanzen zu Instanzen der Klasse K gehen
 - Sei M diese Menge der Instanzen der Klasse K
 - Dann beschränkt die *bei E* hingeschriebene Multiplizität die zulässige Kardinalität von M

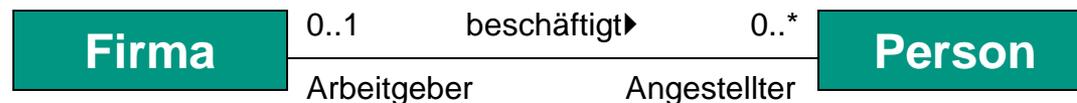
Beispiel

- Gegeben: Assoziation „beschäftigt“, wobei $n=2$

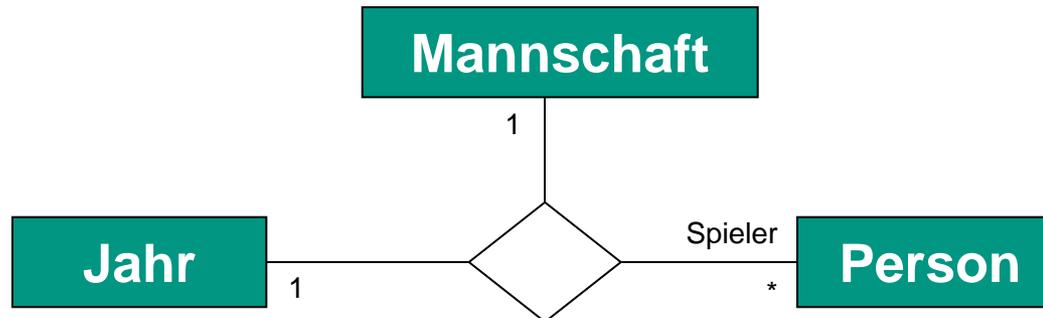
- Wähle $E=$ Angestellter, dann ist $K=$ Person
- Wähle für das andere Ende EnBW:Firma



- Sei V die Menge der Verknüpfungen, die von EnBW:Firma zu Instanzen der Klasse Person gehen
- $M=\{\text{Klaus:Person}, \text{Franz:Person}, \text{Peter:Person}\}$
- Dann beschränkt die bei Angestellter hingeschriebene Multiplizität ($0..*$) die zulässige Kardinalität von M .



Wofür? Mehrstellige Assoziationen!



Übung: Wie ist das zu interpretieren?

Multiplizität bei mehrstelligen Assoziationen

- Gegeben: eine bel. a. f. n-äre Assoziation
 - Wähle E/K=Mannschaft
 - Wähle 2002:Jahr, MiroslavKlose:Person, MarioBasler:Person und HarryKoch:Person
 - Sei V die Menge der 3-stelligen Verknüpfungen, die von den Instanzen der Klassen Jahr und Person zu Instanzen der Klasse Mannschaft gehen
 - Sei M diese Menge der Instanzen der Mannschaft
 - Dann beschränkt die bei Mannschaft hingeschriebene Multiplizität (1) die Mächtigkeit der Menge M
- Das heißt, die Personen MiroslavKlose, MarioBasler und HarryKoch spielen im Jahr 2002 in genau einer Mannschaft (genauer: jede Person muss immer in genau einer Mannschaft spielen)

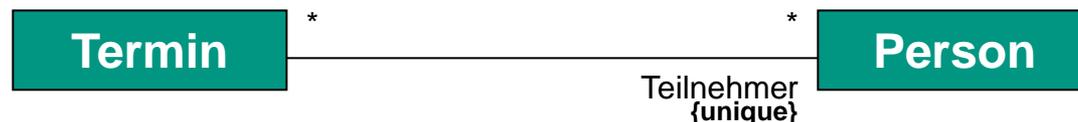
Standardattribute von Assoziationsenden: Restriktionen

- **{ordered}** : Sei $\rho \subseteq K_1 \times K_2$ die einer Assoziation zugrunde liegende Relation und „{ordered}“ auf der Seite von K_2 angegeben. Dann existiert eine lineare Ordnung $\sigma \subseteq K_2^2$ so dass
$$\forall x \in K_1 \forall y, z \in K_2 : x \rho y \wedge x \rho z \Rightarrow y \sigma z \vee z \sigma y$$
- **{unique}** : Sei $\rho \subseteq K_1 \times K_2$ die einer Assoziation zugrunde liegende Relation und „{unique}“ auf der Seite von K_2 angegeben. Dann gilt
$$\forall x \in K_1 \forall y, z \in K_2 : x \rho y \wedge x \rho z \Rightarrow y = z$$

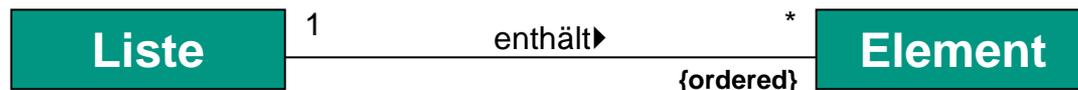
{ordered}	{unique}	Implementierung
nein	nein	z.B. java.util.Collection
nein	ja	z.B. java.util.Set
ja	nein	z.B. java.util.List
ja	ja	z.B. java.util.SortedSet

Beispiele für Restriktionen

- Eine Person kann zwar an beliebig vielen Terminen teilnehmen, aber an einem Termin **nur ein mal**:

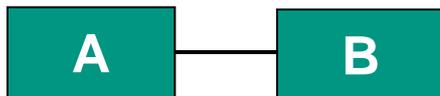


- Die Elemente einer Liste haben eine bestimmte **Reihenfolge**:

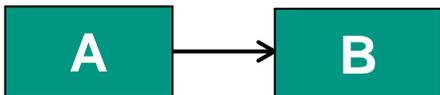


Standardattribute von Assoziationsenden: Navigation

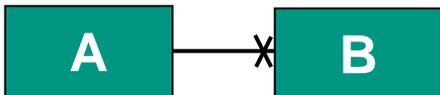
- Exemplare können nur Nachrichten an Exemplare senden, **die sie kennen**. Eine Verknüpfung kann zwei (oder mehr) Exemplare einander bekannt machen. D.h. Assoziationen modellieren Nachrichtenkanäle
- Navigation spezifiziert die **Richtung** des Kanals



Nicht spezifiziert. Navigation in beide Richtungen möglich, aber nicht garantiert.



Von A nach B navigierbar.



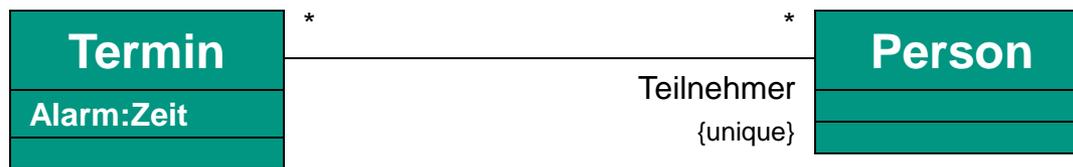
Von A nach B nicht navigierbar.

Zusätzliche Attribute für Assoziationen

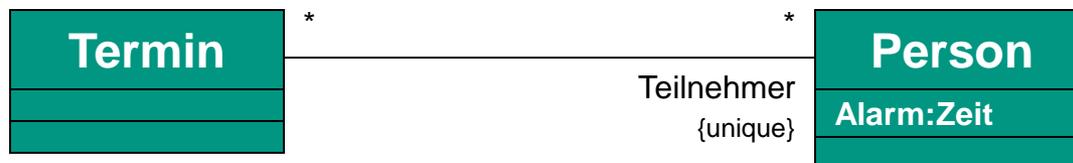
- Manchmal möchte man weitere Eigenschaften für eine Verknüpfung notieren:



- Ein Teilnehmer soll sich für einen Termin einen Alarmzeitpunkt eintragen können:

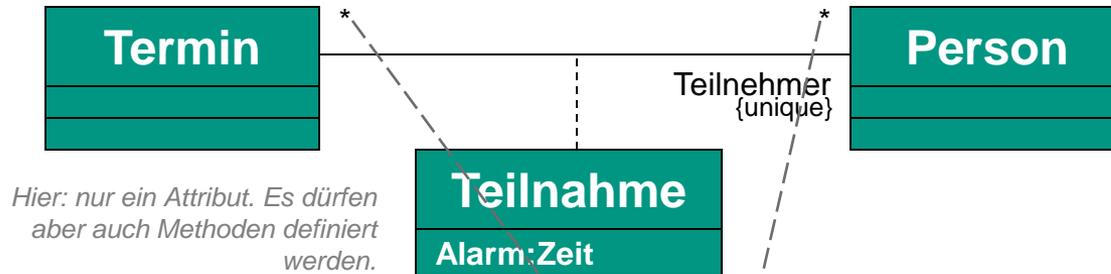


Falsch: Nur ein Alarm pro Termin für alle **Personen**



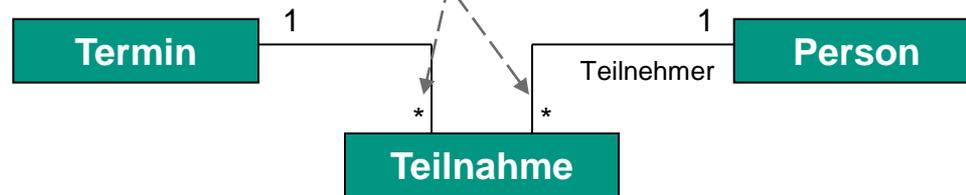
Falsch: Nur ein Alarm pro Person für alle **Termine**

Assoziationsklassen

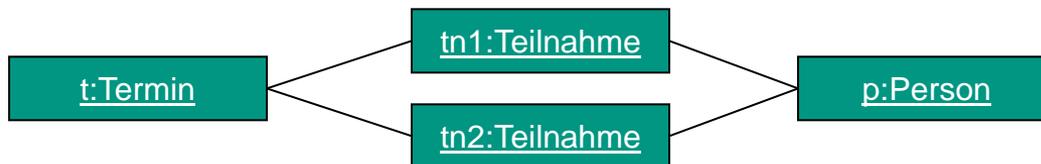


Hinweis:
Klassendiagramme sind also Multi-Hyper-Graphen mit attribuierten Knoten und attribuierten Kanten!

- Es gilt: Klassenname=Assoziationsname
- Simulation durch „normale“ Klasse:



- Aber:



Wäre eine legale Verknüpfung in diesem Modell → Zusicherungen!

Assoziationsklassen

- Hinweis:
 - Die Existenz einer Instanz der Assoziationsklasse hängt an der Existenz der zugehörigen **Verknüpfung!!!**
 - Folge: Wenn die Verknüpfung gelöscht wird, ist auch die Instanz der Assoziationsklasse weg
 - Außerdem: Wenn ein Verknüpfungspartner gelöscht wird, wird die Verknüpfung gelöscht und dann auch die Instanz der Assoziationsklasse
- Folgerung: **Nicht** wie eine normale Klasse im UML-Modell verwenden!

Spezialformen von Assoziationen

- Es gibt Spezialformen von Assoziationen. Also solche mit speziellen Interpretationsvorschriften
 - **Aggregation** (Sonderform der **Assoziation**):
Teil-Ganzes-Beziehung

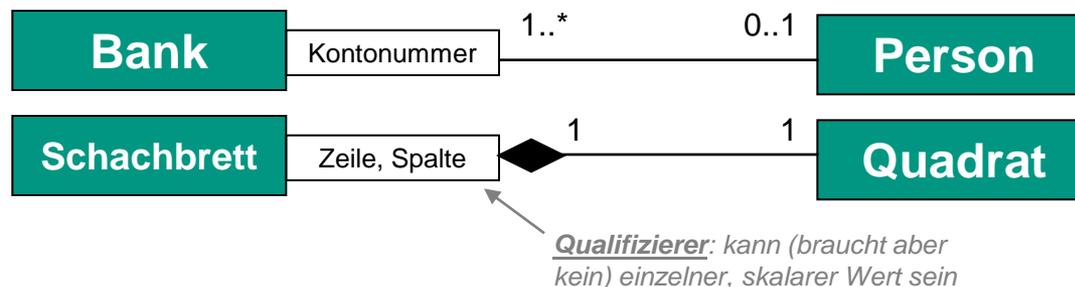


- **Komposition** (Sonderform der **Aggregation**):
Strenger, Teile haben keine Daseinsberechtigung ohne das Ganze, Semantik wichtig z.B. bei Löschooperationen



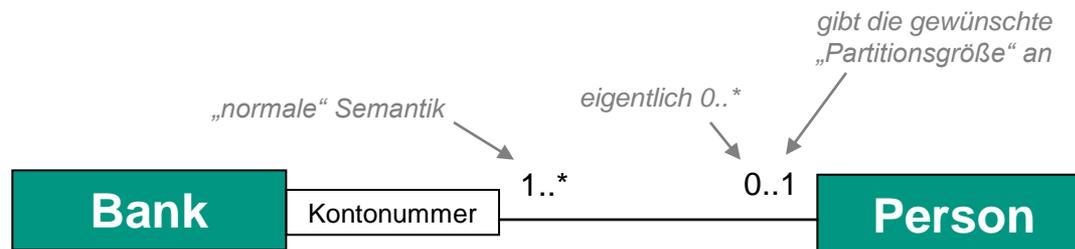
Spezialformen von Assoziationen

- Es gibt weitere Spezialformen... (Forts.)
 - Def. **Qualifizierer**: Ein(e) Attribut(kombination), die eine Partitionierung auf der Menge der assoziierten Exemplaren definiert.
 - Def. **qualifizierte Assoziation**: Eine Assoziation, bei der die Menge der referenzierten Objekte durch einen Qualifizierer partitioniert ist.
 - Beispiele:



Multiplizität bei qualifizierten Assoziationen

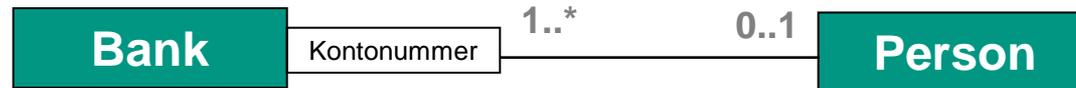
- Aus der Definition folgt: Qualifizierte Assoziationen codieren implizit **immer 1:n- oder m:n-Beziehungen**
 - Die (häufige!) Partitionsgröße „1“ erfordert eine eindeutige Qualifizierer-Instanz.
- Interpretation der Multiplizitäten:



Zitat aus dem UML-Standard

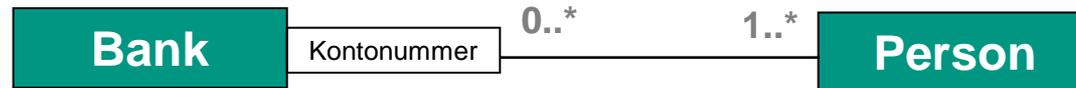
- A qualifier declares a partition of the set of associated instances with respect to an instance at the qualified end (the qualified instance is at the end to which the qualifier is attached). A qualifier instance comprises one value for each qualifier attribute. Given a qualified object and a qualifier instance, the number of objects at the other end of the association is constrained by the declared multiplicity. In the common case in which the multiplicity is 0..1, the qualifier value is unique with respect to the qualified object, and designates at most one associated object. In the general case of multiplicity 0..*, the set of associated instances is partitioned into subsets, each selected by a given qualifier instance. In the case of multiplicity 1 or 0..1, the qualifier has both semantic and implementation consequences. In the case of multiplicity 0..*, it has no real semantic consequences but suggests an implementation that facilitates easy access of sets of associated instances linked by a given qualifier value.
- **Note** – The multiplicity of a qualifier is given assuming that the qualifier value is supplied. The “raw” multiplicity without the qualifier is assumed to be 0..*. This is not fully general but it is almost always adequate, as a situation in which the raw multiplicity is 1 would best be modeled without a qualifier.
- **Note** – A qualified multiplicity whose lower bound is zero indicates that a given qualifier value may be absent, while a lower bound of 1 indicates that any possible qualifier value must be present. The latter is reasonable only for qualifiers with a finite number of values (such as enumerated values or integer ranges) that represent full tables indexed by some finite range of values.

Multiplizität bei qualifizierten Assoziationen – Beispiel



- Bei der Bank X sind **keine** oder **maximal eine** Person der Kontonummer Y zugeordnet.
- **Niemand** oder **maximal 1** Person besitzt das Konto mit der Nummer Y bei der Bank X.
- Eine Person muss **mindestens eine** Kontonummer bei mindestens einer Bank zugeordnet sein.
- Eine Person darf **beliebig viele** Kontonummern bei einer Bank, eine Kontonummer bei beliebig vielen Banken oder beliebig viele Kontonummern bei beliebig vielen Banken haben.

Multiplizität bei qualifizierten Assoziationen – Beispiel



- Bei der Bank X ist **jeder** Kontonummer Y **mindestens eine** Person zugeordnet.
- Mehrere Personen können **gemeinsam eine** Kontonummer bei einer Bank haben.
- Eine Person braucht kein Konto haben (also keiner Kontonummer bei irgend einer Bank zugewiesen zu sein).
- Eine Person kann **beliebig viele** Konten bei **beliebig vielen** Banken haben.

- Wozu könnte man qualifizierte Assoziationen benutzen?

Klassenattribute und -methoden

- Die Klasse als Menge der Exemplare eines Typs wird selbst auch als Objekt aufgefasst. Daher kann sie selbst eigene Attribute und Methoden (unabhängig von den Attributen und Methoden der Exemplare) definieren.

Math
<u>E : double;</u>
<u>PI : double;</u>
<u>sqrt(a:double):double;</u>
<u>cos(a:double):double;</u>

Diese Attribute und Methoden bezeichnet man als **Klassenattribute** und **Klassenmethoden**.

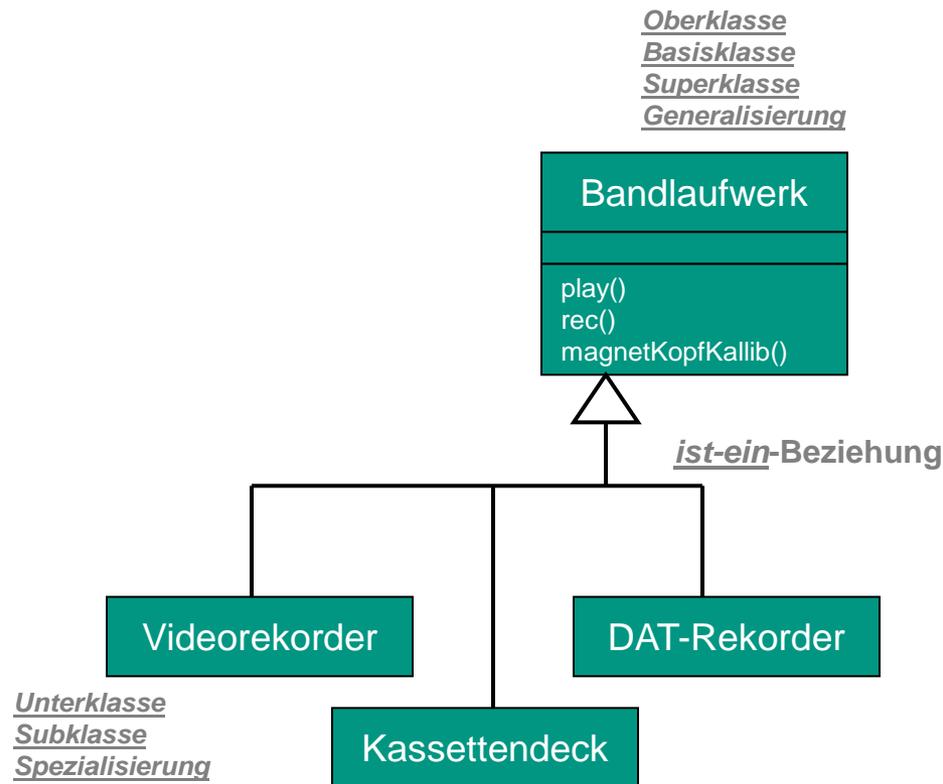
- Markierung durch **Unterstreichen** der Zeile im Kästchen.
- Klassenattribute und -methoden existieren unabhängig von der Existenz von Exemplaren der Klasse
→ globale Verfügbarkeit.
- Die Klassenattribute und -methoden werden üblicherweise zur Laufzeit in jede Instanz „eingebündelt“, können dort also wie die eigenen verwendet werden.
- In Java mit **static** markiert.

Vererbung (engl. *inheritance*)

IPD Tichy, Fakultät für Informatik



Vererbung – Begriffe und Synonyme



Redundanz

- Manchmal
 - ... lassen sich für verschiedene Klassen **gleiche Teilmengen** von Attributen, Zuständen und Assoziationen identifizieren.
 - ... sind nur die Elemente dieser Teilmengen von Bedeutung, dafür sollen aber Exemplare von verschiedenen Klassen verwendet werden.
- In diesem Fall bietet das Konzept „Vererbung“ Vorteile
 - Vermeidung von Entwurfs- und Implementierungsredundanz
 - Theoretisch gut fundiertes Typisierungskonzept

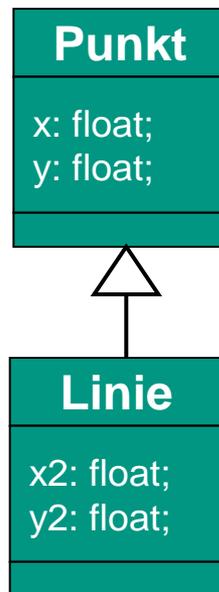
Substitutionsprinzip

- Def. Ist-ein-Semantik/**Substitutionsprinzip**: Jedes Exemplar einer Unterklasse hat die gleichen Eigenschaften*, die ein Exemplar seiner Oberklasse hätte; es lässt sich genau so verwenden.
 - Alle Eigenschaften der **Oberklasse** müssen in der Unterklasse vorhanden sein.
 - Alle Eigenschaften der **Unterklasse** müssen in allen Situationen so verwendbar sein, wie es Eigenschaften der Oberklasse wären und „kompatible“ Ergebnisse zurückgeben.

* Attribute, Assoziationen, Zusicherungen, Zustände, Methoden

Substitutionsprinzip

- Zu modellieren sei eine Linie. Dazu könnte man die Klasse Punkt benutzen. Ist die Vererbung hier eine gute Idee?



Eine Linie ist kein Sonderfall eines Punktes.
Diese Modellierung ist faul!

Folgerungen aus dem Substitutionsprinzip (I)

Hinzufügen/Weglassen von Eigenschaften

- Vererbte Eigenschaften* stehen in **der Unterklasse genau so** zur Verfügung, wie sie in der Oberklasse definiert sind.
- Die Unterklasse darf noch **zusätzliche** Eigenschaften* definieren, die sie spezieller machen.
- Die Unterklasse kann keine Eigenschaften* der Oberklasse **weglassen**.
 - Wäre dies nötig, handelt es sich nicht um eine Vererbungsbeziehung.
 - In diesem Falle zur Vermeidung von Implementierungsredundanz statt Vererbung von einer Klasse die „Delegation“ an diese Klasse einsetzen!

* Attribute, Assoziationen, Zusicherungen und Zustände

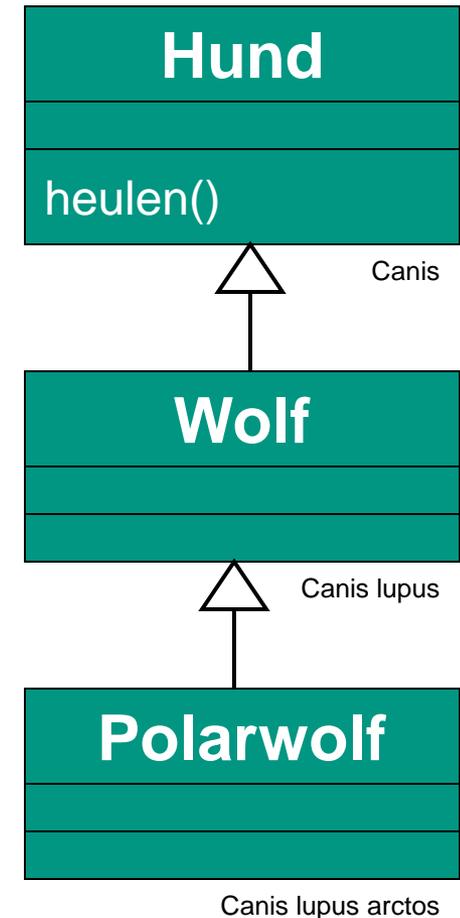
Folgerungen aus dem Substitutionsprinzip (II)

Transitivität

- Die Vererbungsbeziehung ist **transitiv**.
 - Ein Hund kann heulen().
 - Ein Wolf ist ein Hund.
 - Der Wolf erbt die Methode heulen() vom Hund
⇒ der Wolf kann ebenfalls heulen().
 - Ein Polarwolf ist ein Wolf.
 - Der Polarwolf erbt alle Methoden vom Wolf inkl. heulen()
⇒ der Polarwolf kann ebenfalls heulen().



⇒ Man kann einer Polarwolfinstanz die Nachricht *heulen()* schicken



Unterscheide!

- Def. **Signaturvererbung**: Eine in der Oberklasse definierte und (evtl.) implementierte Methode überträgt **nur ihre Signatur** auf die Unterklasse.
- Def. **Implementierungsvererbung**: Eine in der Oberklasse definierte und implementierte Methode überträgt ihre **Signatur und ihre Implementierung** auf die Unterklasse.
- Implementierungsvererbung geht nicht ohne Signaturvererbung, aber umgekehrt geht es schon.
- „Signaturvererbung reicht für ‚alles‘ in der OOA/OOD/OOP aus, Implementierungsvererbung ist nicht notwendig (aber bequem).“
- Beispiel: Java und C# bieten sowohl Implementierungsvererbung als auch reine Signaturvererbung an.

Folgerungen aus dem Substitutionsprinzip (III)

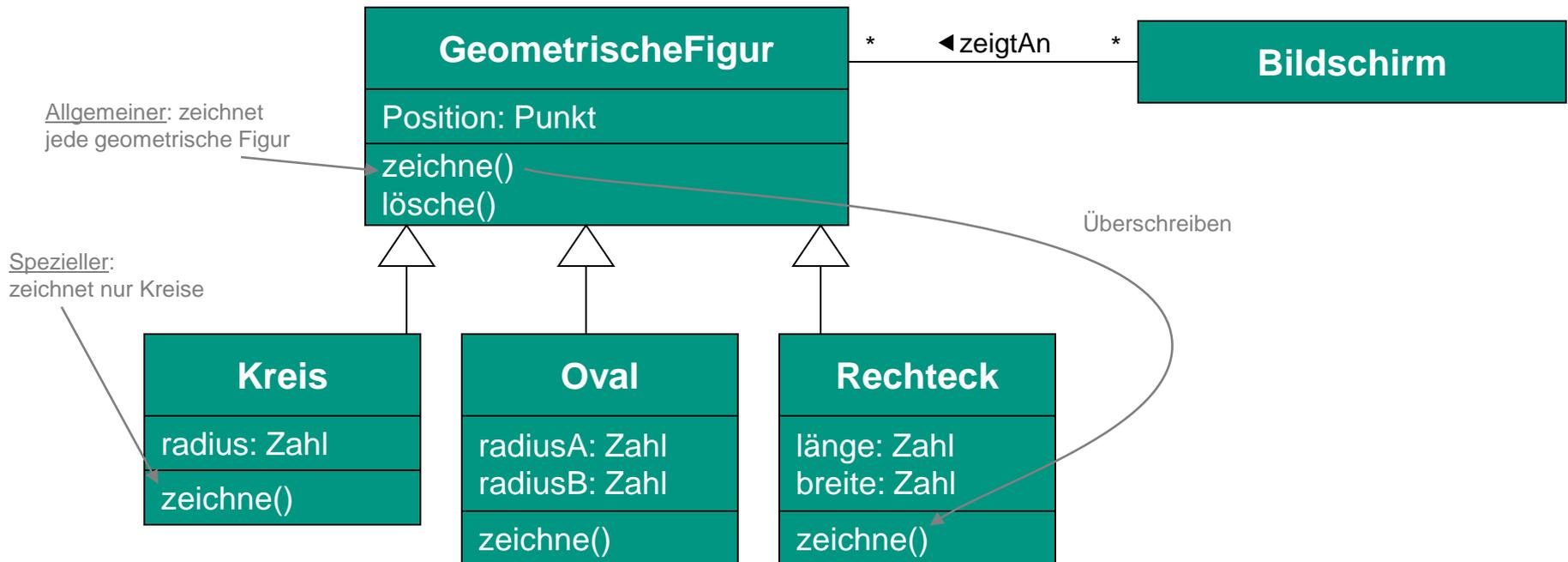
Anpassung ererbter Eigenschaften

- Eigenschaften der Basisklasse können an die Bedürfnisse der Spezialisierung angepasst werden (z.B. Methoden verändert werden)
- Def. **Überschreiben**: eine geerbte Methode unter Beibehaltung der Signatur wird **neu implementiert**

Hat nichts mit Vererbung zu tun!

Im Gegensatz zum „Überladen“:
Dabei wird eine **neue Methode**
mit **gleichem Namen**
aber **anderer Signatur** definiert.

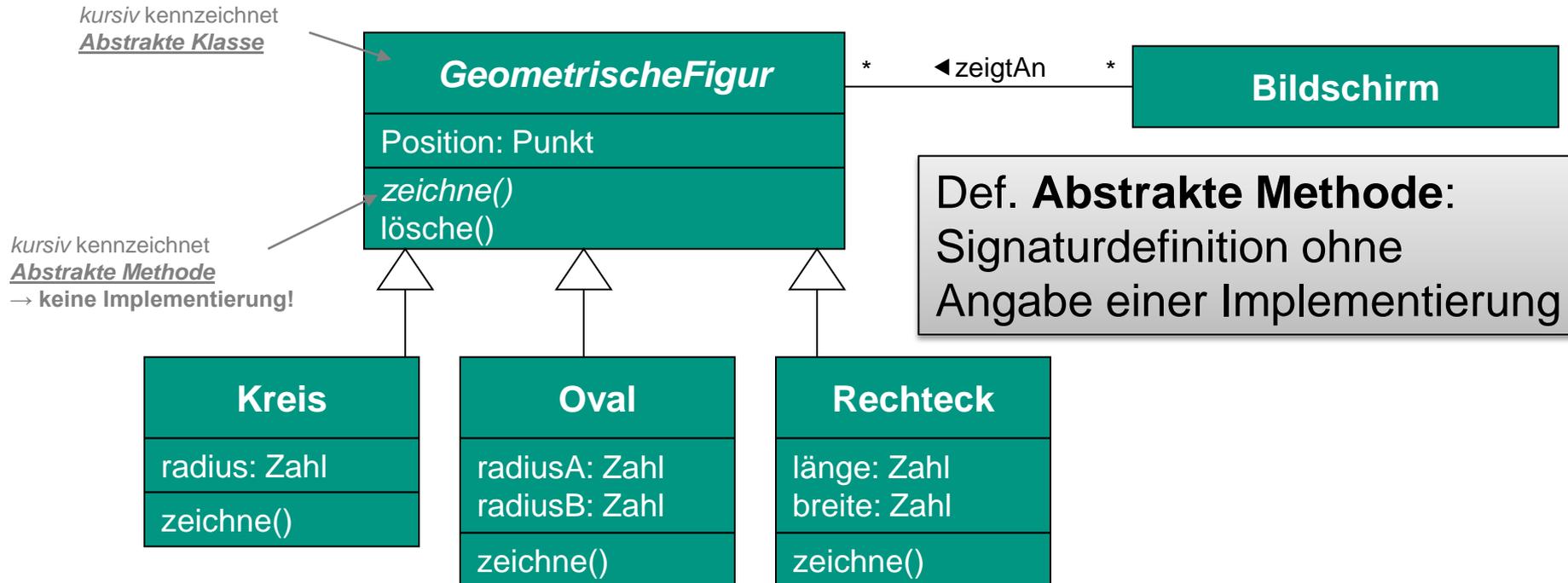
Überschreiben – Beispiel



■ Einsicht:

- Jede der drei Spezialisierungen muss ihre eigene **zeichne-** Methode implementieren
- Kann es überhaupt eine sinnvolle Implementierung für **zeichne** in **GeometrischeFigur** geben?

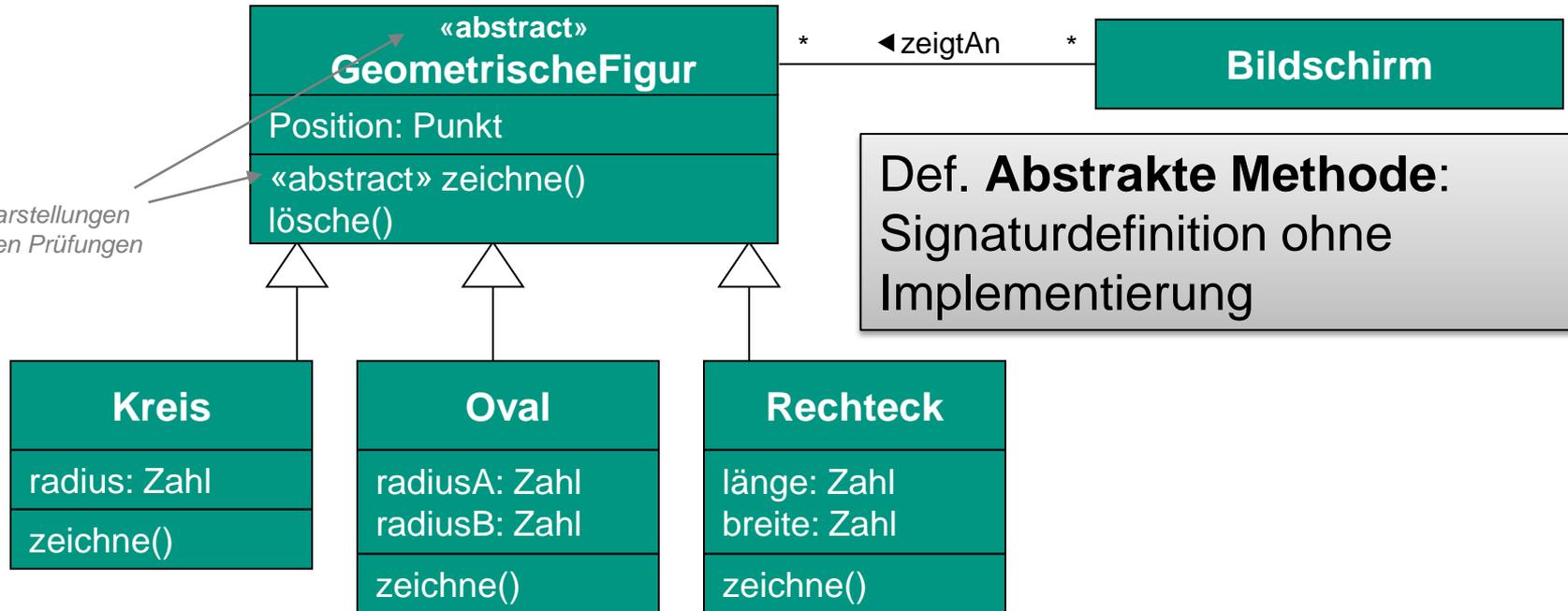
Lösung: Abstrakte Methoden



- Wenn es eine abstrakte Methode gibt, dann ist auch die sie enthaltende Klasse abstrakt.
- Von abstrakten Klassen selbst gibt es keine Exemplare. Warum?
- Abstrakte Klassen „vererben die Verpflichtung“, eine Methode zu implementieren. Warum?

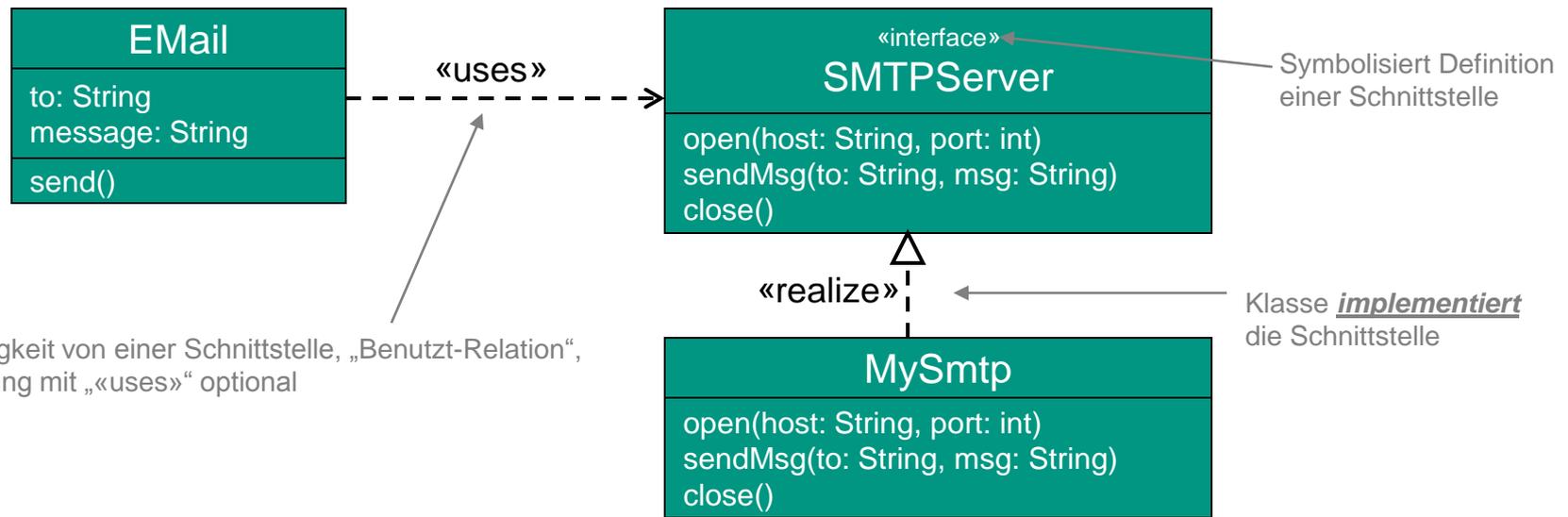
Abstrakte Methoden (alternative Darstellung)

alternative Darstellungen
(in schriftlichen Prüfungen
bevorzugt ☺)



Schnittstelle

- Def. **Schnittstelle**: Definition einer Menge **abstrakter Methoden**, die von den Klassen, die sie implementieren, anbieten müssen.
 - Schnittstelle nicht direkt instanziiierbar
 - Nutzende Klasse darf bel. implementierende Klassen instanziiieren.



Abhängigkeit von einer Schnittstelle, „Benutzt-Relation“, Markierung mit „«uses»“ optional

Klasse implementiert die Schnittstelle

Verwendung von Schnittstellen

- Schnittstellen übertragen die **Verpflichtung**, bestimmte Methoden zu implementieren.
 - Damit geben Schnittstellen aber auch die **Garantie**, dass bestimmte Methoden vorhanden sind.
 - Exemplare einer Klasse, die eine bestimmte Schnittstelle implementiert, können genauso verwendet werden, als wären sie ein „Exemplar der Schnittstelle“ (*ist-ein*-Semantik)
- Idee: mit einer Schnittstelle gibt man an, **wie** ein Objekt zu verwenden ist, **nicht was** es darstellt.

Schnittstelle – Beispiel

- Sie bekommen einen „schwarzen“ Kasten.
- Sie sehen, dass man die Nachrichten
 - play()
 - stop()
 - ff()
 - rwd()
 - skip()an das Gerät senden kann.
- Obwohl Sie nicht wissen, was für ein Gerät das ist, können Sie es benutzen, da Sie die Schnittstelle kennen.



„Vererbung“ bei Schnittstellen

- Bei Schnittstellen gibt es **keine Vererbung!**
 - Def: Wenn eine Schnittstelle A eine Schnittstelle B **erweitert**, dann ist die Menge der abstrakten Methoden von A eine Teilmenge der Menge der abstrakten Methoden von B (oder kürzer: $A \subseteq B$).
 - Def: Wenn eine Klasse X eine Schnittstelle A **implementiert**, dann ist die Menge der abstrakten Methoden von A eine Teilmenge der Methodendefinitionen von X, wobei X zusätzlich jeweils eine Implementierung angeben darf.
- Da es sich nur um Mengeninklusion handelt, ist „Mehrfachvererbung“ von Schnittstellen unproblematisch.
- Übung: Welche Probleme können bei Mehrfachvererbung von Klassen auftreten?

Folgerungen aus dem Substitutionsprinzip (IV)

Signaturanpassung

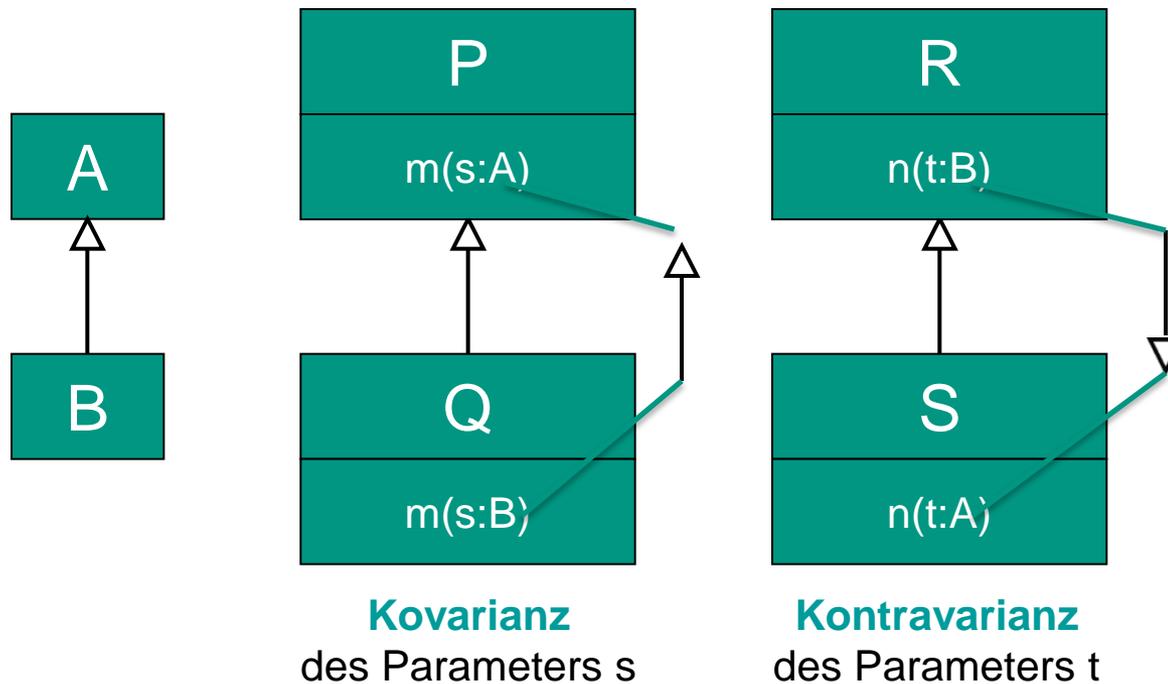
- Das Substitutionsprinzip fordert nur, dass man ein Exemplar der Unterklasse so **verwenden** kann, als wäre es ein Exemplar der Oberklasse. Es wird nicht gefordert, dass die Signatur **gleich** bleibt!
- ⇒ Änderungen der Signatur sollten eingeschränkt möglich sein – aber welche?

Folgerungen aus dem Substitutionsprinzip (IV)

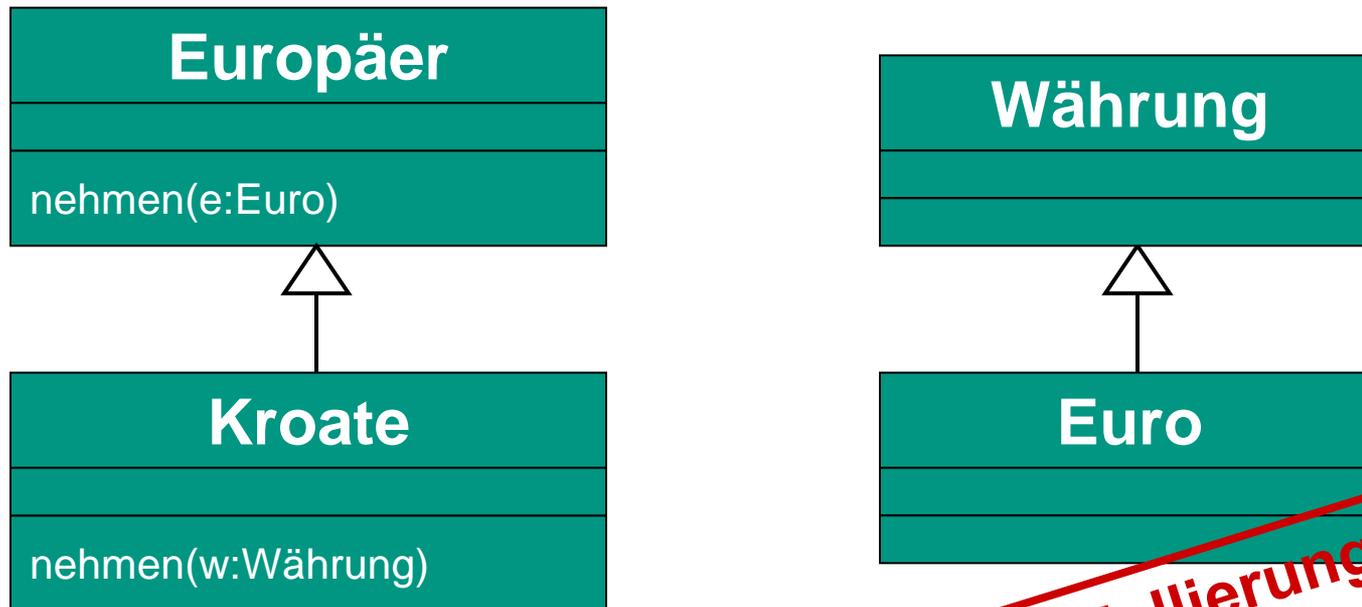
Parameter-Varianz

- Def. **Varianz**: Modifikation der Typen der Parameter einer überschriebenen Methode
- Def. **Kovarianz**: Verwendung einer Spezialisierung des Parametertyps in der überschreibenden Methode
- Def. **Kontravarianz**: Verwendung einer Verallgemeinerung des Parametertyps in der überschreibenden Methode
- Def. **Invarianz**: keine Modifikation des Typs

Beispiel Ko-/Kontravarianz

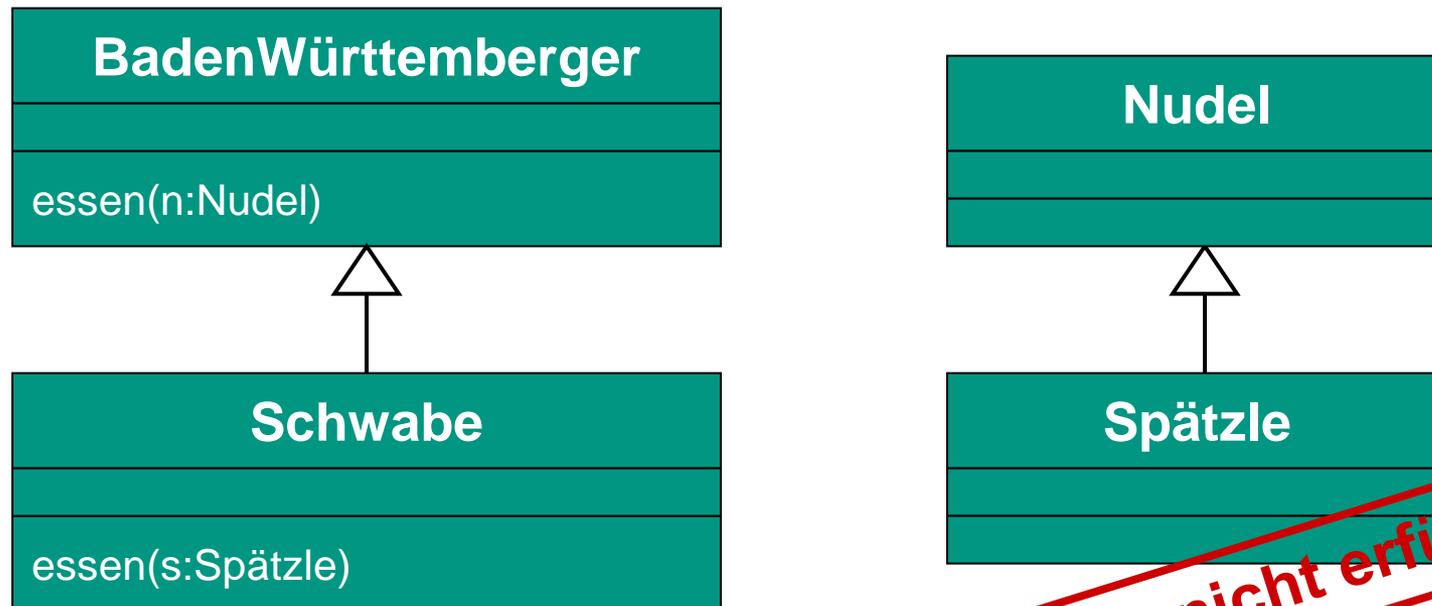


Kontravarianz und Substitutionsprinzip – Beispiel



**Zulässige Modellierung:
Substitutionsprinzip erfüllt**

Kovarianz und Substitutionsprinzip – Beispiel



Substitutionsprinzip nicht erfüllt!

Folgerungen aus dem Substitutionsprinzip (IV)

Zulässige Varianz

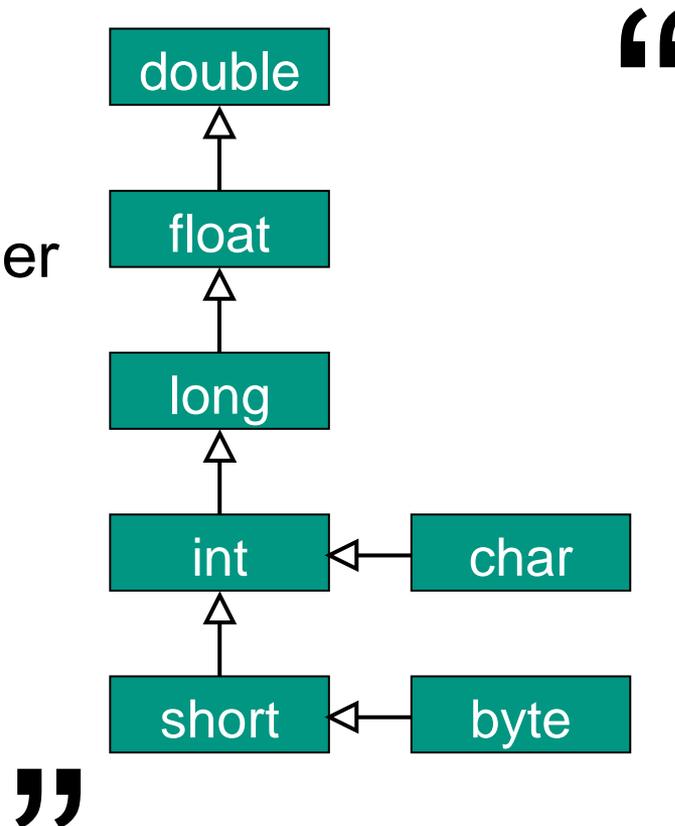
- Um das Substitutionsprinzip zu erfüllen sind folgende Modifikationen der Parametertypen bei überschreibenden Methoden zulässig:

Eingabeparameter	Kontravarianz
Ausgabeparameter (auch Rückgabewert und Ausnahmen)	Kovarianz
Parameter, die gleichzeitig Ein- und Ausgabeparameter sind	Invarianz

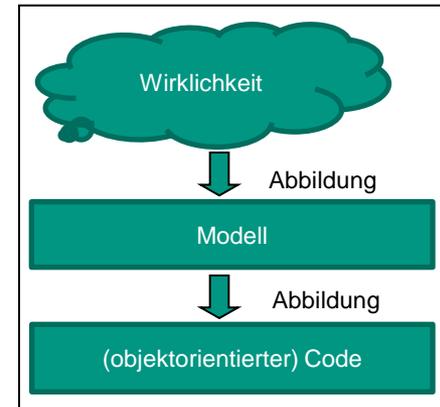
- Hinweis: In Java oder C# sind nicht all diese Variationen zulässig!

Hinweis für Java-Programmierer

Bezüglich Varianz lässt sich folgender „Vererbungsbaum“ der primitiven Typen angeben:



“



siehe: The Java Language Specification 2.0, §5.1.2, Widening Primitive Conversion

Polymorphie

- Vielgestaltigkeit
- „statisch“ (**Überladen**)
 - Es kann mehrere Methoden mit dem gleichen Namen geben (Signatur* muss unterschiedlich sein, damit der Compiler weiß, welche er gerade zu verwenden hat)
- „dynamisch“ („Verwendung der **Vererbung**“)
 - Es wird diejenige Methode mit der angegebenen Signatur aufgerufen, die in der Vererbungshierarchie (von der Klasse der aktuellen Instanz aus gesehen) am speziellsten ist



Hat nichts
mit OO oder
Vererbung
zu tun!

* im Falle von Java muss die Parameterliste unterschiedlich sein, im Falle von Haskell der Rückgabetyt

Beispiel „Dynamische Polymorphie“

```
Hund h;  
Wolf w = new Wolf();
```

```
h = w;  
h.heulen();
```

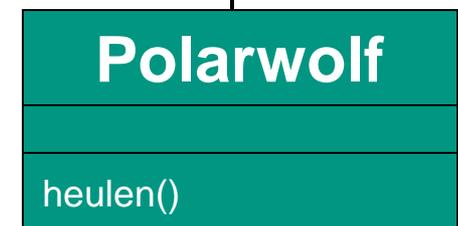
:Wolf



Canis



Canis lupus



Canis lupus arctos

Beispiel „Dynamische Polymorphie“

```
Hund h;  
Wolf w = new Wolf();
```

```
h = w;  
h.heulen();
```

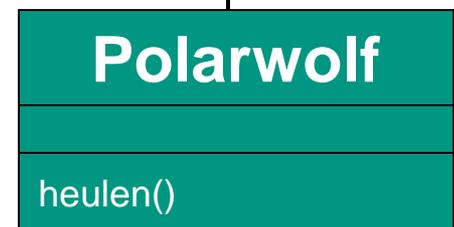
:Wolf



Canis



Canis lupus



Canis lupus arctos

Beispiel „Dynamische Polymorphie“

```
Hund h;  
Wolf w = new Wolf();
```

```
h = w;  
h.heulen();
```

← gültig wegen
is-a-Semantik

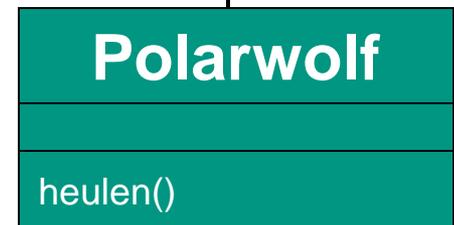
:Wolf



Canis



Canis lupus

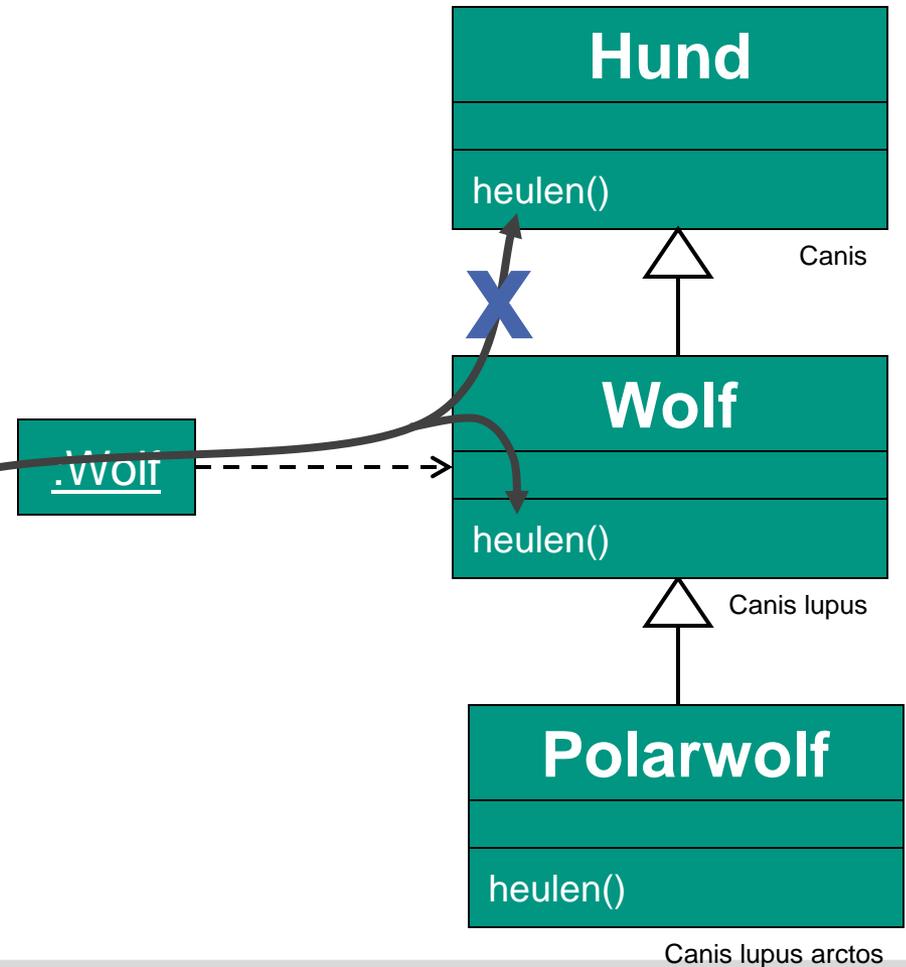


Canis lupus arctos

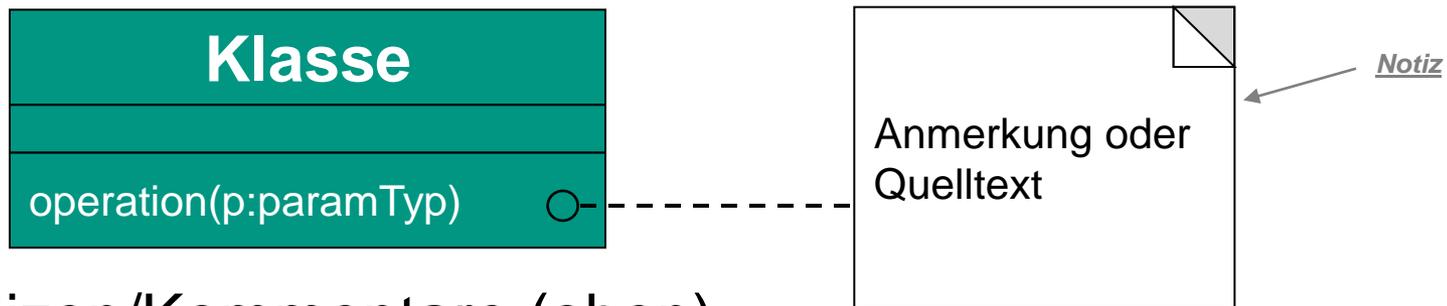
Beispiel „Dynamische Polymorphie“

```
Hund h;  
Wolf w = new Wolf();
```

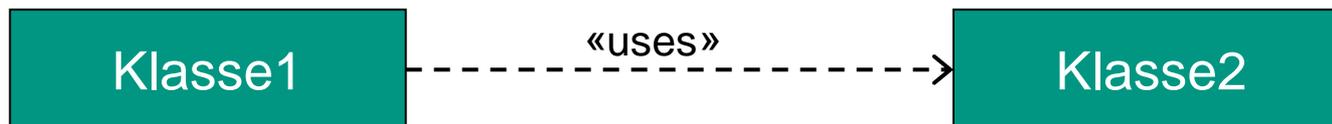
```
h = w;  
h.heulen();
```



Was es beim Klassendiagramm noch gibt...



- Notizen/Kommentare (oben)
- Abhängigkeit (unten, hier: Klasse1 hängt von Klasse2 ab, z.B. weil sie Klasse2 als Parameter, lokale Variable oder Rückgabewert verwendet)



Sichtbarkeit/„Zugriffsschutz“

- Die Attribute und Methoden eines Objektes lassen sich vor dem Zugriff durch andere Objekte schützen. Zugriff erlaubt bei:
 - **privat** („-“): nur (aber alle!) Exemplare der selben Klasse
 - **geschützt** („#“): Exemplare der selben Klasse und aller abgeleiteten Klassen, sowie Exemplare aus dem gleichen Paket (Subsystem/Bibliothek)
 - **öffentlich** („+“): jedes Exemplar
 - Das entsprechende Symbol wird einfach dem Attribut- bzw. Methodennamen vorangestellt.
- Beachte: „privat“ heißt nicht „nur das Exemplar selbst“!
- Der „Schutz“ beschränkt sich auf den Zeitpunkt des Übersetzens
 - Sämtliche Zugriffsrechte können über Code-Manipulation (z.B. mit BCEL für Java) geändert werden

Beispiel zu Sichtbarkeit

Kreis
-radius : double; #x: double = 0; #y : double = 0;
+setzeRadius(r: double) +setzeZentrum(p:Punkt) +anzeigen() +verstecken() +vergrößern(faktor : double)

Keine Angabe der Sichtbarkeit würde bedeuten: öffentlich

Literatur

- Bernd Oestereich, „Analyse und Design mit UML 2.x – Objektorientierte Softwareentwicklung“
- UML-Spezifikation unter <http://www.uml.org/#UML2.0>
 - Speziell „UML 2.0 Superstructure Specification“
 - und „UML 2.0 Infrastructure Specification“, insbes. Ch. 4 Terms and Definitions

Übung

1. Wie heißen Klassenattribute und -methoden in Java? Gibt es einen Unterschied zu UML?
2. Wie kann Multiplizität die folgenden Eigenschaften von Relationen definieren: linkstotal, rechtstotal (surjektiv), rechtseindeutig, linkseindeutig, injektiv, bijektiv, funktional?
3. Wie unterscheiden sich Assoziation und Vererbung? (Hinweis: über welche UML-Elemente sagen Assoziation bzw. Vererbung primär etwas aus?)
4. Was darf in Java Schnittstellen (interfaces) vorkommen? Gibt es einen Unterschied zu UML Schnittstellen?
5. Angenommen es gäbe Schnittstellen, aber in normalen Klassen keine abstrakten Methoden. Wäre das ein Nachteil?
6. Welche Art von Parameter-Varianz ist in Java erlaubt? (Achtung: Rückgabetyt nicht vergessen)