



Vienna University of Technology

# Objektorientierte Modellierung

## Strukturmodellierung



**Business Informatics Group**

*Institute of Software Technology and Interactive Systems  
Vienna University of Technology*

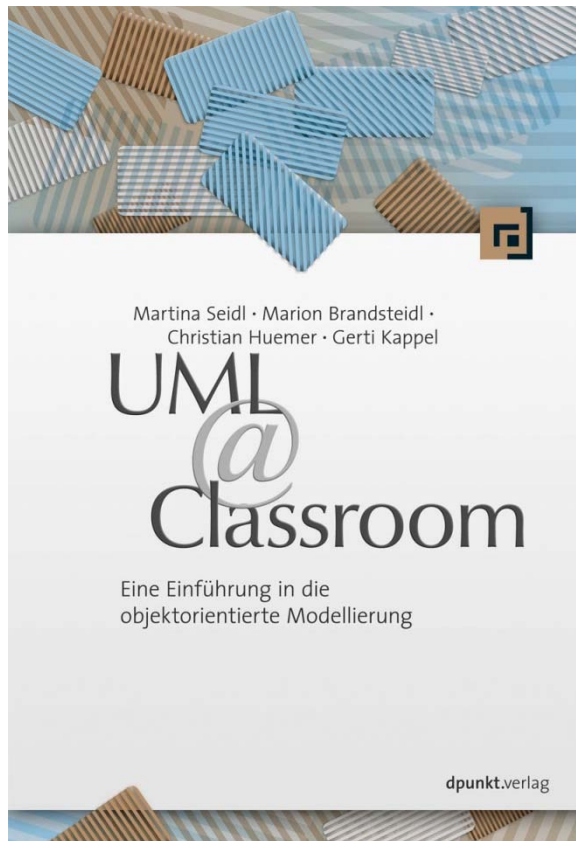
*Favoritenstraße 9-11/188-3, 1040 Vienna, Austria*

*phone: +43 (1) 58801-18804 (secretary), fax: +43 (1) 58801-18896  
office @big.tuwien.ac.at, www.big.tuwien.ac.at*

# Literatur

---

- Die Vorlesung basiert auf folgendem Buch:



**UML @ Classroom:**  
**Eine Einführung in die objekt-  
orientierte Modellierung**  
*Martina Seidl, Marion Brandsteidl,  
Christian Huemer und Gerti Kappel*

dpunkt.verlag

Juli 2012

ISBN 3898647765

- *Anwendungsfalldiagramm*
- **Strukturmodellierung**
- *Zustandsdiagramm*
- *Sequenzdiagramm*
- *Aktivitätsdiagramm*

# Inhalt

---

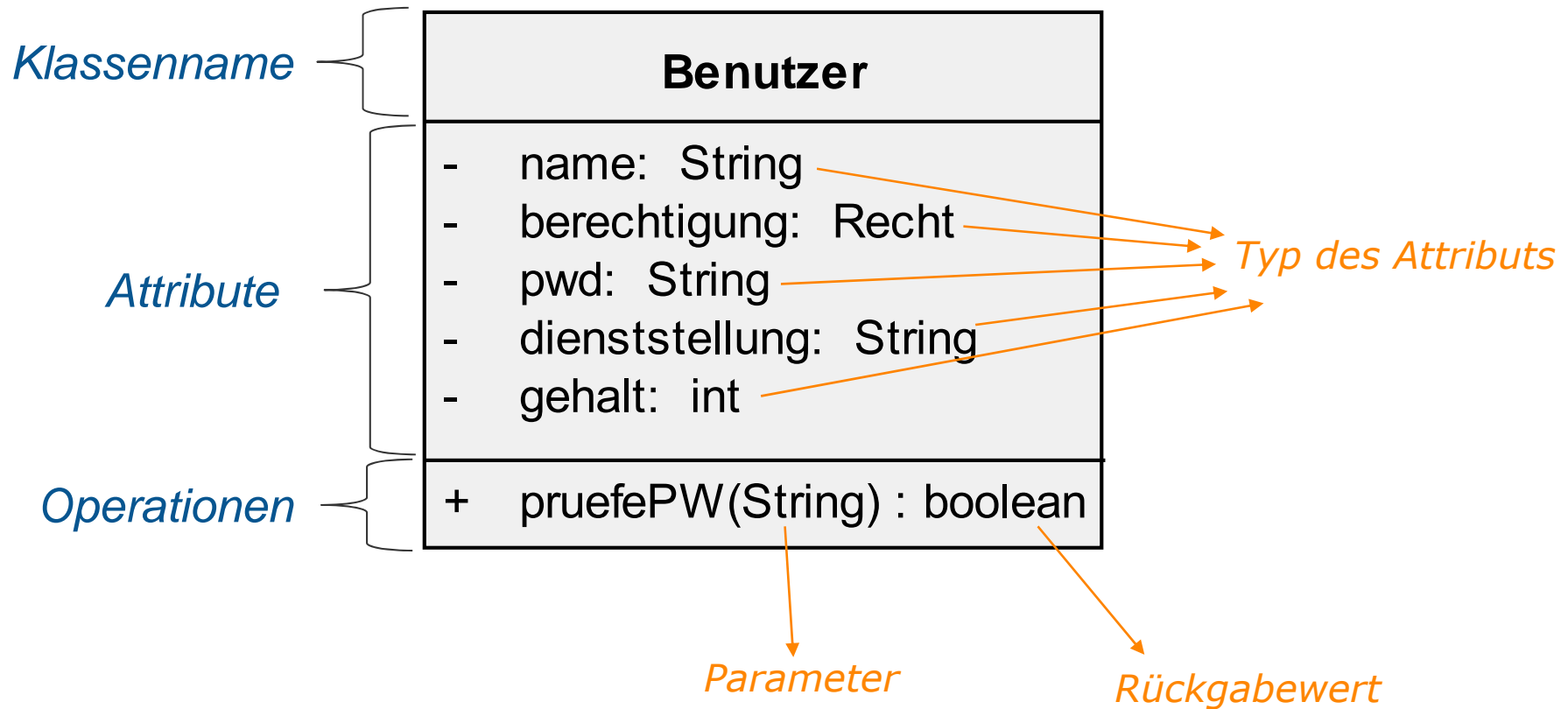
- **Klassendiagramm**
  - Klassen
  - Attribute und Operationen
  - Assoziationen
  - Schwache Aggregation
  - Starke Aggregation
  - Generalisierung
  - Zusammenfassendes Beispiel
  - Übersetzung nach Java
  - Datentypen in UML
- Objektdiagramm
- Paketdiagramm
- Abhängigkeiten

# Klassendiagramm

---

- **Klasse in UML:** Schablone, Typ
- **Objekt:** Ausprägung einer Klasse
- **Klassendiagramm**
  - Beschreibt den **strukturellen Aspekt** eines Systems auf **Typebene** in Form von Klassen, Interfaces und Beziehungen

# Notation für Klassen



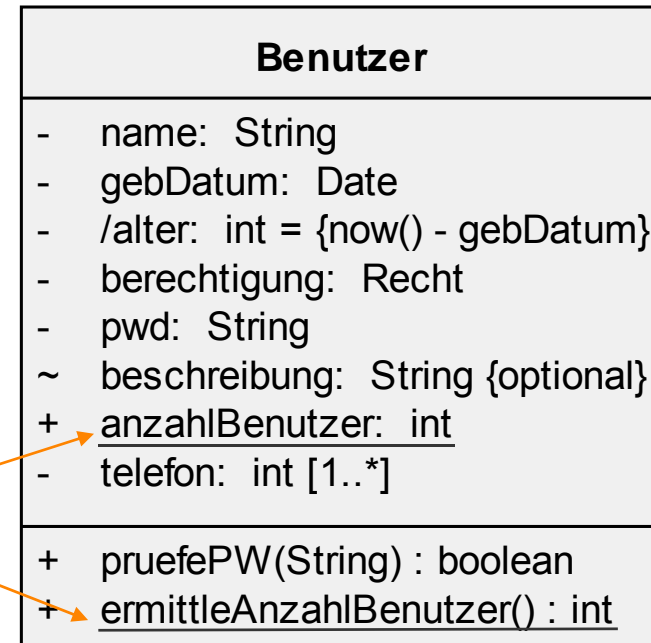
# Attribute und Operationen

- **Sichtbarkeiten** von Attributen und Operationen:

- + ... public
- - ... private
- # ... protected
- ~ ... package (vgl. Java)

- **Eigenschaften von Attributen:**

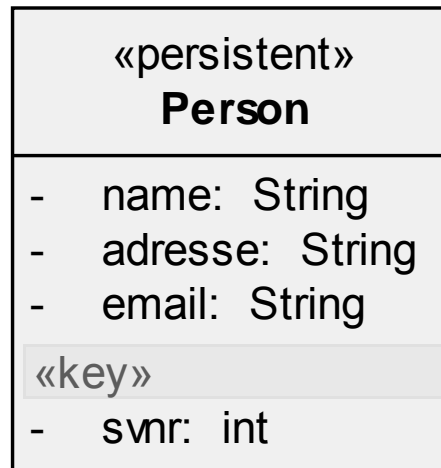
- „/“ attributname: abgeleitetes Attribut
  - Bsp.: /alter:int
- {optional}: Nullwerte sind erlaubt
- [n..m]: Multiplizität



*Klassenattribute/-operationen*

# Erweiterung von UML zur Datenmodellierung

---



## Exkurs: Identifikation von Klassen

---

- Linguistische Analyse der Problembeschreibung nach R.J. Abbott, Program Design by Informal English Descriptions, CACM, Vol. 26, No. 11, 1983
- **Hauptwörter** herausfiltern
- **Faustregeln**
  - Eliminierung von irrelevanten Begriffen
  - Entfernen von Namen von Ausprägungen
  - Beseitigung vager Begriffe
  - Identifikation von Attributen
  - Identifikation von Operationen
  - Eliminierung von Begriffen, die zu Beziehungen aufgelöst werden können





## Exkurs: Identifikation von Attributen

---

- **Adjektive** und **Mittelwörter** herausfiltern
- **Faustregeln**
  - Attribute beschreiben Objekte und sollten weder klassenwertig noch mehrwertig sein
  - abgeleitete Attribute sollten als solche gekennzeichnet werden
  - kontextabhängige Attribute sollten eher Assoziationen zugeordnet werden als Klassen
- Attribute sind i.A. nur unvollständig in der Anforderungsbeschreibung definiert

# Exkurs: Identifikation von Operationen

---

- **Verben** herausfiltern
  
- **Faustregeln**
  - Welche Operationen kann man mit einem Objekt ausführen?
  - Nicht nur momentane Anforderungen berücksichtigen, sondern Wiederverwendbarkeit im Auge behalten
  - Welche Ereignisse können eintreten?
  - Welche Objekte können auf diese Ereignisse reagieren?
  - Welche anderen Ereignisse werden dadurch ausgelöst?

## Bsp. - Bibliotheksverwaltung

---

Franz Müller soll neben anderen Leuten die Bibliothek der Universität benutzen können. Im Verwaltungssystem werden die **Benutzer** erfasst, von denen eine **eindeutige ID**, **Name** und **Adresse** bekannt sind, und die **Bücher**, von denen **Titel**, **Autor** und **ISBN-Nummer** gespeichert sind.

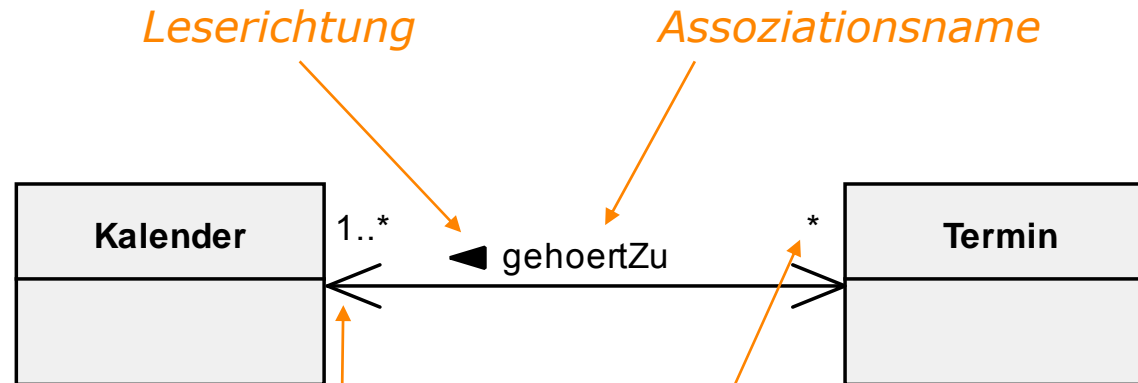
<b>Buch</b>	
+	Titel: String
+	Autor: String
+	ISBN: int

<b>Benutzer</b>	
+	ID: int
+	Name: String
+	Adresse: String

**Frage: Was ist mit Franz Müller?**

# Assoziation

- Assoziationen zwischen Klassen modellieren mögliche **Objektbeziehungen (Links)** zwischen den **Instanzen der Klassen**



*Navigationsrichtung*

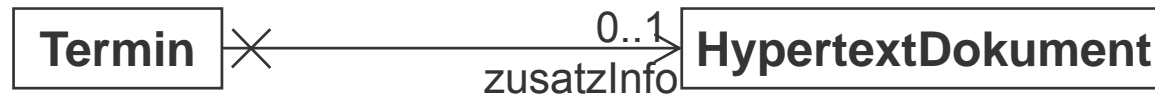
*Multiplizität*

Mögliche Anzahl jener Objekte, die mit genau einem Objekt der gegenüberliegenden Seite in Beziehung stehen können

## Assoziation: Navigationsrichtung

---

- Eine gerichtete Kante gibt an, **in welche Richtung die Navigation** von einem Objekt zu seinem Partnerobjekt **erfolgen kann**
- Ein **nicht-navigierbares Assoziationsende** wird durch ein "X" am Assoziationsende angezeigt



- Navigation von einem bestimmten Termin zum entsprechenden Dokument
- Umgekehrte Richtung - welche Termine beziehen sich auf ein bestimmtes Dokument? - wird nicht unterstützt
- **Ungerichtete Kanten** bedeuten "**keine Angabe** über Navigationsmöglichkeiten"
  - In Praxis wird oft bidirektionale Navigierbarkeit angenommen
- Die Angabe von Navigationsrichtungen stellt einen Hinweis für die spätere Entwicklung dar

## Assoziation als Attribut

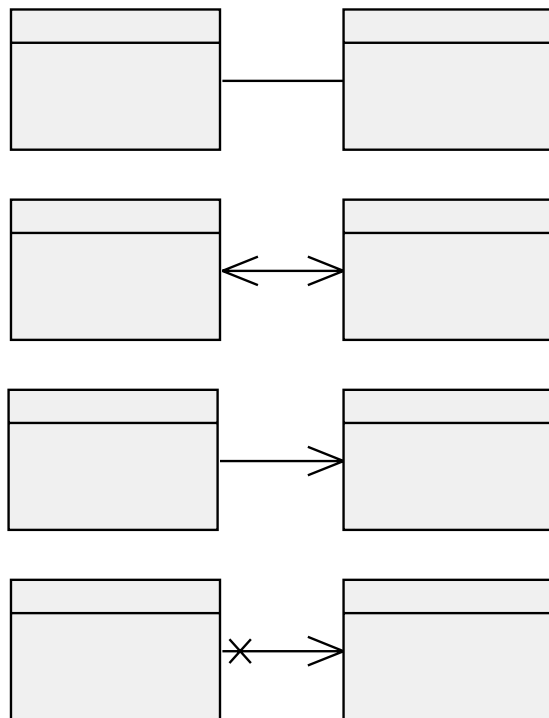
---

- Ein **navigierbares Assoziationsende** hat die gleiche Semantik wie ein Attribut der Klasse am gegenüberliegenden Assoziationsende
- Ein navigierbares Assoziationsende kann daher **anstatt** mit einer **gerichteten Kante** auch als **Attribut** modelliert werden
  - Die mit dem Assoziationsende verbundene Klasse muss dem **Typ** des Attributs entsprechen
  - Die **Multiplizitäten** müssen gleich sein
- Für ein navigierbares Assoziationsende sind somit alle Eigenschaften und Notationen von Attributen anwendbar



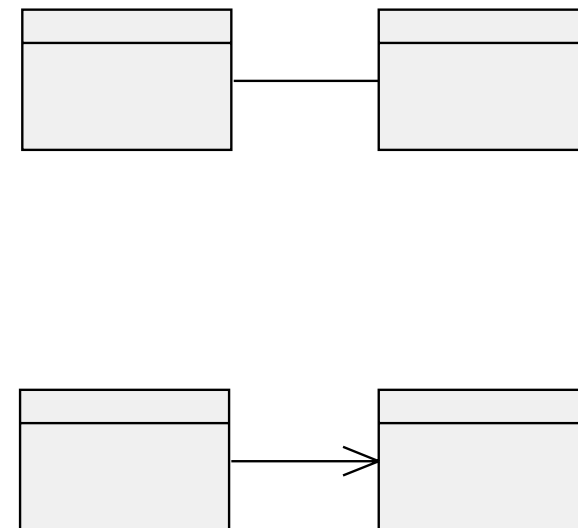
# Assoziation: Beispiele – UML Standard vs. Best Practice

*UML Standard*



↔

*Best Practice*



## Assoziation: Multiplizität

---

- **Bereich:** "min .. max"
- Beliebige **Anzahl:** "\*" (= 0.. \*)
- **Aufzählung** möglicher Kardinalitäten (durch Kommas getrennt)
- **Defaultwert:** 1

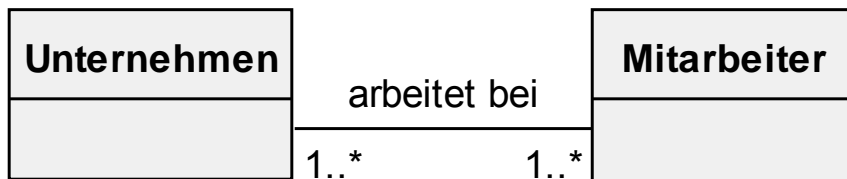
<i>genau 1:</i>	1
<i>&gt;= 0:</i>	* oder 0..*
<i>0 oder 1:</i>	0..1 oder 0,1
<i>fixe Anzahl (z.B. 3):</i>	3
<i>Bereich (z.B. &gt;= 3):</i>	3..*
<i>Bereich (z.B. 3 - 6):</i>	3..6
<i>Aufzählung:</i>	3,6,7,8,9 oder 3, 6..9



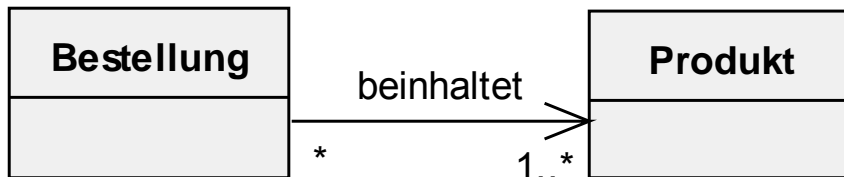
## Assoziation: Beispiele



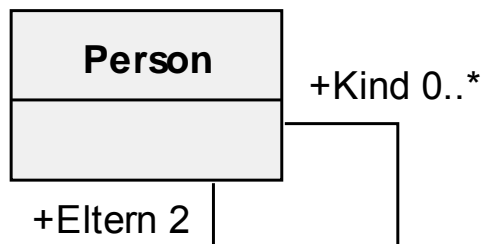
Ein Auto hat genau einen Besitzer, eine Person kann aber mehrere Autos besitzen (oder keines).



In einem Unternehmen arbeitet mind. ein Mitarbeiter, ein Mitarbeiter arbeitet mind. in einem Unternehmen



Eine Bestellung besteht aus 1-n Produkten, Produkte können beliebig oft bestellt werden. Von einer Bestellung kann festgestellt werden, welche Produkte sie beinhaltet.

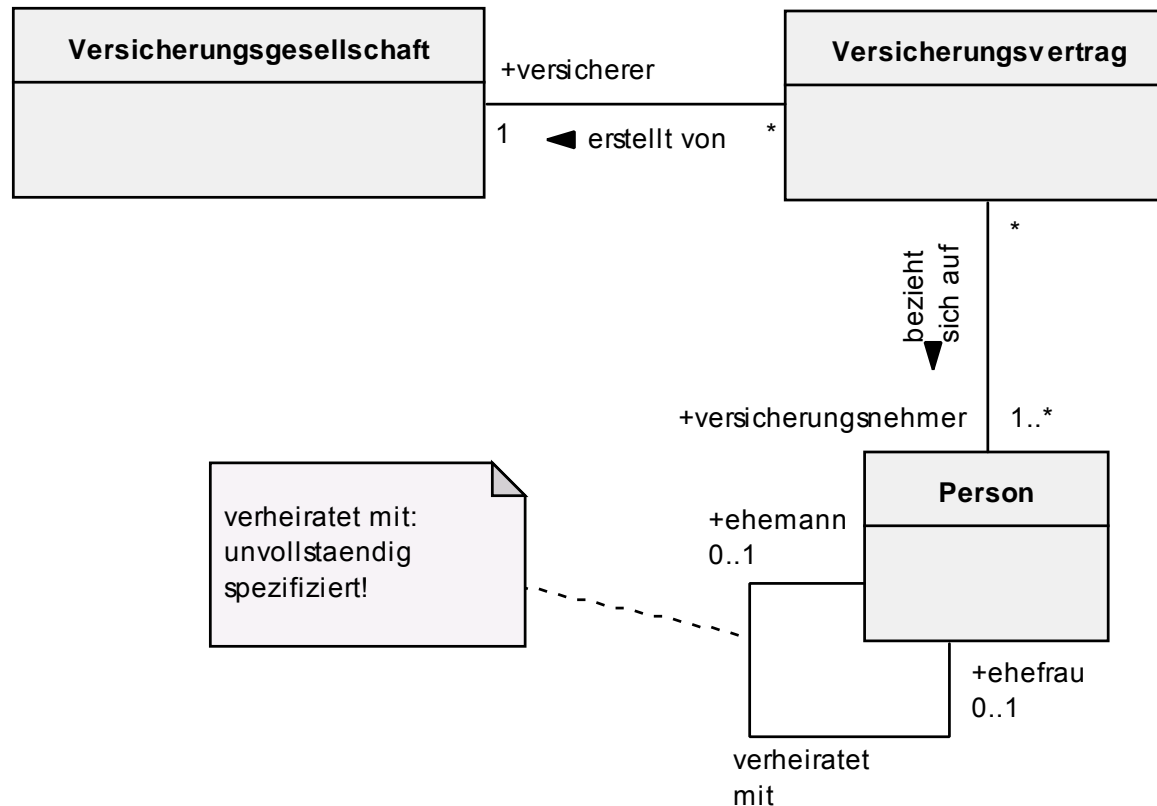


Eine Person hat 2 Eltern, die Personen sind, und 0 bis beliebig viele Kinder.  
*Ist durch dieses Modell ausgeschlossen, dass eine Person Kind von sich selbst ist?*



# Assoziation: Rollen

- Es können die **Rollen** festgelegt werden, die von den einzelnen Objekten in den Objektbeziehungen gespielt werden



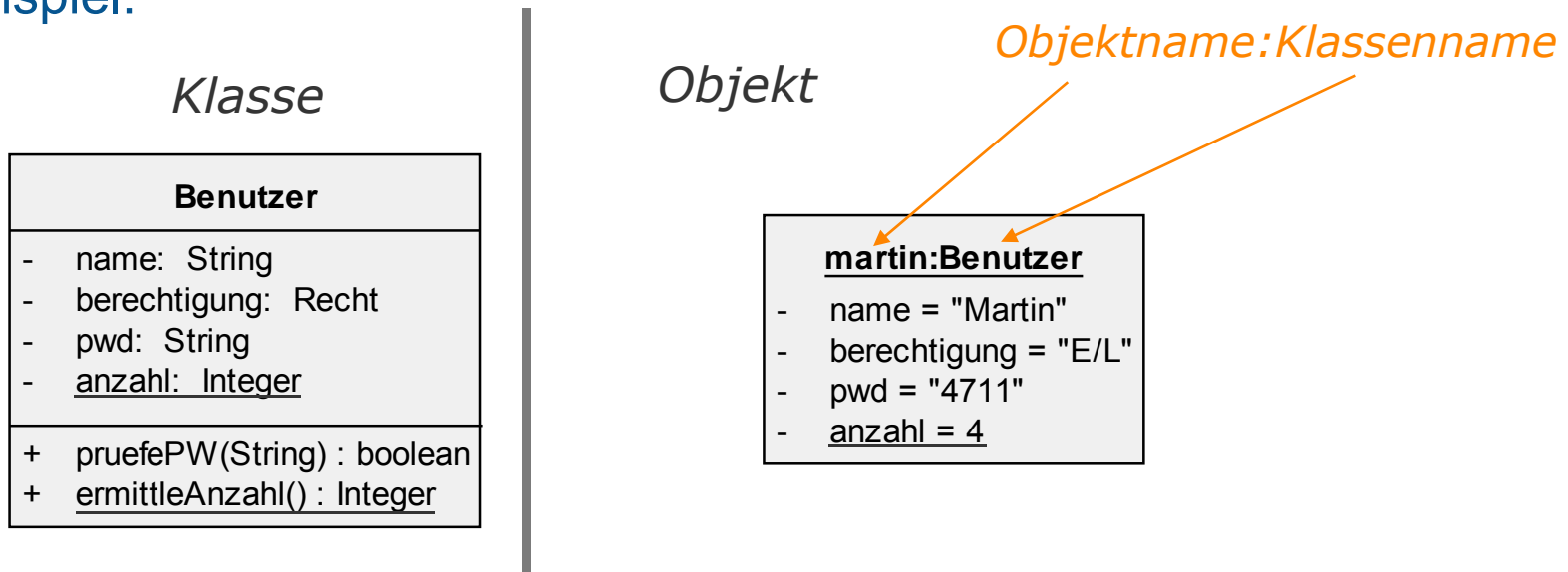
# Inhalt

---

- **Klassendiagramm**
  - Klassen
  - Attribute und Operationen
  - Assoziationen
  - Schwache Aggregation
  - Starke Aggregation
  - Generalisierung
  - Zusammenfassendes Beispiel
  - Übersetzung nach Java
  - Datentypen in UML
- **Objektdiagramm**
- **Paketdiagramm**
- **Abhängigkeiten**

# Objektdiagramm

- Beschreibt den strukturellen Aspekt eines Systems auf Instanzebene in Form von Objekten und Links
- Momentaufnahme (snapshot) des Systems – konkretes Szenario
- Ausprägung zu einem Klassendiagramm
- Eigentlich eine »Instanzspezifikation«
- Prinzipiell kann jede Diagrammart auf Instanzebene modelliert werden
- Beispiel:



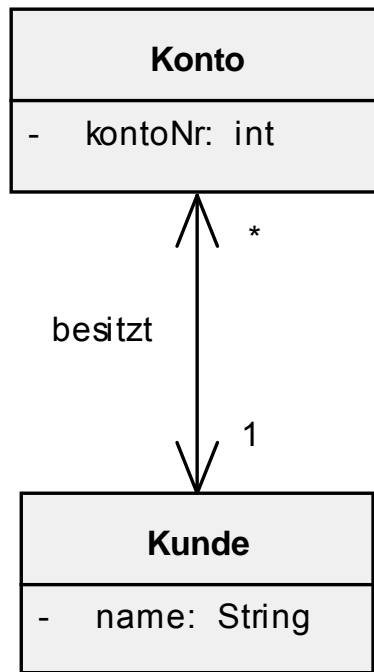
# Objektdiagramm: Basiskonzepte

---

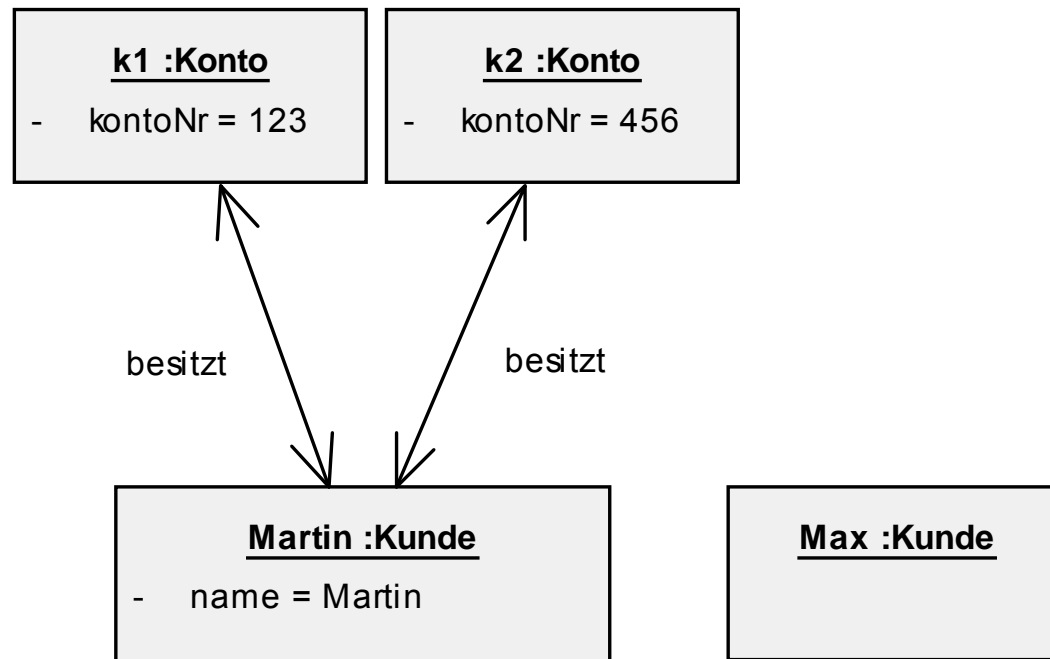
- **Basiskonzepte** des Objektdiagramms
  - Instanz einer Klasse: **Objekt**
  - Instanz einer Assoziation: **Link**
  - Instanz eines Datentyps: **Wert**
- **Einheitliche Notationskonventionen**
  - Gleiches Notationselement wie auf Typebene benutzen
  - Unterstreichen (bei Links optional)
- Objektdiagramm muss **nicht vollständig** sein
  - Z.B. können Werte benötigter Attribute fehlen, aber auch Instanzspezifikation abstrakter Klassen modelliert werden

# Objektdiagramm: Beispiel

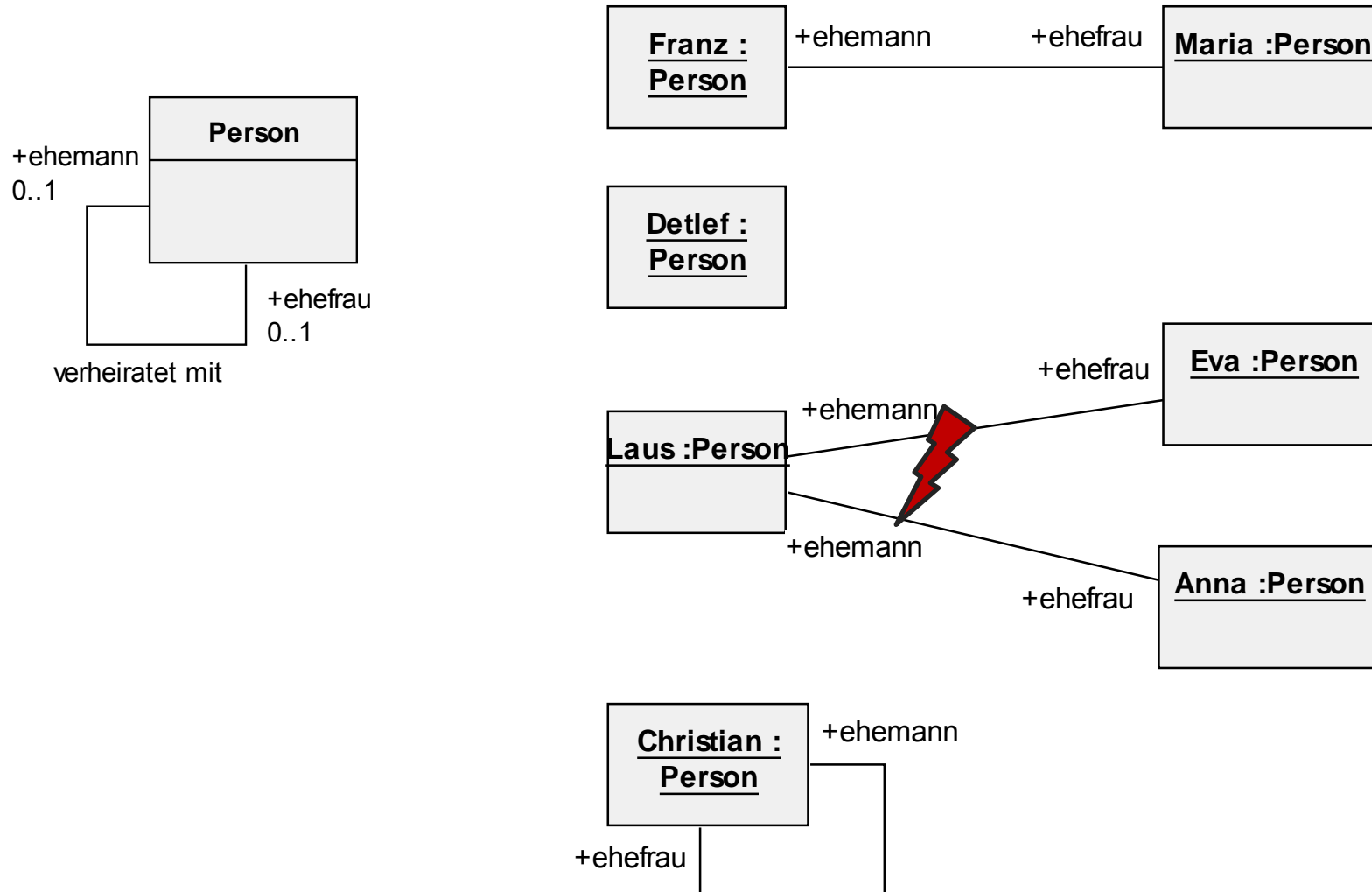
Klassendiagramm



Objektdiagramm



# Objektdiagramm: Beispiel bei unärer Assoziation



# Inhalt

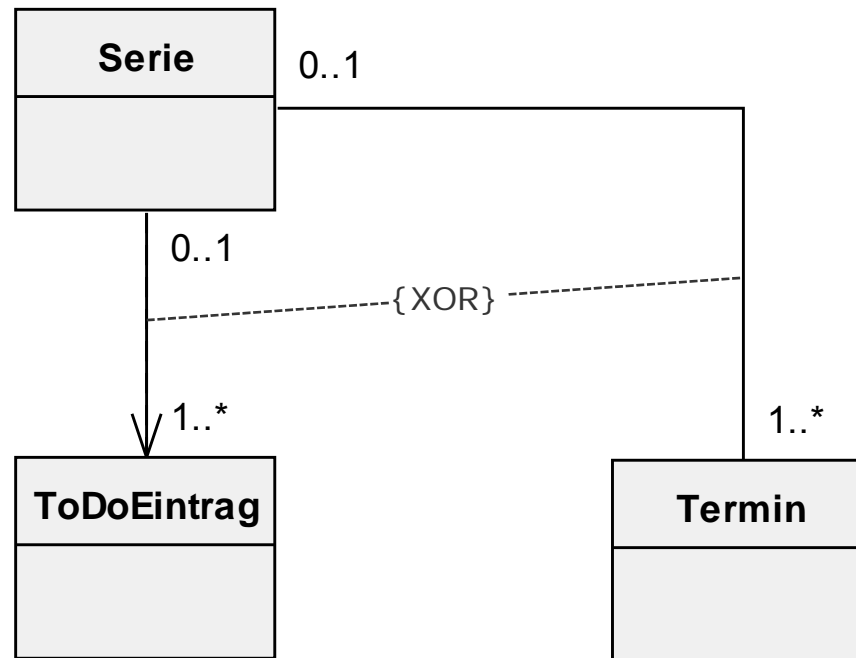
---

- Klassendiagramm
  - Klassen
  - Attribute und Operationen
  - **Assoziationen**
  - Schwache Aggregation
  - Starke Aggregation
  - Generalisierung
  - Zusammenfassendes Beispiel
  - Übersetzung nach Java
  - Datentypen in UML
- Objektdiagramm
- Paketdiagramm
- Abhängigkeiten



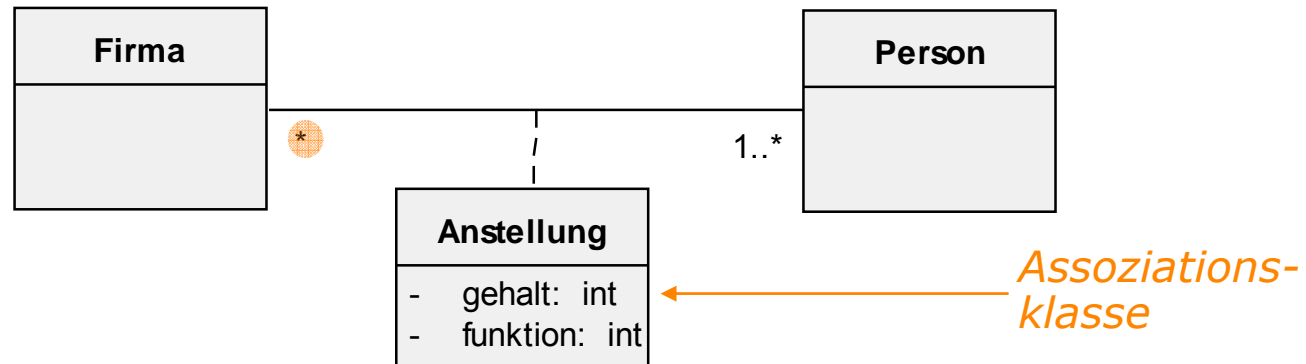
## Exklusive Assoziation

- Für ein bestimmtes Objekt kann zu einem bestimmten Zeitpunkt **nur eine von verschiedenen möglichen Assoziationen** instanziiert werden: **{xor}**

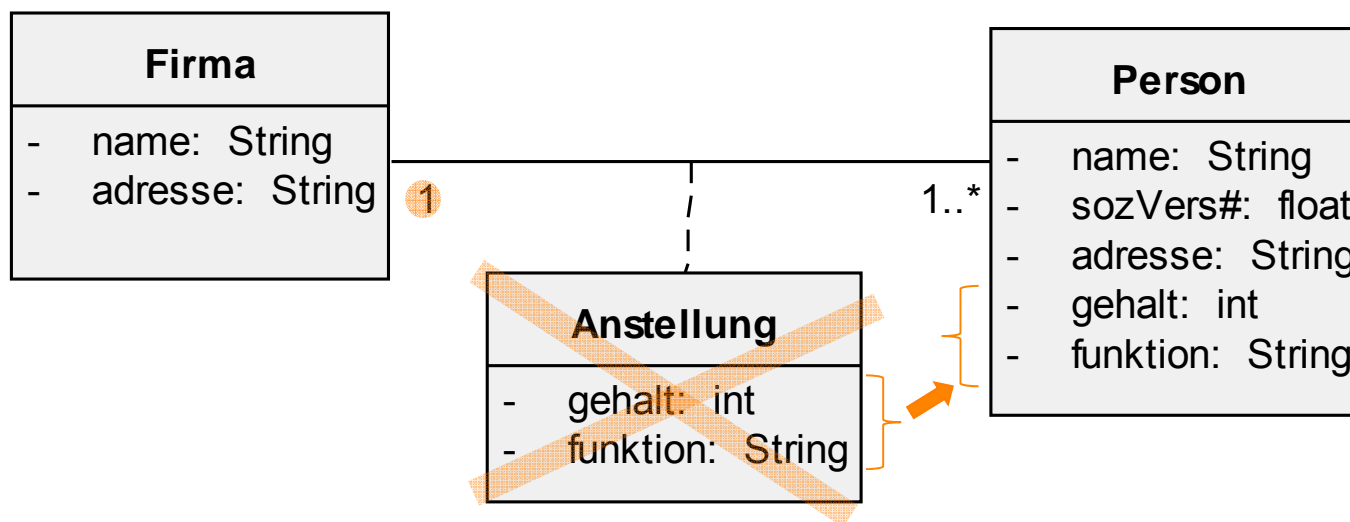


# Assoziationsklasse (1/2)

- Kann **Attribute der Assoziation** enthalten
  - Bei m:n-Assoziationen mit Attributen notwendig

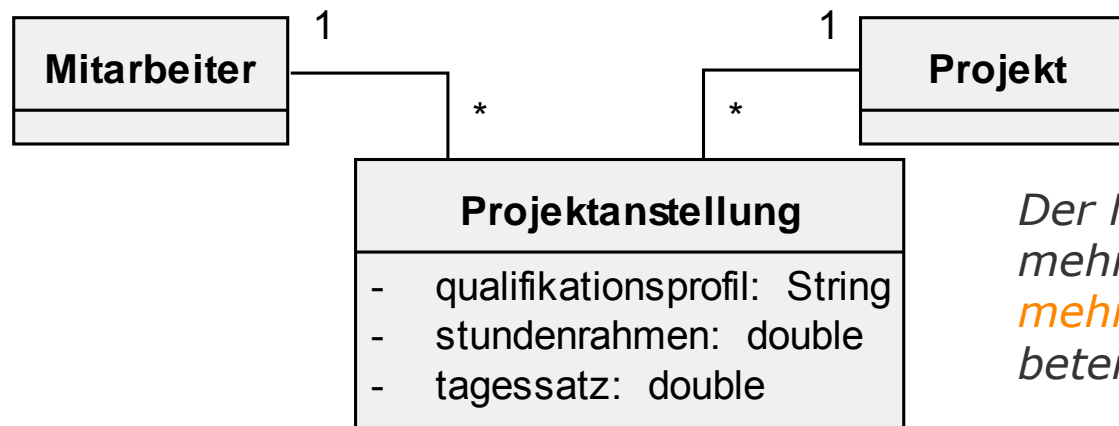


- Bei 1:1 und 1:n-Assoziationen sinnvoll aus Flexibilitätsgründen (Änderung der Multiplizität möglich)

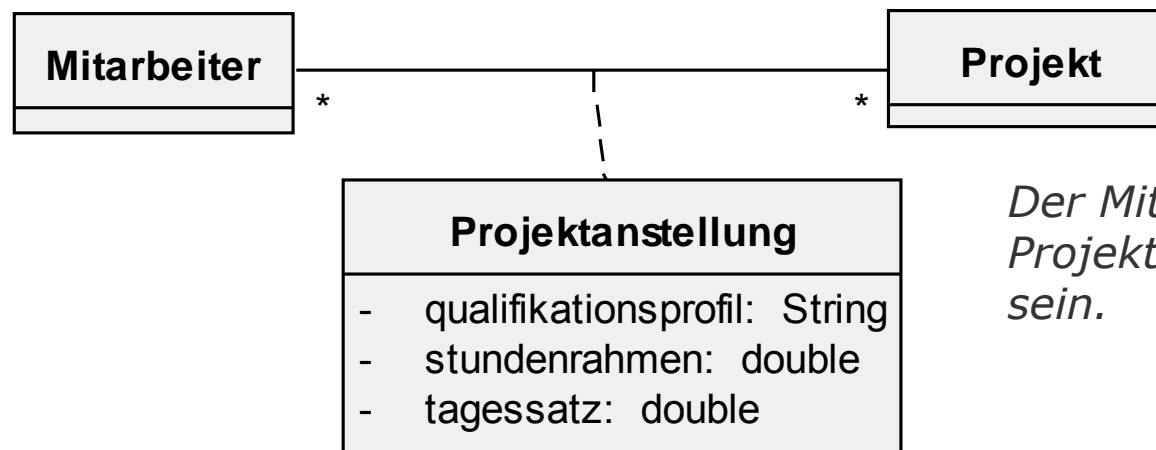


## Assoziationsklasse (2/2)

- Normale Klasse ungleich Assoziationsklasse



Der Mitarbeiter *M* kann über mehrere Projektanstellungen **mehrfach** an dem Projekt *P* beteiligt sein.



Der Mitarbeiter *M* kann an dem Projekt *P* nur **einmal** beteiligt sein.



## n-äre Assoziation (1/3)

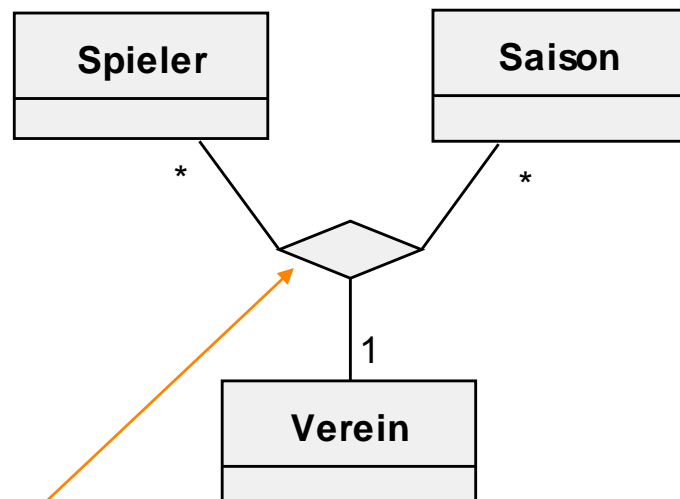
---

- **Beziehung zwischen mehr als zwei Klassen**
  - Navigationsrichtung kann nicht angegeben werden
  - Multiplizitäten geben an, wie viele Objekte einer Rolle/Klasse einem festen (n-1)-Tupel von Objekten der anderen Rollen/Klassen zugeordnet sein können
- **Multiplizitäten implizieren Einschränkungen**, in einem bestimmten Fall funktionale Abhängigkeiten
- Liegen (n-1) Klassen (z.B. A und B) einer Klasse (z.B. C) mit Multiplizität von max. 1 gegenüber, so existiert eine funktionale Abhängigkeit  $(A, B) \rightarrow (C)$

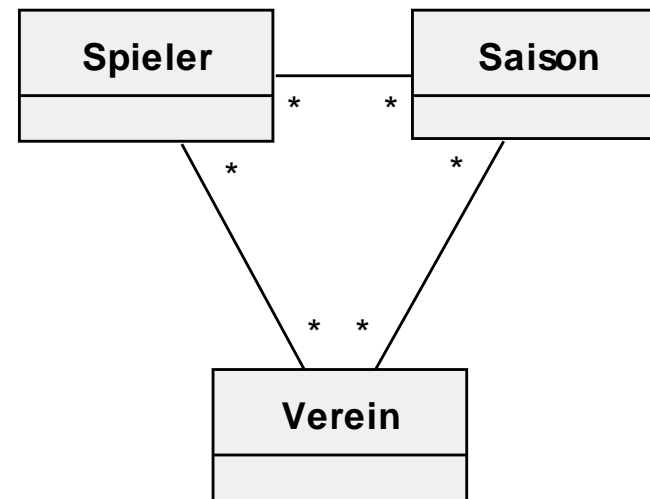
## n-äre Assoziation (2/3)

### ■ Beispiel

- (Spieler, Saison) → (Verein)
  - Ein Spieler spielt in einer Saison bei genau **einem** Verein
- (Saison, Verein) ✂ (Spieler)
  - In einer Saison spielen bei einem Verein **mehrere** Spieler
- (Verein, Spieler) ✂ (Saison)
  - Ein Spieler spielt in einem Verein in **mehreren** Saisonen

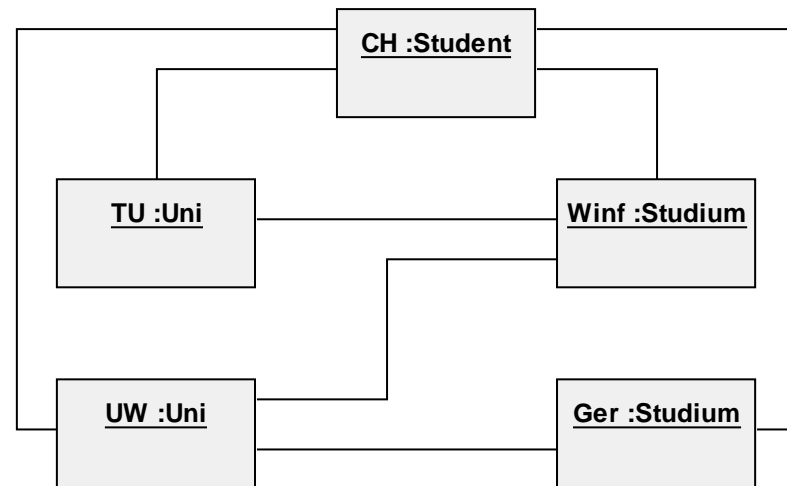
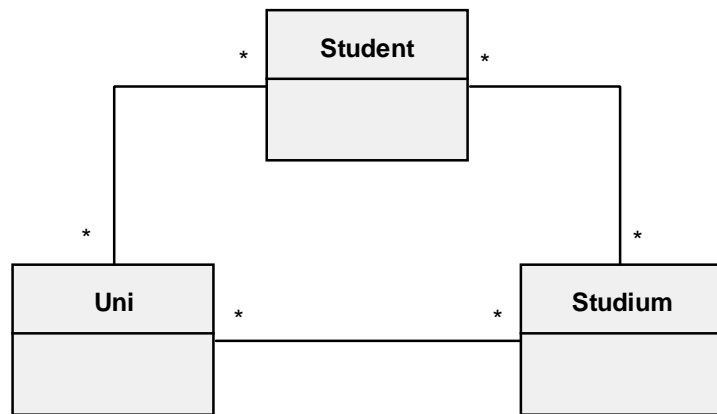
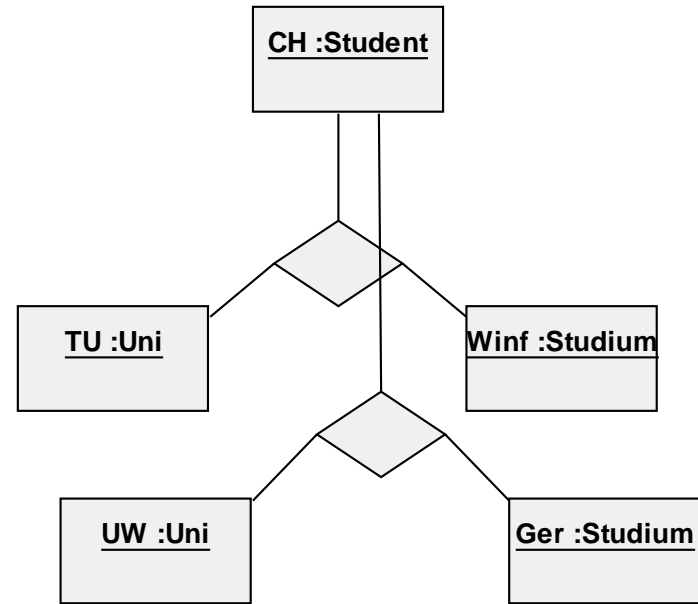
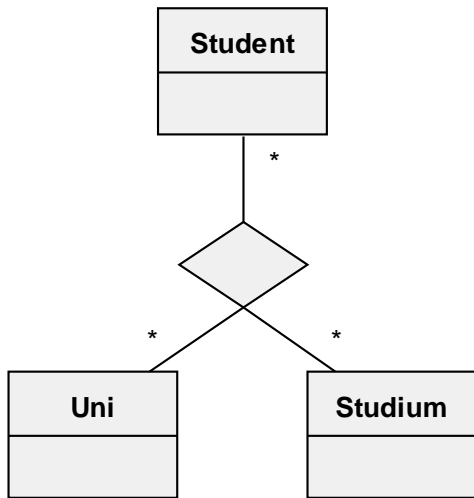


≠



*n-äre Assoziation  
(hier: ternär)*

# n-äre Assoziation (3/3)

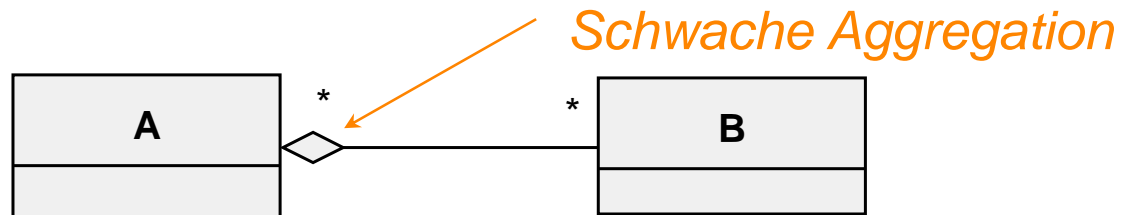


# Aggregation

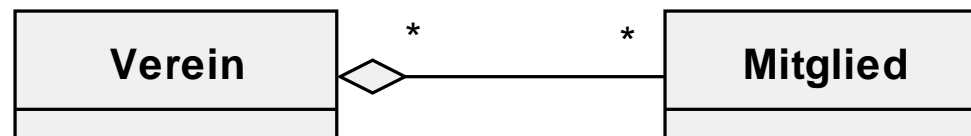
---

- Aggregation ist eine spezielle Form der Assoziation mit folgenden Eigenschaften:
  - Transitivität:  
C ist Teil von B u. B ist Teil von A  $\rightarrow$  C ist Teil von A
    - Bsp.: Kühlung = Teil von Motor & Motor = Teil von Auto  
 $\rightarrow$  Kühlung ist (indirekter) Teil von Auto
  - Anti-Symmetrie:  
B ist Teil von A  $\rightarrow$  A ist nicht Teil von B
    - Bsp.: Motor ist Teil von Auto, Auto ist nicht Teil von Motor
- UML unterscheidet zwei Arten von Aggregationen:
  - Schwache Aggregation (shared aggregation)
  - Starke Aggregation – Komposition (composite aggregation)

# Schwache Aggregation

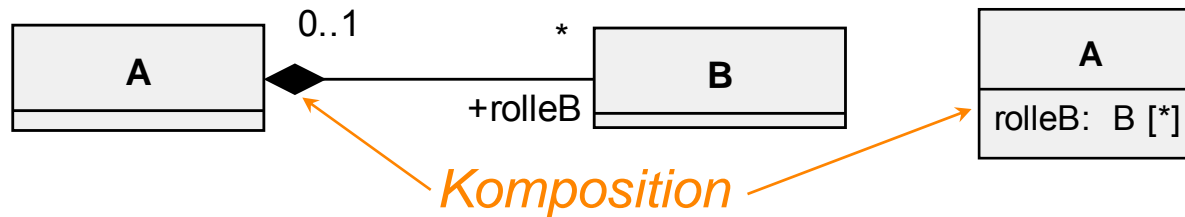


- Schwache Zugehörigkeit der Teile, d.h. Teile sind **unabhängig** von ihrem Ganzen
- Die Multiplizität des aggregierenden Endes der Beziehung (Raute) kann  $> 1$  sein
- Es gilt nur eingeschränkte **Propagierungssemantik**
- Die zusammengesetzten Objekte bilden einen **gerichteten, azyklischen Graphen**

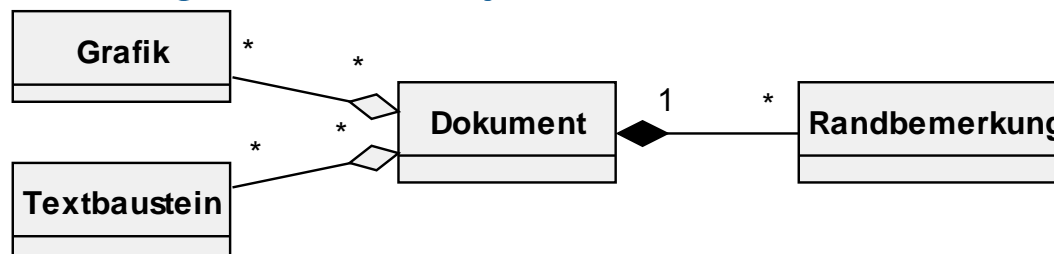




## Starke Aggregation (= Komposition)

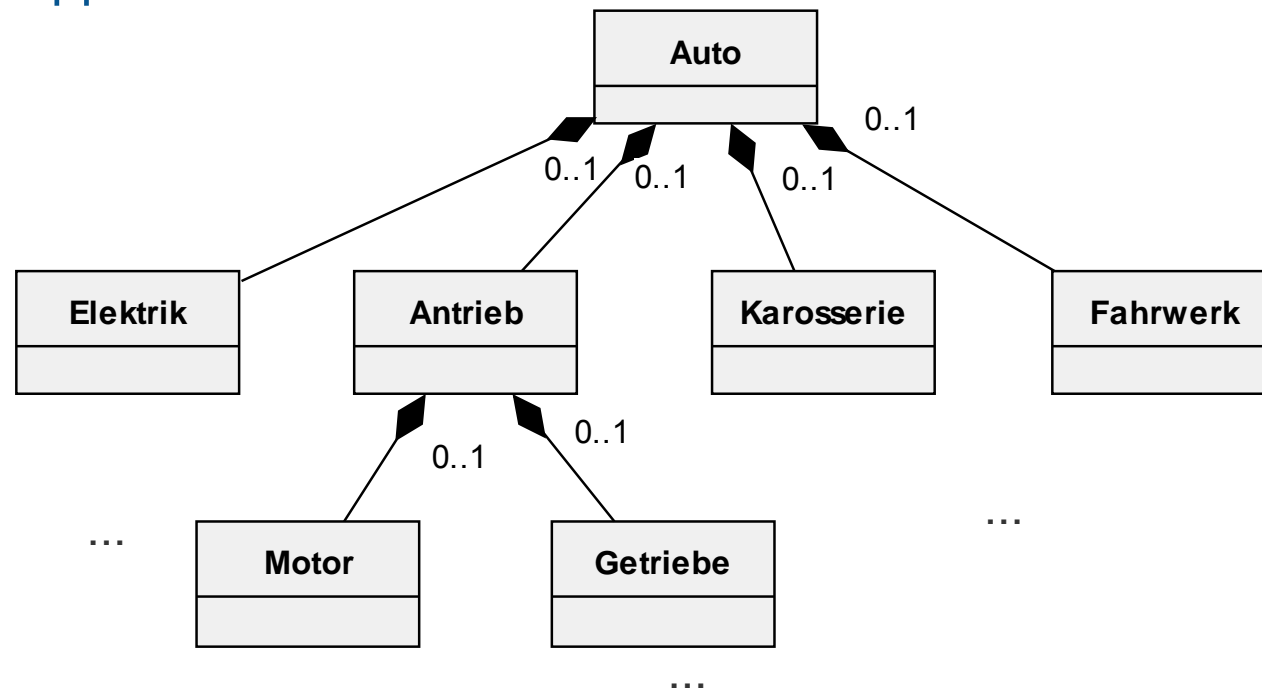


- Ein bestimmter Teil darf zu einem bestimmten Zeitpunkt in **maximal einem zusammengesetzten Objekt enthalten** sein
- Die **Multiplizität** des aggregierenden Endes der Beziehung kann (maximal) 1 sein
- **Abhängigkeit** der Teile vom zusammengesetzten Objekt
- **Propagierungssemantik**
- Die zusammengesetzten Objekte bilden einen **Baum**



# Starke Aggregation

- Mittels starker Aggregation kann eine Hierarchie von „Teil-von“-Beziehungen dargestellt werden (Transitivität!)
- Beispiel: Baugruppen von Auto



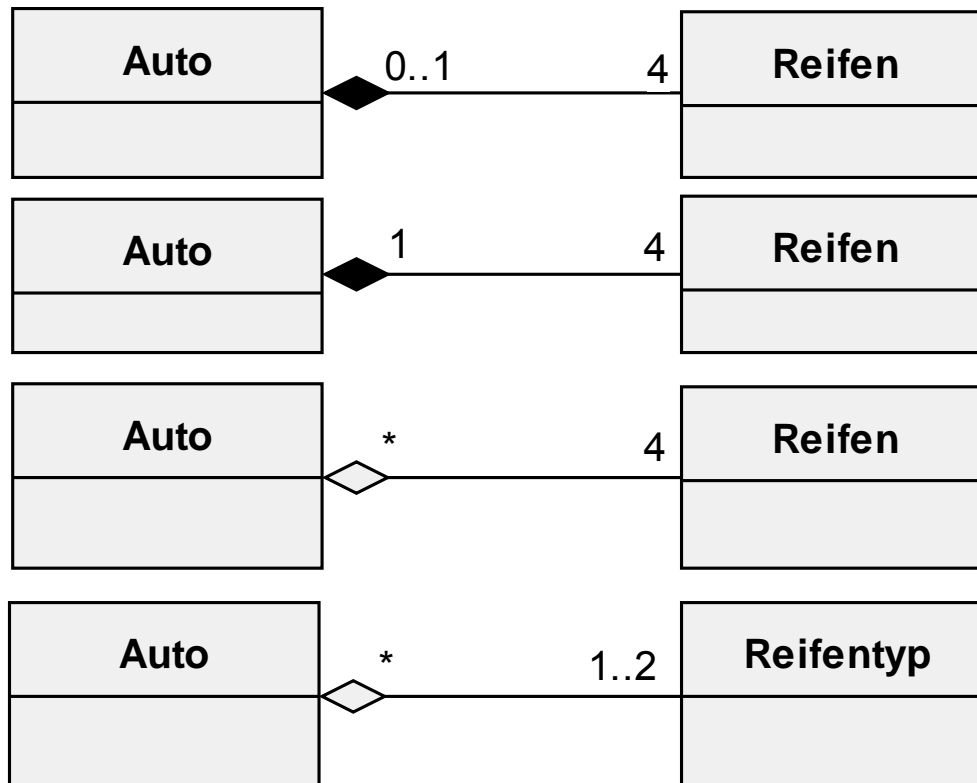
# Starke Aggregation vs. Assoziation - Faustregeln

---

- **Einbettung**
  - Die Teile sind i.A. physisch im Kompositum enthalten
  - Über Assoziation verbundene Objekte werden über Referenzen realisiert
- **Sichtbarkeit**
  - Ein Teil ist nur für das Kompositum sichtbar
  - Das über eine Assoziation verbundene Objekt ist i.A. öffentlich sichtbar
- **Lebensdauer**
  - Das Kompositum erzeugt und löscht seine Teile
  - Keine Existenzabhängigkeit zwischen assoziierten Objekten
- **Kopien**
  - Kompositum und Teile werden kopiert
  - Nur die Referenzen auf assoziierte Objekte werden kopiert

# Komposition und Aggregation

- Welche der folgenden Beziehungen trifft zu?



# Komposition und Aggregation

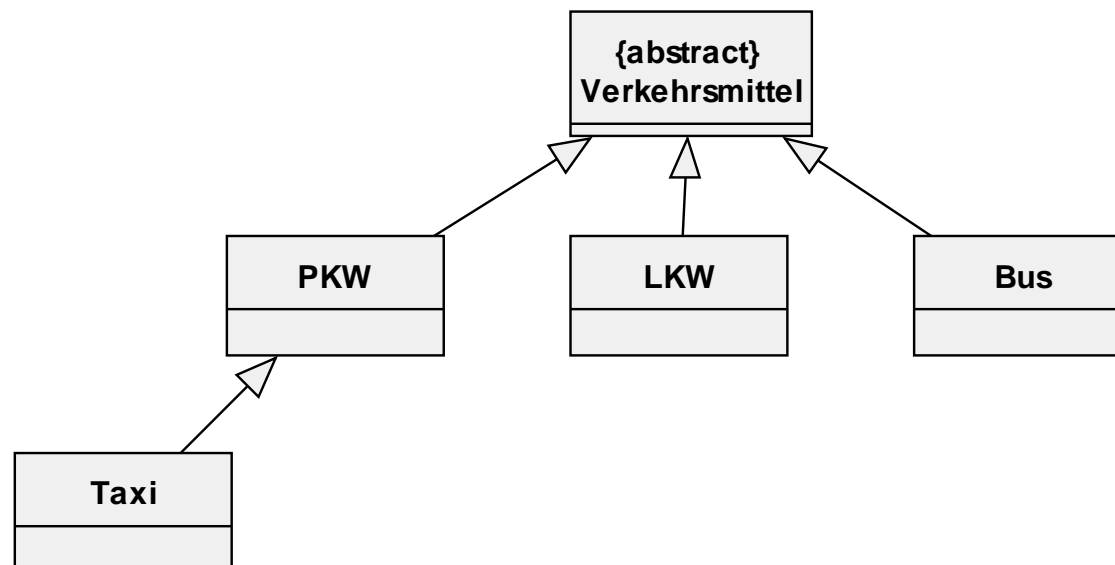
- Welche der folgenden Beziehungen trifft zu?

	<p>Ein Reifen kann auch ohne Auto existieren. Weiters gehört ein Reifen zu maximal einem Auto.</p>	<p>---JA---</p>
	<p>Ein Reifen kann nicht ohne Auto existieren.</p>	<p>---NEIN---</p>
	<p>Ein Reifen kann Bestandteil mehrerer Autos sein.</p>	<p>---NEIN---</p>
	<p>Ein Auto hat 1 oder 2 Typen von Reifen. Mehrere Autos können den selben Reifentyp haben.</p>	<p>---JA---</p>

# Generalisierung

---

- **Taxonomische Beziehung** zwischen einer spezialisierten Klasse und einer allgemeineren Klasse
  - Die spezialisierte Klasse **erbt** die Eigenschaften der allgemeineren Klasse
  - Kann **weitere Eigenschaften** hinzufügen
  - Eine **Instanz der Unterklasse** kann überall dort verwendet werden, wo eine **Instanz der Oberklasse** erlaubt ist (zumindest syntaktisch)
- Mittels Generalisierung wird eine Hierarchie von „ist-ein“- Beziehungen dargestellt (Transitivität!)



## Abstrakte Klasse (1/2)

---

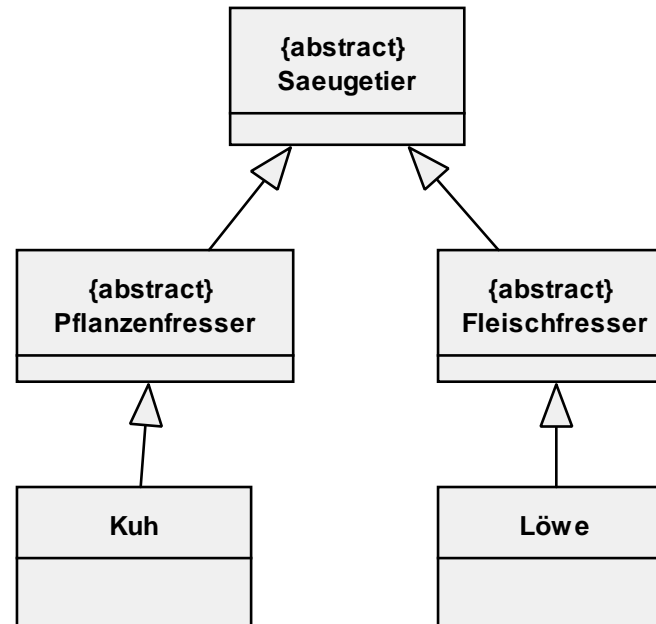
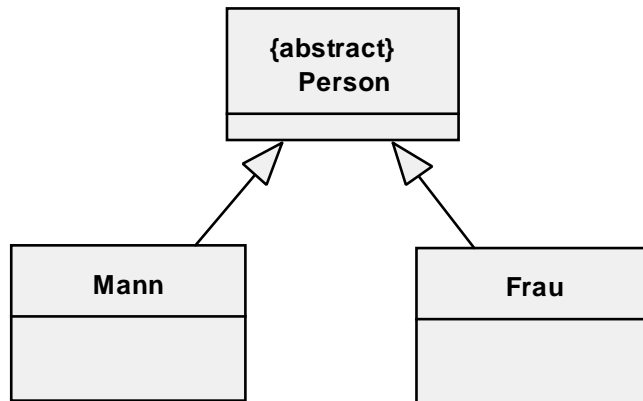
- Klasse, die **nicht instanziiert** werden kann
- **Nur in Generalisierungshierarchien** sinnvoll
- Dient zum "**Herausheben**" **gemeinsamer Merkmale** einer Reihe von Unterklassen
- Notation: Schlüsselwort {abstract} oder Klassenname in kursiver Schrift



- Mit analoger Notation wird zwischen konkreten (= implementierten) und abstrakten (= nur spezifizierten) **Operationen** einer Klasse unterschieden

# Abstrakte Klasse (2/2)

- Beispiele:



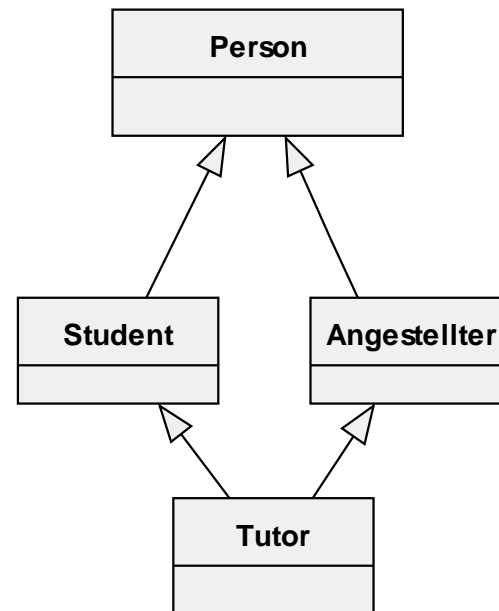


# Mehrfachvererbung

---

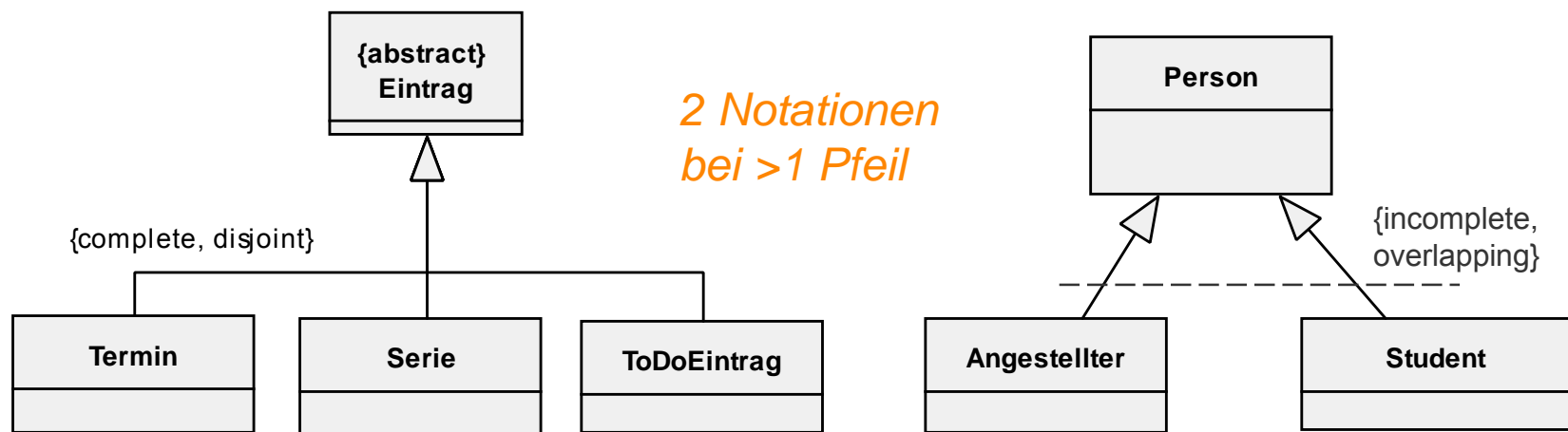
- Klassen müssen nicht nur eine Oberklasse haben, sondern können auch von mehreren Klassen erben

- Beispiel:



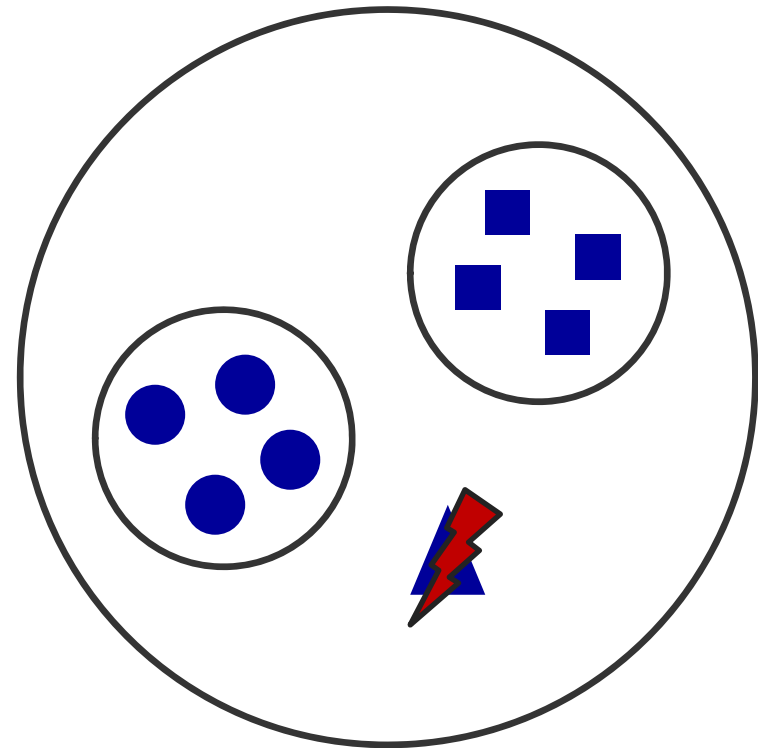
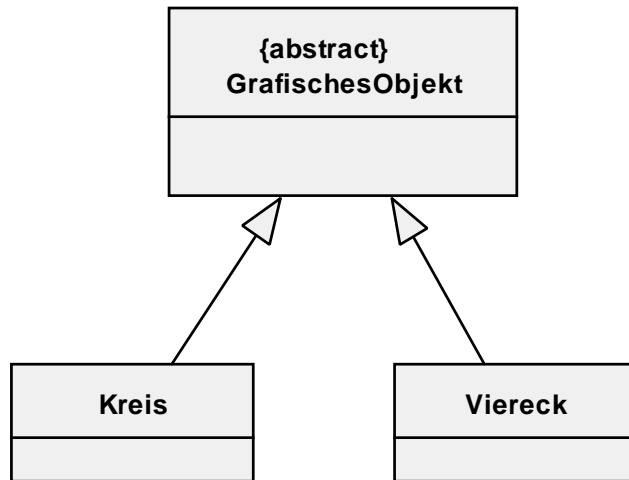
## Generalisierung: Eigenschaften (1/2)

- Unterscheidung kann vorgenommen werden in
  - **Unvollständig / vollständig:**  
In einer vollständigen Generalisierungshierarchie muss jede Instanz der Superklasse auch Instanz mindestens einer Subklasse sein
  - **Überlappend / disjunkt:**  
In einer überlappenden Generalisierungshierarchie kann ein Objekt Instanz von mehr als einer Subklasse sein
  - **Default:** unvollständig, disjunkt



## Generalisierung: Eigenschaften (2/2)

---



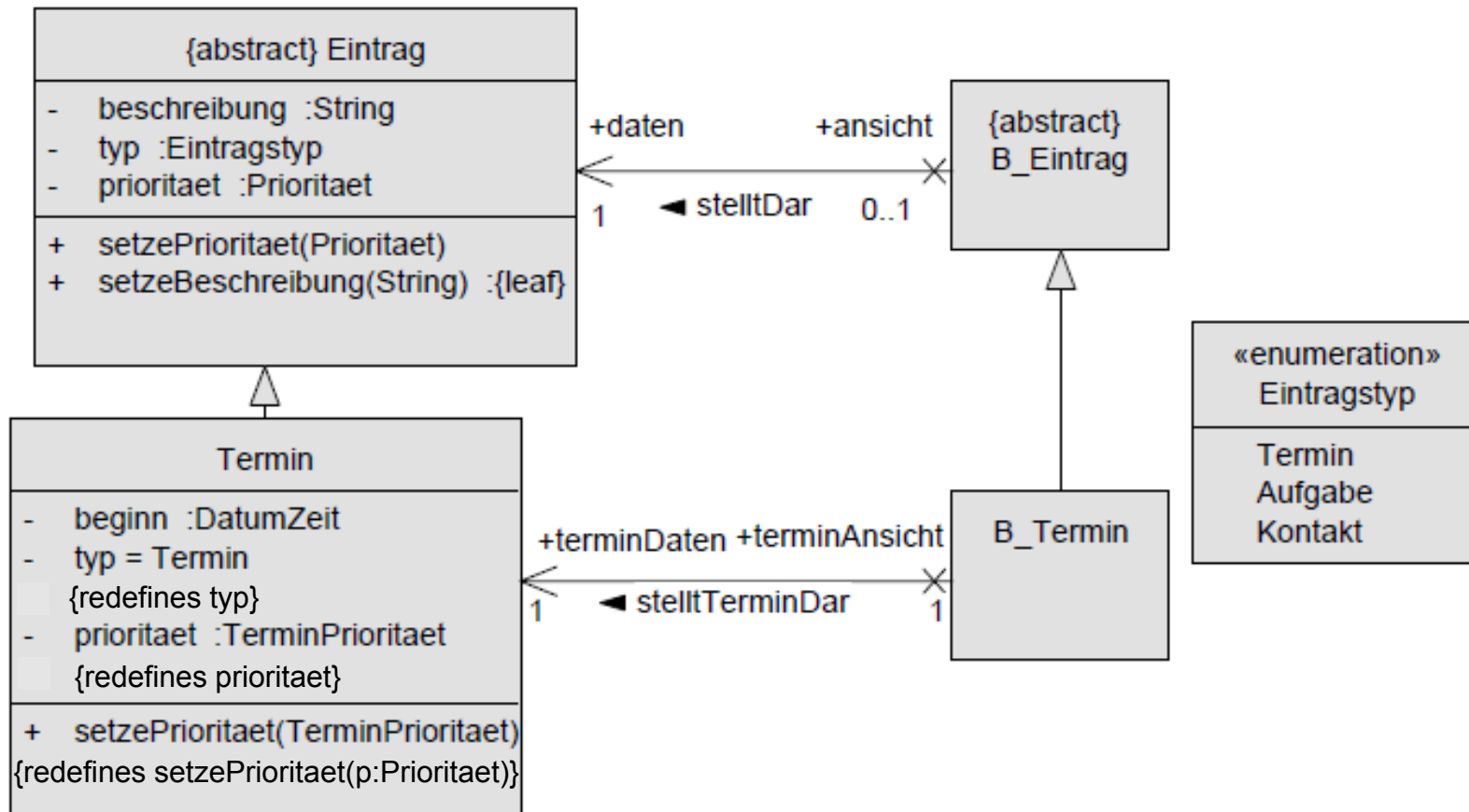
## Generalisierung: Redefinition von geerbten Merkmalen (1/2)

---

- Geerbte Merkmale können **in Subklasse redefiniert** werden
  - {redefines <feature>}
- **Redefinierbare Merkmale**
  - Attribute
  - Navigierbare Assoziationsenden
  - Operationen
- Redefinition von Operationen in (in)direkten Subklassen kann auch verhindert werden, indem die Operation mit der Eigenschaft {leaf} gekennzeichnet wird
- Das redefinierte Merkmal muss konsistent zum ursprünglichen Merkmal sein – verschiedenste **Konsistenzregeln** – z.B.
  - Ein redefiniertes Attribut ist konsistent zum ursprünglichen Attribut, wenn sein **Typ gleich oder ein Subtyp** des ursprünglichen Typs ist
  - Das **Intervall der Multiplizität** muss in jenem des ursprünglichen Attributs **enthalten** sein
  - Die Signatur einer Operation muss die **gleiche Anzahl an Parametern** aufweisen, etc.



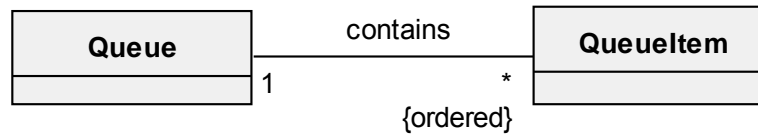
# Generalisierung: Redefinition von geerbten Merkmalen (2/2)



# Exkurs: Ordnung und Eindeutigkeit von Assoziationen

---

- **Ordnung** {ordered} ist **unabhängig von Attributen**

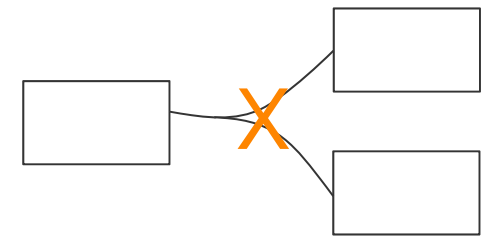
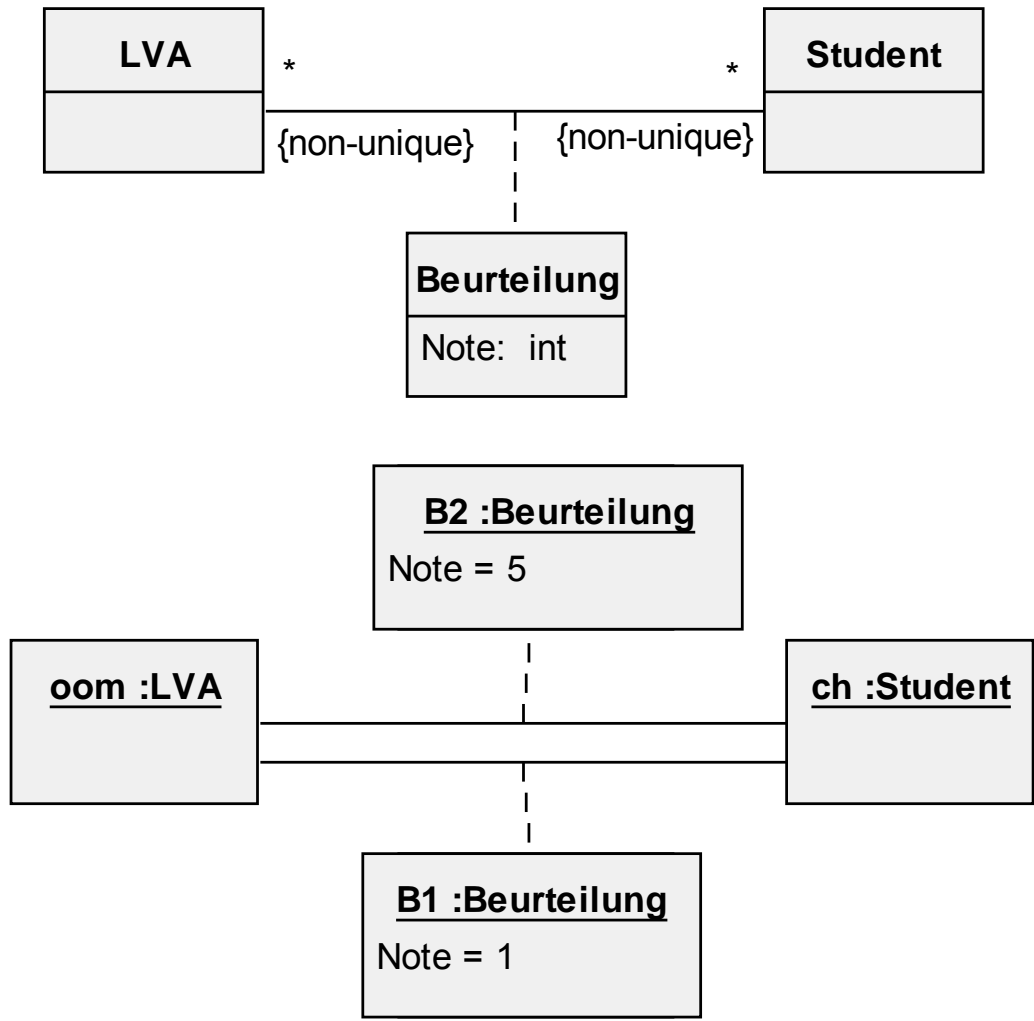


- **Eindeutigkeit**
  - Wie bei Attributen durch {unique} und {nonunique}
  - Kombination mit Ordnung {set}, {bag}, {sequence} bzw. {seq}

---

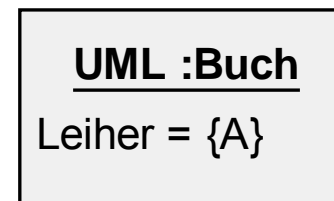
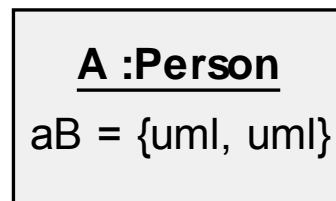
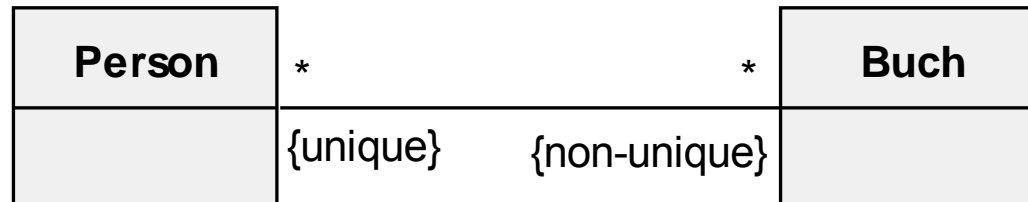
<b>Eindeutigkeit</b>	<b>Ordnung</b>	<b>Kombination</b>	<b>Beschreibung</b>
<i>unique</i>	<i>unordered</i>	set	Menge (Standardwert)
<i>unique</i>	<i>ordered</i>	<i>orderedSet</i>	Geordnete Menge
<i>nonunique</i>	<i>unordered</i>	<i>bag</i>	Multimenge, d.h. Menge mit Duplikaten
<i>nonunique</i>	<i>ordered</i>	<i>sequence</i>	Geordnete Menge mit Duplikaten (Liste)

# Unique / Non-Unique (1/2)



## Unique / Non-Unique (2/2)

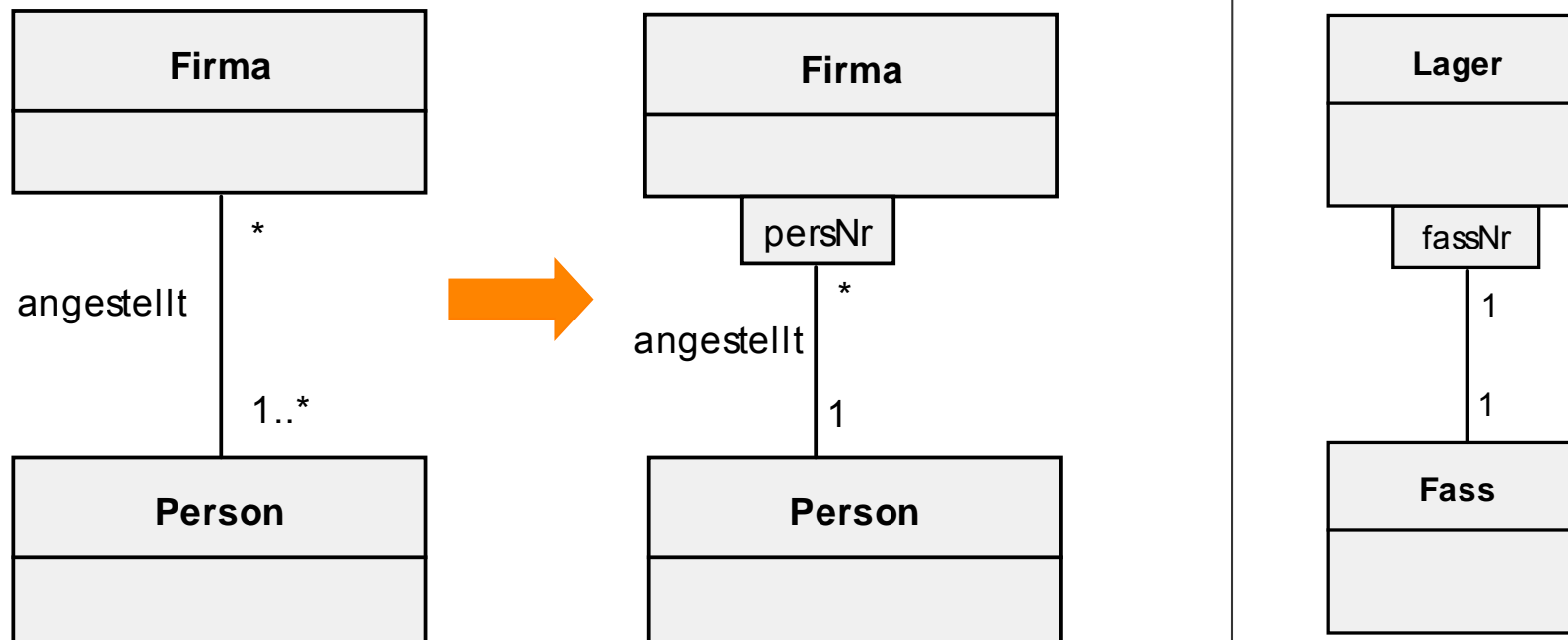
---



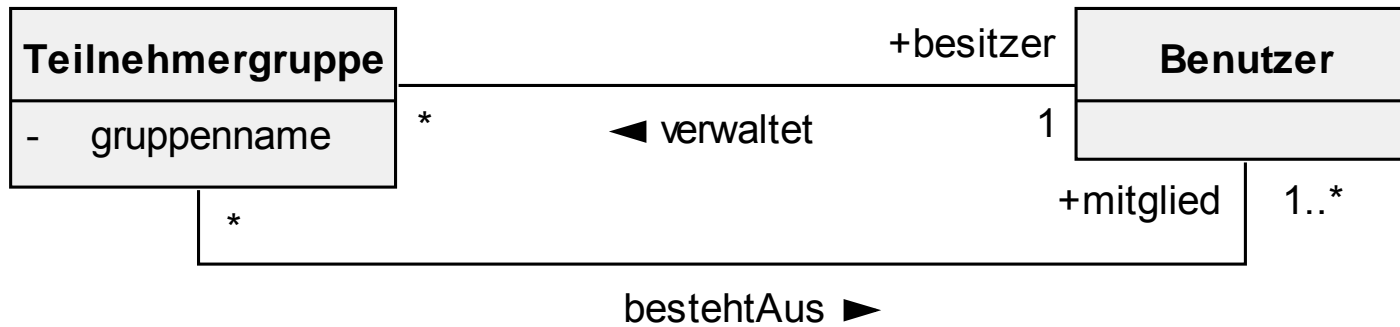


## Exkurs: Qualifizierte Assoziation (1/2)

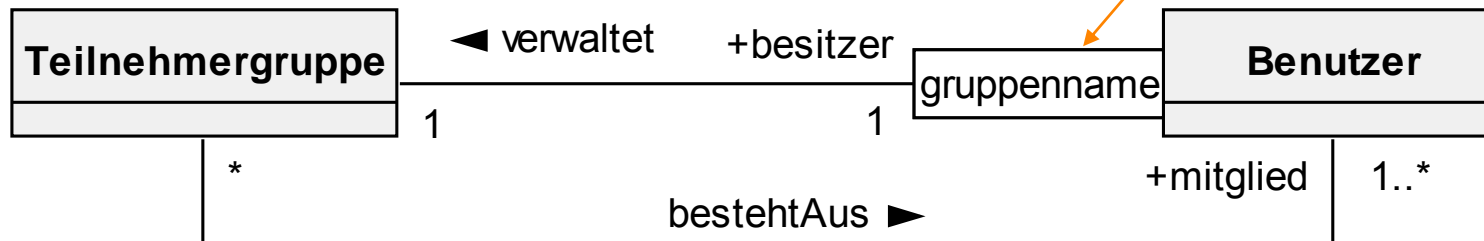
- Besteht aus einem **Attribut** oder einer **Liste von Attributen**, deren Werte die Objekte der assoziierten Klasse partitionieren
- **Reduziert** meist die **Multiplizität**
- Stellt eine **Eigenschaft der Assoziation** dar



# Exkurs: Qualifizierte Assoziation (2/2)



*muss je Benutzer eindeutig sein!*



# Aufgabenstellung

---

- Gesucht ist ein vereinfacht dargestelltes Modell der TU Wien entsprechend der folgenden Spezifikation.

Die TU besteht aus mehreren Fakultäten, die sich wiederum aus verschiedenen Instituten zusammensetzen. Jede Fakultät und jedes Institut besitzt eine Bezeichnung. Für jedes Institut ist eine Adresse bekannt. Jede Fakultät wird von ihrem Dekan, einem Mitarbeiter, geleitet.

Die Gesamtanzahl der Mitarbeiter ist bekannt. Mitarbeiter haben eine Sozialversicherungsnummer, einen Namen und eine E-Mail-Adresse. Es wird zwischen wissenschaftlichem und nicht-wissenschaftlichem Personal unterschieden.

Wissenschaftliche Mitarbeiter sind zumindest einem Institut zugeordnet.

Für jeden wissenschaftlichen Mitarbeiter ist seine Fachrichtung bekannt.

Weiters können wissenschaftliche Mitarbeiter für eine gewisse Anzahl an Stunden an Projekten beteiligt sein, von welchen ein Name und Anfangs- und Enddatum bekannt sind.

Manche wissenschaftliche Mitarbeiter führen Lehrveranstaltungen durch – diese werden als Vortragende bezeichnet. LVAs haben eine ID, einen Namen und eine Stundenanzahl.



## Identifikation von Klassen (1/2)

---

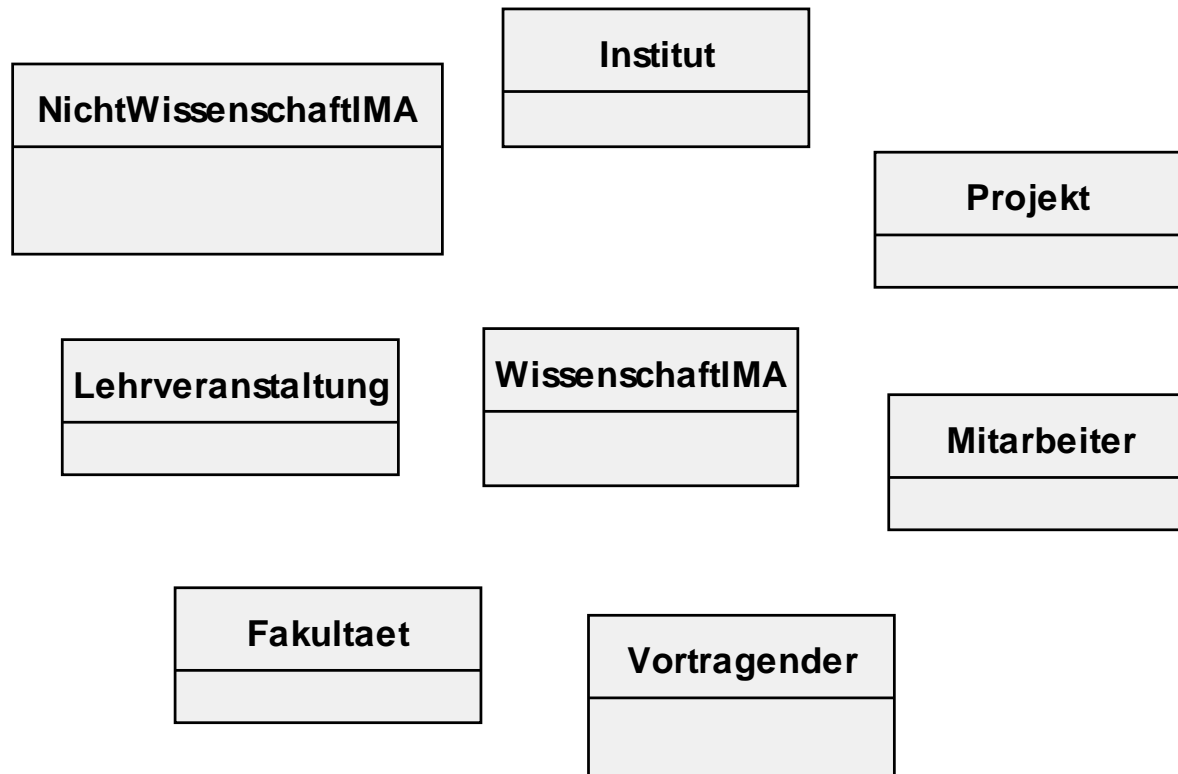
Die TU besteht aus mehreren **Fakultäten**, die sich wiederum aus verschiedenen **Instituten** zusammensetzen. Jede Fakultät und jedes Institut besitzt eine Bezeichnung. Für jedes Institut ist eine Adresse bekannt. Jede Fakultät wird von ihrem Dekan, einem Mitarbeiter, geleitet. Die Gesamtanzahl der **Mitarbeiter** ist bekannt. Mitarbeiter haben eine Sozialversicherungsnummer, einen Namen und eine E-Mail-Adresse. Es wird zwischen **wissenschaftlichem** und **nicht-wissenschaftlichem Personal** unterschieden.

Wissenschaftliche Mitarbeiter sind zumindest einem Institut zugeordnet. Für jeden wissenschaftlichen Mitarbeiter ist seine Fachrichtung bekannt. Weiters können wissenschaftliche Mitarbeiter für eine gewisse Anzahl an Stunden an **Projekten** beteiligt sein, von welchen ein Name und Anfangs- und Enddatum bekannt sind.

Manche wissenschaftliche Mitarbeiter führen **Lehrveranstaltungen** durch – diese werden als **Vortragende** bezeichnet. LVAs haben eine ID, einen Namen und eine Stundenanzahl.

## Identifikation von Klassen (2/2)

---



## Identifikation von Attributen (1/2)

---

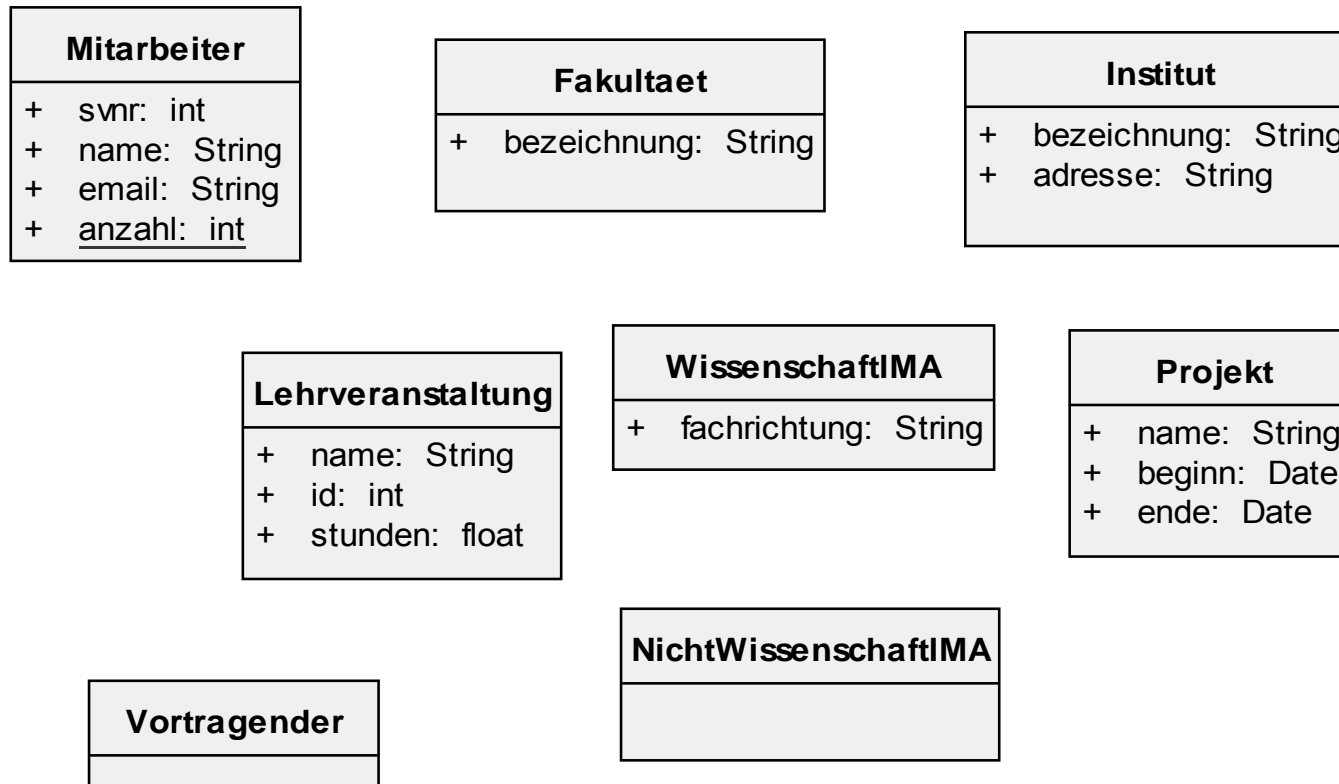
Die TU besteht aus mehreren Fakultäten, die sich wiederum aus verschiedenen Instituten zusammensetzen. Jede Fakultät und jedes Institut besitzt eine **Bezeichnung**. Für jedes Institut ist eine **Adresse** bekannt. Jede Fakultät wird von ihrem Dekan, einem Mitarbeiter, geleitet. Die **Gesamtanzahl** der Mitarbeiter ist bekannt. Mitarbeiter haben eine **Sozialversicherungsnummer**, einen **Namen** und eine **E-Mail-Adresse**. Es wird zwischen wissenschaftlichem und nicht-wissenschaftlichem Personal unterschieden.

Wissenschaftliche Mitarbeiter sind zumindest einem Institut zugeordnet. Für jeden wissenschaftlichen Mitarbeiter ist seine **Fachrichtung** bekannt. Weiters können wissenschaftliche Mitarbeiter für eine **gewisse Anzahl an Stunden** an Projekten beteiligt sein, von welchen ein **Name** und **Anfangs- und Enddatum** bekannt sind.

Manche wissenschaftliche Mitarbeiter führen Lehrveranstaltungen durch - diese werden als Vortragende bezeichnet. LVAs haben eine **ID**, einen **Namen** und eine **Stundenanzahl**.

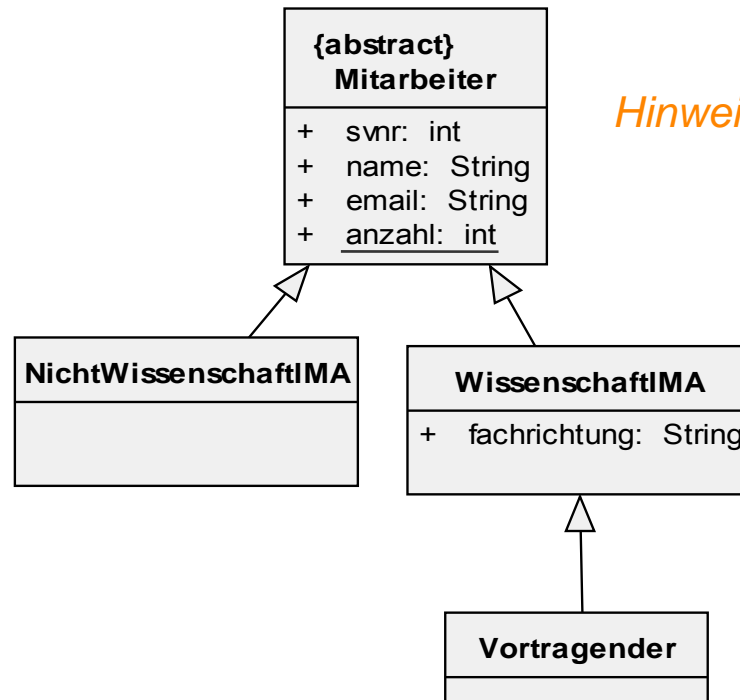
## Identifikation von Attributen (2/2)

---



# Generalisierung

- Es wird zwischen wissenschaftlichem und nicht-wissenschaftlichem Personal unterschieden.
- Manche wissenschaftliche Mitarbeiter führen Lehrveranstaltungen durch – diese werden als Vortragende bezeichnet.

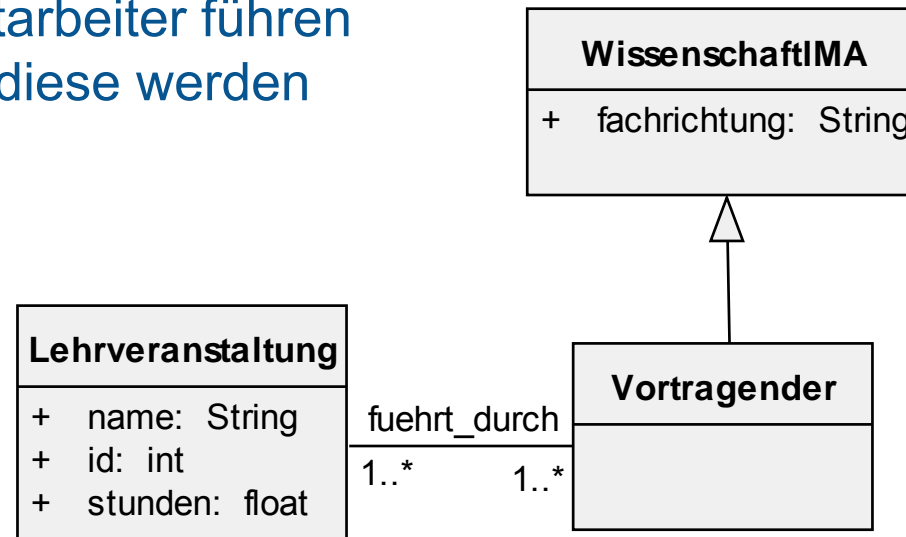


*Hinweis: Mitarbeiter ist nun eine abstrakte Klasse.*

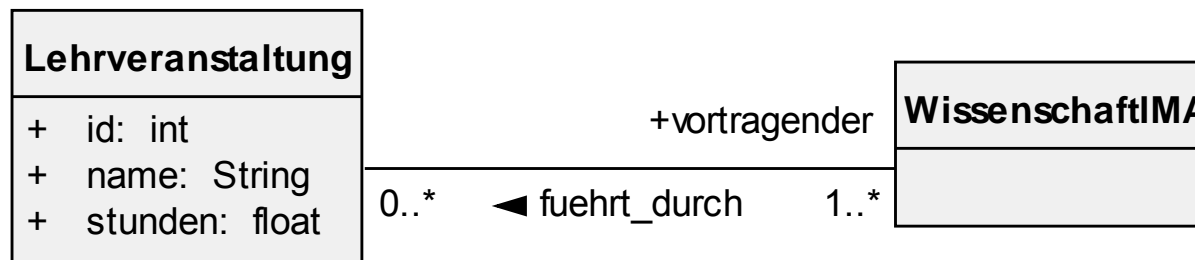


## Assoziation (1/2)

- Manche wissenschaftliche Mitarbeiter führen Lehrveranstaltungen durch – diese werden als Vortragende bezeichnet.



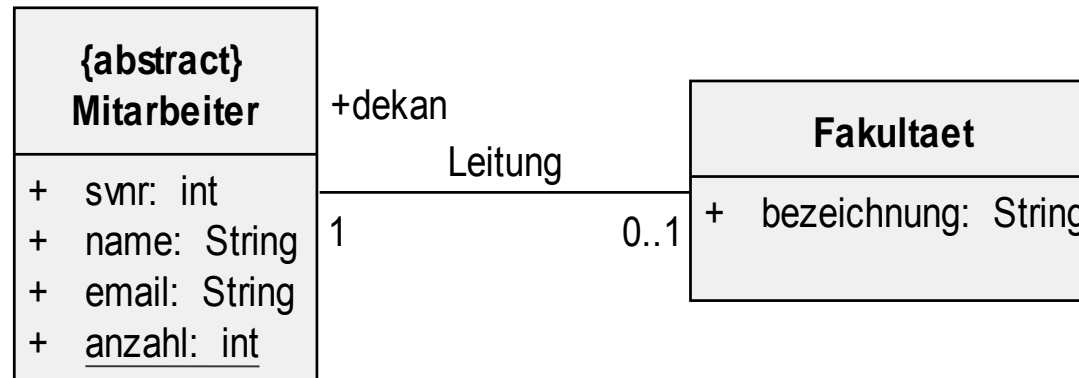
- Alternative Modellierung:



## Assoziation (2/2)

---

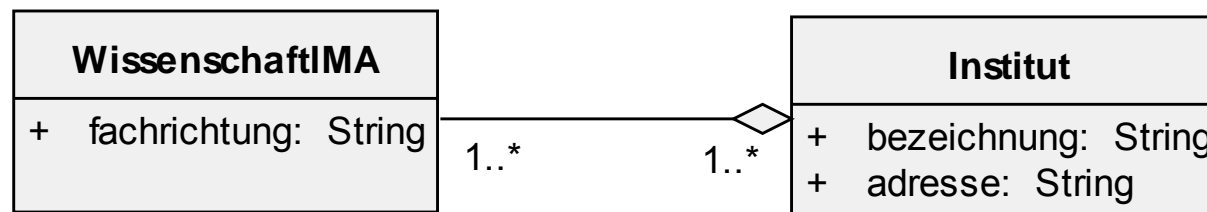
- Jede Fakultät wird von ihrem Dekan, einem Mitarbeiter, geleitet.



## Schwache Aggregation

---

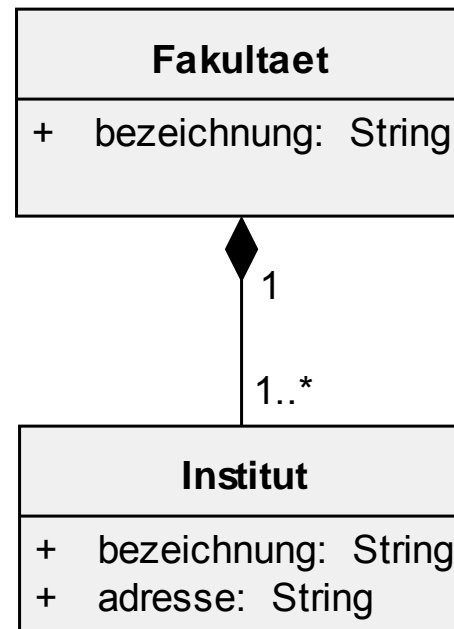
- Wissenschaftliche Mitarbeiter sind zumindest einem Institut zugeordnet.



# Starke Aggregation

---

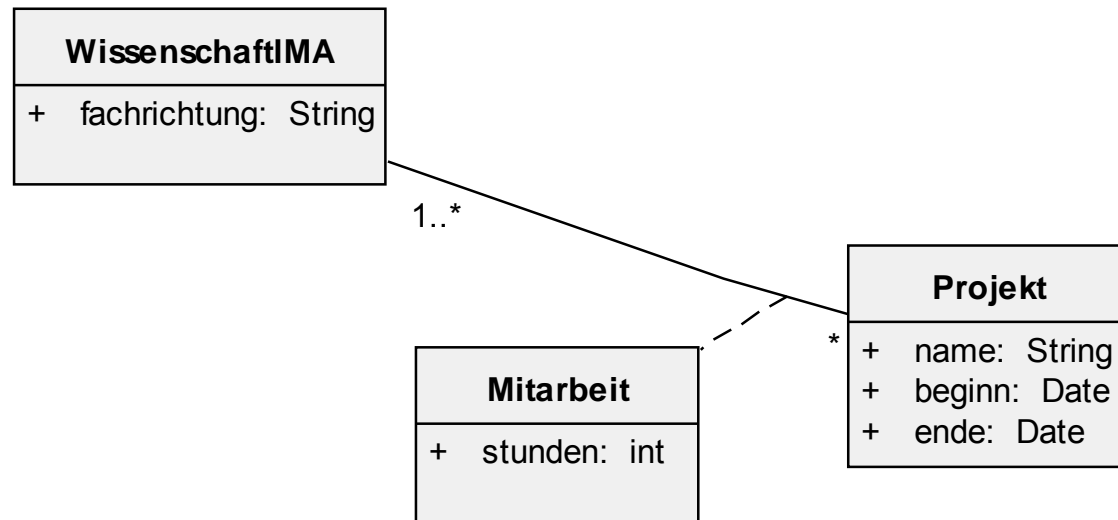
- Die TU besteht aus mehreren Fakultäten, die sich wiederum aus verschiedenen Instituten zusammensetzen.



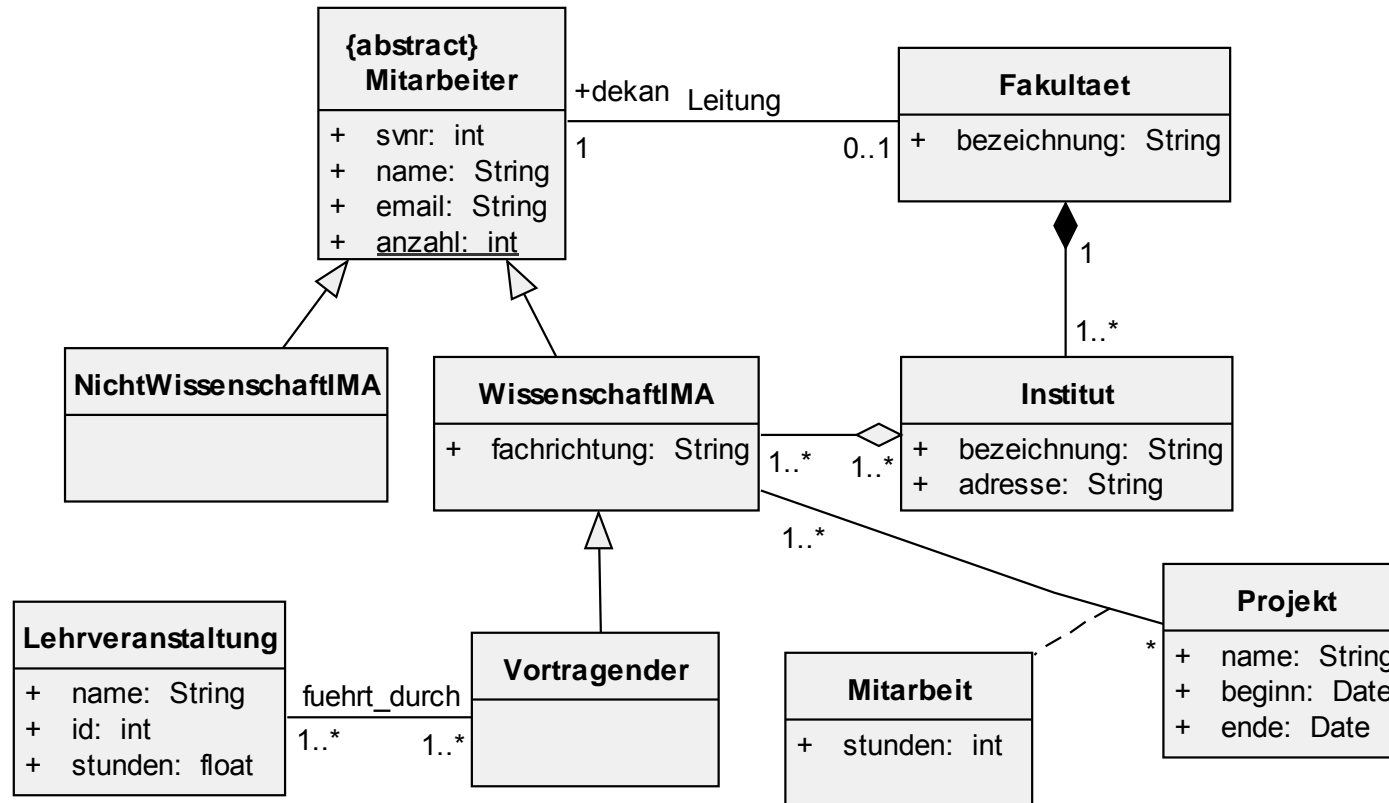
# Assoziationsklasse

---

- Weiters können wissenschaftliche Mitarbeiter an Projekten beteiligt sein.



## ... das gesamte Diagramm



# Übersetzung nach Java: Klassen (1/2)

---

Lehrveranstaltung
+ name: String
+ id: int
+ stunden: float



```
class Lehrveranstaltung {  
  
    public String name;  
    public int id;  
    public float stunden;  
  
    public Lehrveranstaltung(String name, int id,  
                             float stunden) {  
  
        this.name = name;  
        this.id = id;  
        this.stunden = stunden;  
    }  
}
```

- Erstellung einer konkreten Instanz oom:  
Lehrveranstaltung oom = new Lehrveranstaltung(  
 "Objektorientierte Modellierung", 394, 4);
- Zugriff auf Attribut name: oom.name;

# Übersetzung nach Java: Klassen (2/2)

Lehrveranstaltung
- name: String
- id: int
- stunden: float
+ getName() : String
+ getID() : int
+ getStunden() : float



```
class Lehrveranstaltung {  
  
    private String name;  
    private int id;  
    private float stunden;  
  
    public Lehrveranstaltung(String name, int id,  
                             float stunden) {  
        this.name = name;  
        this.id = id;  
        this.stunden = stunden;  
    }  
    public String getName() { return name; }  
    public int getId() { return id; }  
    public float getStunden() { return stunden; }  
}
```

- *Erstellung einer Instanz oom: wie vorher*
- *Zugriff auf Attribut name:* ~~oom.name~~; oom.getName();





# Übersetzung nach Java: Abstrakte Klasse

---

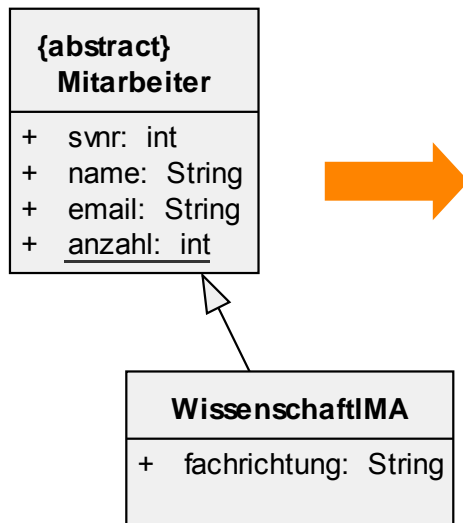
<b>{abstract}</b> <b>Mitarbeiter</b>
+ svnr: int + name: String + email: String + <u>anzahl: int</u>



```
abstract class Mitarbeiter {  
    public int svnr;  
    public String name;  
    public String email;  
    public static int anzahl = 0;  
  
    public Mitarbeiter(int svnr, String name,  
                        String email) {  
        this.svnr = svnr;  
        this.name = name;  
        this.email = email;  
    }  
}
```

- ~~Nicht möglich: ma1=new Mitarbeiter(123,"abc", "abc@xyz.at");~~

# Übersetzung nach Java: Generalisierung



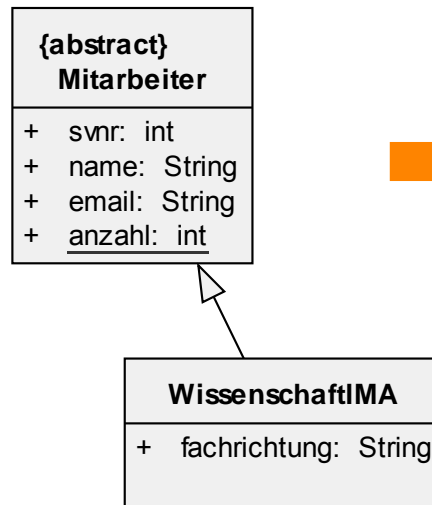
```
class WissenschaftlMA extends Mitarbeiter {
    public String fachrichtung;

    public WissenschaftlMA(int svnr, String name,
        String email, String fachrichtung) {
        super(svnr, name, email);
        this.fachrichtung = fachrichtung;
    }
}
```

- Neue Instanz: `wma1 = new WissenschaftlMA(123, "abc", "abc@xyz.at", "Informatik");`
- Zugriff auf Name: `wma1.name;`

# Übersetzung nach Java: Klassenvariable

- Angabe: Die Anzahl der Mitarbeiter ist bekannt
- Realisierung: Klassenvariable

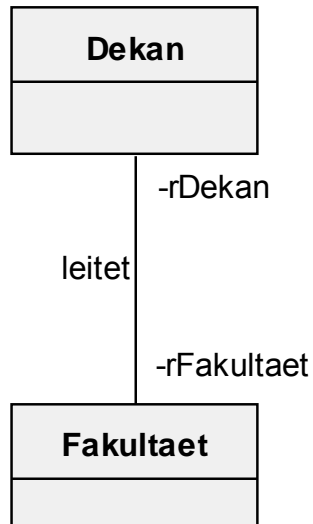


```
abstract class Mitarbeiter {
    ...
    public static int anzahl = 0;

    public Mitarbeiter (...) {
        ...
        anzahl++;
    }
}
class WissenschaftlMA extends Mitarbeiter {
    ...
    public WissenschaftlMA (...) {
        super(...);
    }
}
```

- Wird eine Klasse, die von Mitarbeiter erbt, instanziiert, wird `anzahl` um 1 erhöht
- Durch `Mitarbeiter.anzahl` oder `WissenschaftlMA.anzahl` oder `wma1.anzahl` bekommt man die Anzahl der Mitarbeiter

# Übersetzung nach Java: 1:1-Assoziation

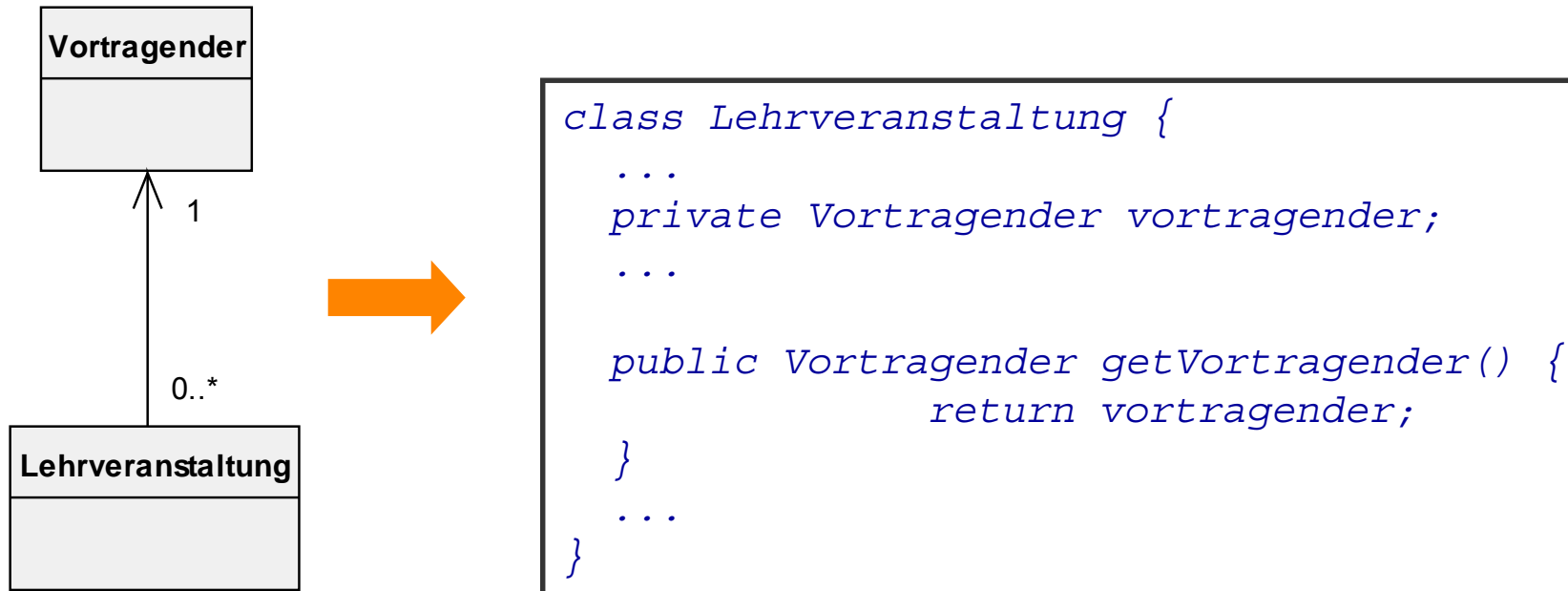


```
class Dekan {
    ...
    private Fakultaeet rFakultaet;
    ...
    public Fakultaeet getFakultaet() {
        return rFakultaet; }
}
class Fakultaeet {
    ...
    private Dekan rDekan;
    ...
    public Dekan getDekan() { return rDekan; }
}
```

- Die entsprechenden Rollen werden als Attribute in die jeweils gegenüberliegende Klasse eingefügt
- Ist die Multiplizität 0..1, kann das entsprechende Attribut unter Umständen auch den Wert `null` haben

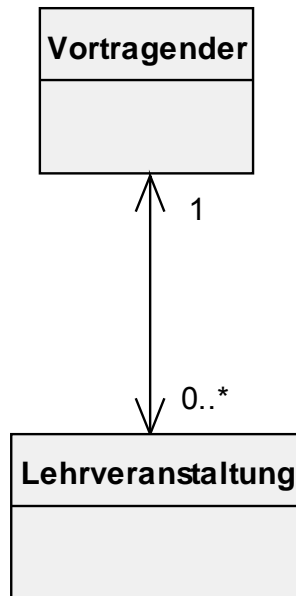
# Übersetzung nach Java: Unidirektionale ?:1-Assoziation

---



- Keine Änderung an Vortragender!
- Viel leichter zu implementieren als bidirektionale Assoziation

# Übersetzung nach Java: Bidirektionale 1:\*-Assoziation



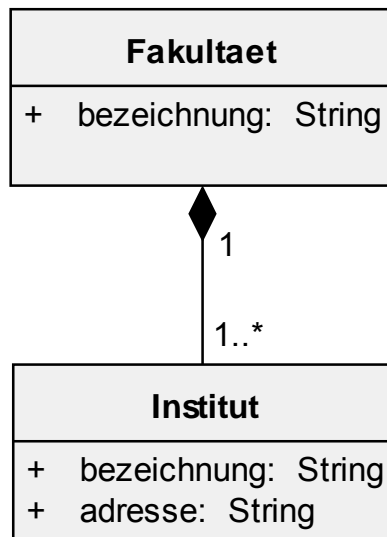
```
java.util.ArrayList;

class Vortragender {
    ...
    private ArrayList lvas;
    ...

    public Vortragender {
        lvas = new ArrayList();
    }
    ...
}
```

- Lehrveranstaltung wird wie vorher implementiert
- Wenn  $n$  fix vorgegeben ist (nicht  $*$ ), dann kann auch ein Array verwendet werden

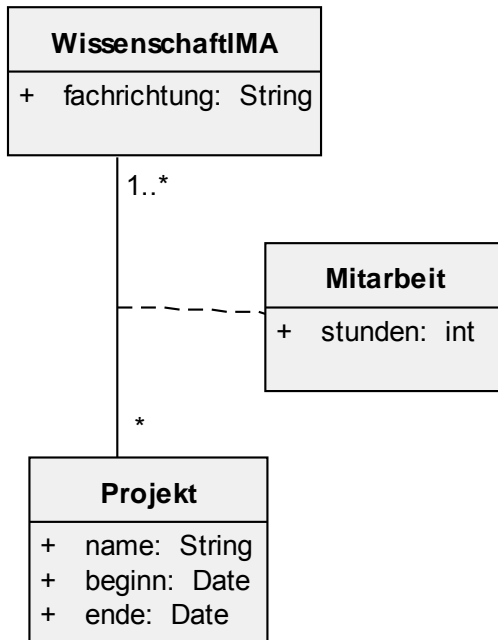
# Übersetzung nach Java: Starke Aggregation



```
class Fakultat {
    ...
    private Institut i1, i2, ..., in;
    ...
    public Fakultat () {
        i1 = new Institut();
        i2 = new Institut();
        ...
        in = new Insitut();
    }
    ...
}
```

- Nun müssen die Operationen, die auf den Instituten durchgeführt werden können, durch die Fakultät zur Verfügung gestellt werden

# Übersetzung nach Java: Assoziationsklasse



```
import java.util.Hashtable;

class WissenschaftlMA {
    ...
    private Hashtable rProjekt;
        //Schluessel: Projekt
        // Wert: Mitarbeit
    ...
}
```

- Die Assoziation wird mit Hilfe einer Hashtable abgebildet
- Ist die Assoziation nicht gerichtet, muss in der gegenüberliegenden Klasse ebenfalls eine Hashtable eingefügt werden



# Übersetzung nach Java:

## Zusammenfassung

---

- Klassen werden nach Java-Klassen übersetzt
- Attribute und Operationen werden in Java als Instanzvariablen und Methoden dargestellt
- Klassenvariable und –operationen (unterstrichen im Klassendiagramm) werden mit dem Schlüsselwort `static` versehen
- Assoziationen werden mit Hilfe von Variablen ausgedrückt
  - für 1:1-Beziehungen reicht jeweils eine Variable vom Typ der verbundenen Klasse
  - für 1:n-Beziehungen braucht man Arrays, ArrayLists oder Ähnliches
  - Angabe von Navigationsrichtung (unidirektional) vereinfacht i.A. die Implementierung
  - Operationen zur Verwaltung der Assoziationen müssen eingefügt werden
- Einfachvererbung wird von Java direkt unterstützt (`extends`)
- Assoziationsklassen können mit Hashtables dargestellt werden

# Datentypen in UML

---

- Datentypen können wie Klassen Attribute und Operationen haben
- Aber: Instanzen eines Datentyps haben keine Identität
  - Objekte: Instanzen einer Klasse
  - Werte: Instanzen eines Datentyps (z.B. Zahl 2)
- Notation: Klassensymbol mit Schlüsselwort «datatype»
- Beispiel:



- Arten von Datentypen:
  - Primitive Datentypen
  - Aufzählungstypen

# Arten von Datentypen:

## Primitive Datentypen

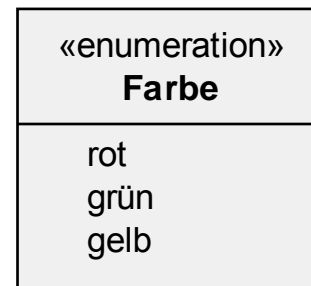
---

- Primitive Datentypen: Datentypen ohne innere Struktur
- Von UML vordefinierte primitive Datentypen:
  - Boolean
  - Integer
  - UnlimitedNatural
  - String
- Primitive Datentypen können auch selbst definiert werden:
  - wie beliebige Datentypen, nur mit dem Schlüsselwort «primitive»
- Primitive Datentypen können ebenfalls Operationen haben

# Arten von Datentypen: Aufzählungstypen

---

- Festlegung der Ausprägungsmenge per Aufzählung
- Notation: Klassensymbol mit Schlüsselwort «enumeration»
- Aufzählungstypen können Attribute und Operationen haben
- Mögliche Ausprägungen werden durch benutzerdefinierte Bezeichner (Literale) angegeben
  
- Beispiel:



# Inhalt

---

- **Klassendiagramm**
  - Klassen
  - Attribute und Operationen
  - Assoziationen
  - Schwache Aggregation
  - Starke Aggregation
  - Generalisierung
  - Zusammenfassendes Beispiel
  - Übersetzung nach Java
  - Datentypen in UML
- **Objektdiagramm**
- **Paketdiagramm**
- **Abhängigkeiten**

# Paketdiagramm

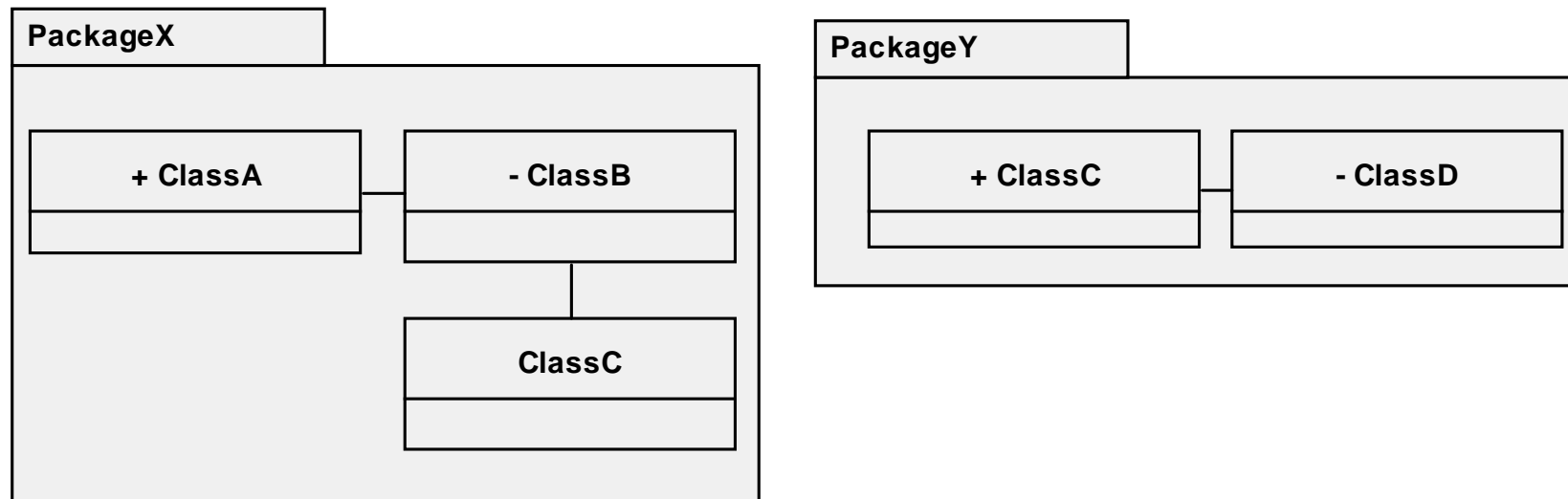
---

- UML-Abstraktionsmechanismus: Paket
- Modellelemente können höchstens **einem** Paket zugeordnet sein
- Partitionierungskriterien:
  - Funktionale Kohäsion
  - Informationskohäsion
  - Zugriffskontrolle
  - Verteilungsstruktur
  - ....
- Pakete bilden einen eigenen Namensraum
- Sichtbarkeit der Elemente kann definiert werden als »+« oder »-«

## Verwendung von Elementen anderer Pakete

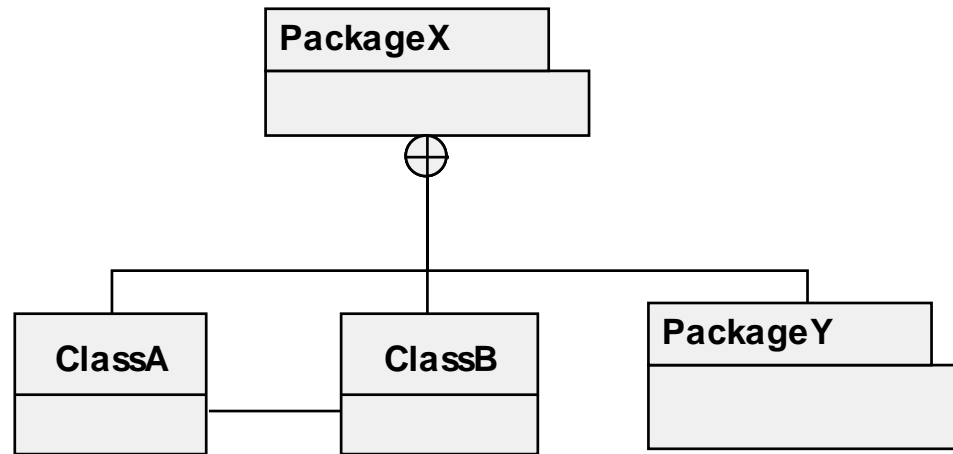
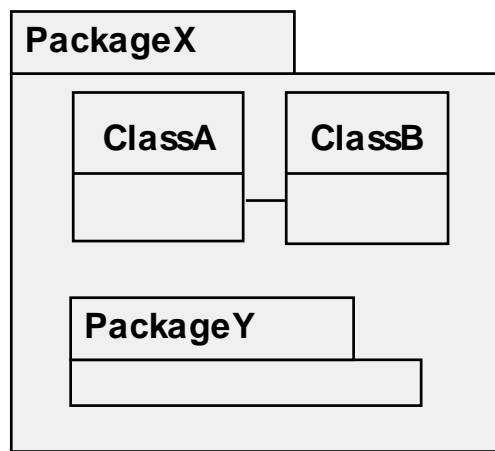
---

- Elemente eines Pakets benötigen Elemente eines anderen
- Qualifizierung dieser „externen“ Elemente
  - Zugriff über qualifizierten Namen
  - Nur auf öffentliche Elemente eines Pakets



# Hierarchien von Paketen

- Pakete können geschachtelt werden
  - Semantik wird durch die Implementierungssprache bestimmt
  - Beliebige Tiefe
  - Paket-Hierarchie bildet einen Baum
- Zwei Darstellungsformen





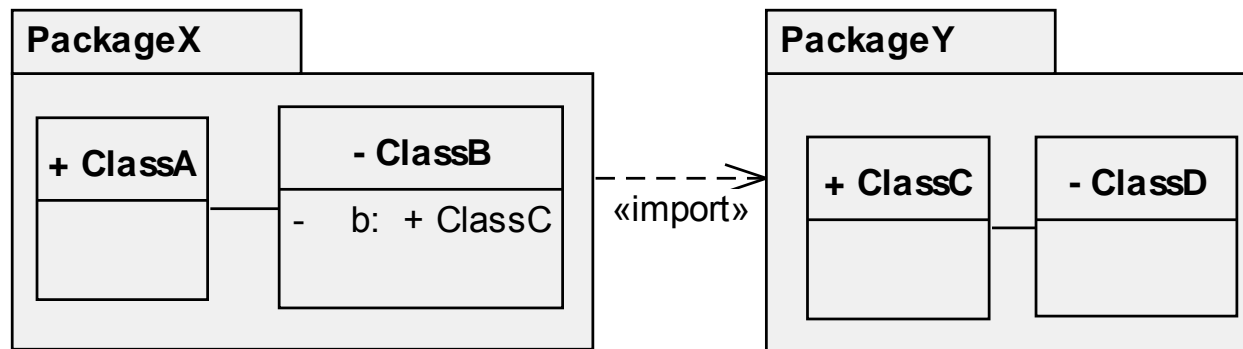
# Import von Elementen und Paketen

---

- **Import einzelner Elemente**
  - Voraussetzung: Sichtbarkeit des Elements ist öffentlich
- **Import ganzer Pakete**
  - Äquivalent mit Element-Import aller öffentlich sichtbaren Elemente des importierten Pakets
- **Sichtbarkeiten**
  - Beim Import kann die Sichtbarkeit der importierten Elemente und Pakete neu bestimmt werden
  - Sichtbarkeit nur öffentlich oder privat („+“ oder „-“)
  - «import»-Beziehungen für öffentliche Sichtbarkeit
  - «access»-Beziehungen für private Sichtbarkeit

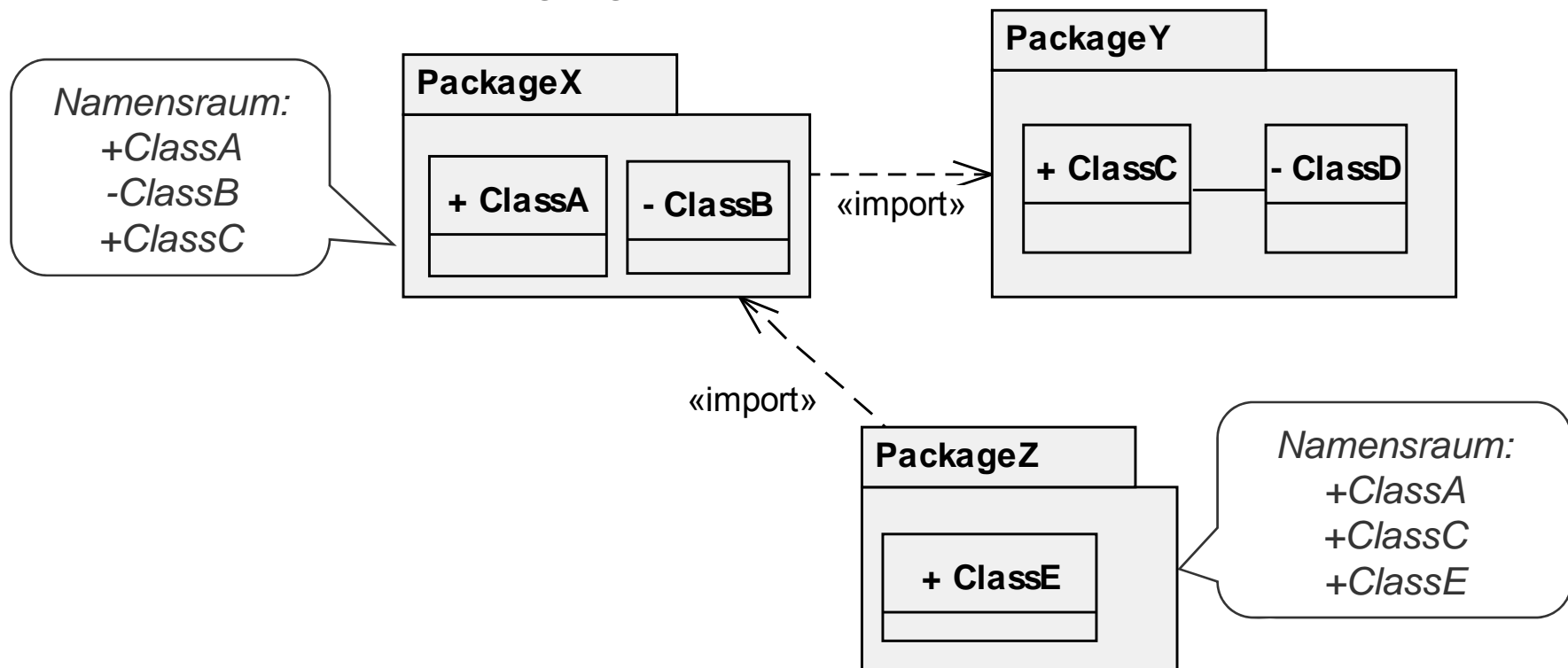
## Import von Elementen und Paketen – «import» (1/2)

- Veränderung des Namensraums
  - Lädt die Namen des importierten Pakets in den Namensraum des Klienten
  - Ändert damit den Namensraum des Klienten
  - Qualifizierte Namen sind nicht mehr nötig



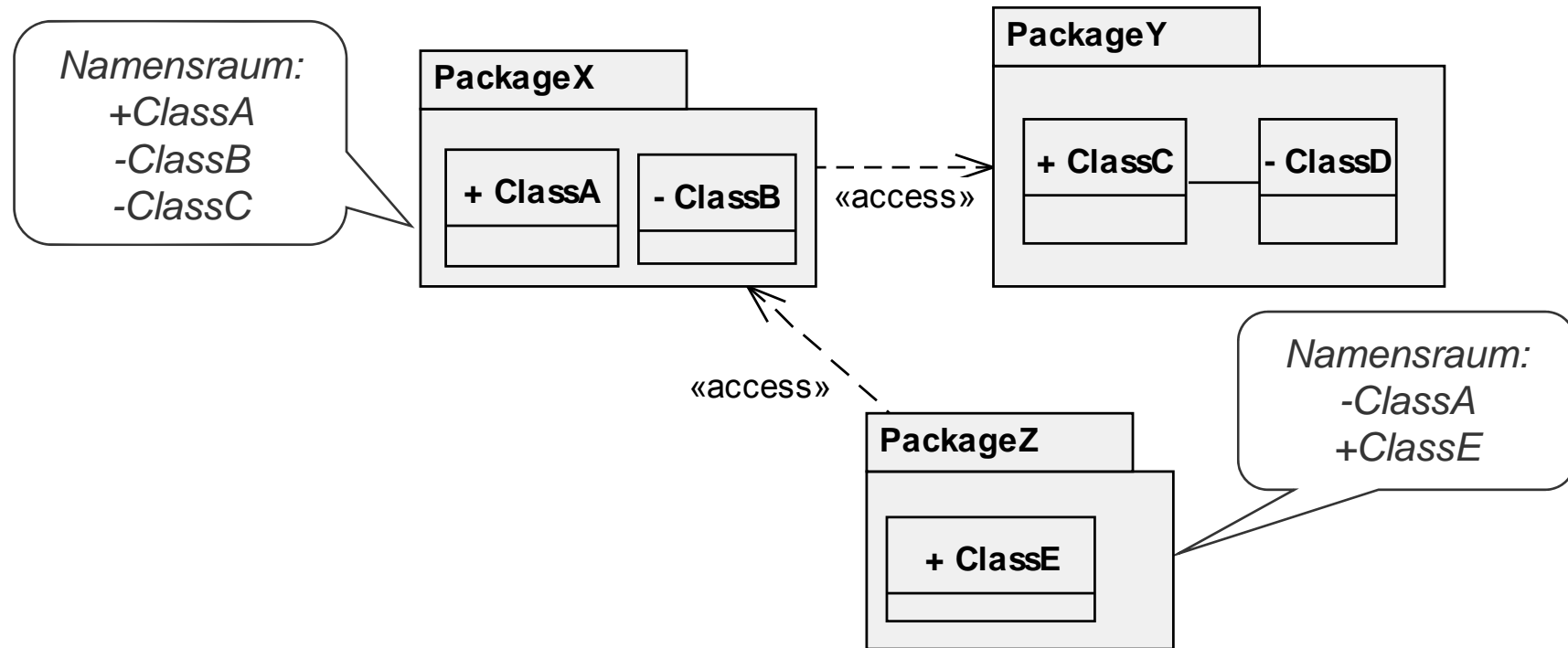
## Import von Elementen und Paketen – «import» (2/2)

- **Transitivität**
  - Die importierten Namen sind öffentlich und finden somit bei erneutem Import Berücksichtigung



# Import von Elementen und Paketen – «access»

- Nicht-transitiv
- Änderung der Sichtbarkeit der importierten Elemente auf privat



# Inhalt

---

- Klassendiagramm
  - Klassen
  - Attribute und Operationen
  - Assoziationen
  - Schwache Aggregation
  - Starke Aggregation
  - Generalisierung
  - Zusammenfassendes Beispiel
  - Übersetzung nach Java
  - Datentypen in UML
- Objektdiagramm
- Paketdiagramm
- **Abhängigkeiten**

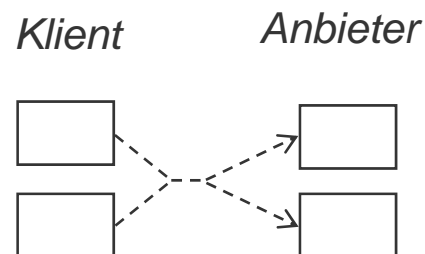
# Abhängigkeiten

---

- Eine Abhängigkeit stellt eine **allgemeine Kopplung zweier Modellelemente** (Klassen, Interfaces, Pakete usw.) dar
  - Änderungen in einem Element (Anbieter) können Änderungen im abhängigen Element (Klient) nach sich ziehen
- Abhängigkeit zwischen genau zwei Modellelementen (Normalfall)



- Abhängigkeit zwischen zwei Mengen von Modellelementen



# Arten von Abhängigkeiten

- Dependency:** Allgemeine Abhängigkeit ----->
  - macht keine Aussage über genaue Art der Abhängigkeit
- Deployment:** Verteilungsabhängigkeit

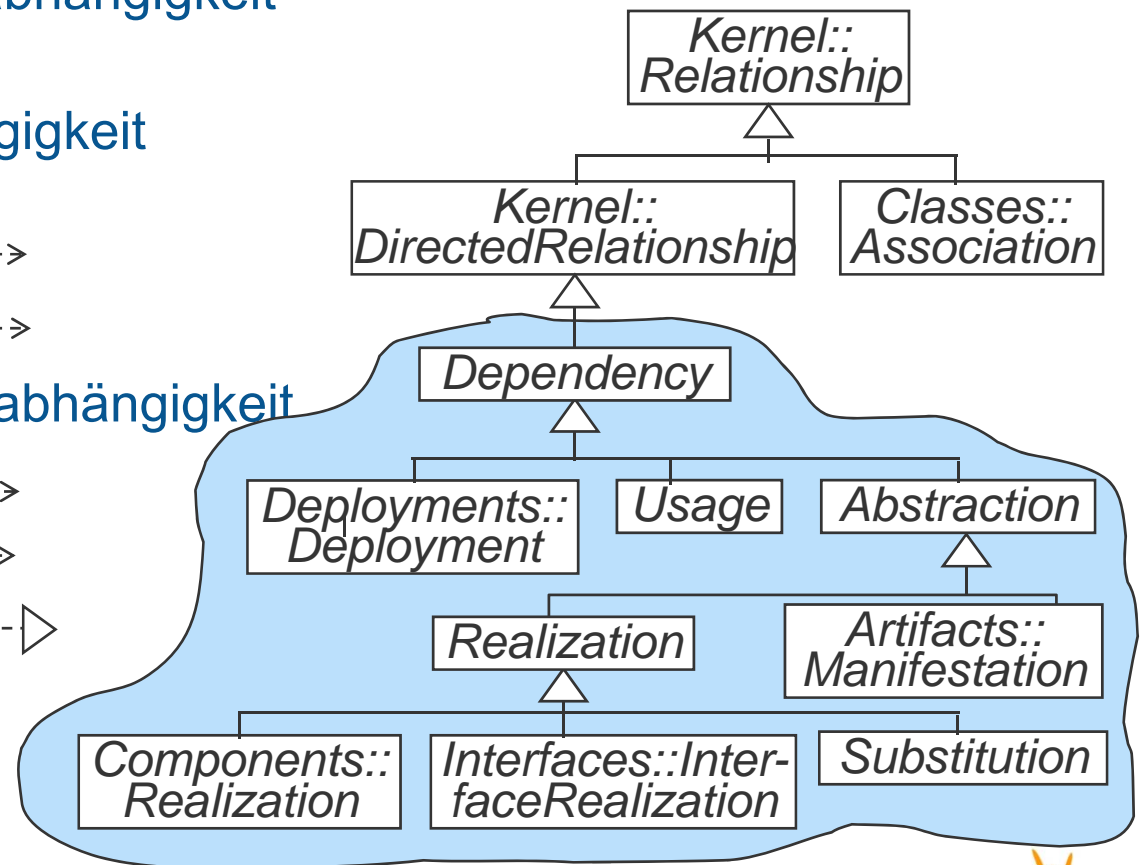
«deploy»  
----->

- Usage:** Benutzungsabhängigkeit

«use» ----->      «call» ----->  
 «instantiate» ----->      «send» ----->  
 «create» ----->

- Abstraction:** Abstraktionsabhängigkeit

«abstraction» ----->      «trace» ----->  
 «refine» ----->      «derive» ----->  
 Realisierungs-  
abhängigkeit -----> ▹  
 «substitute» ----->

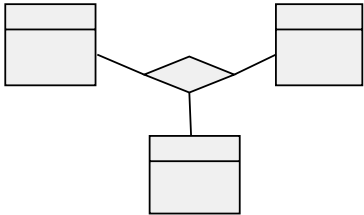
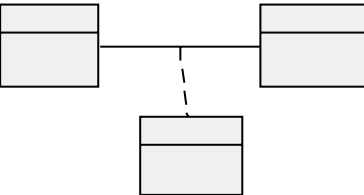
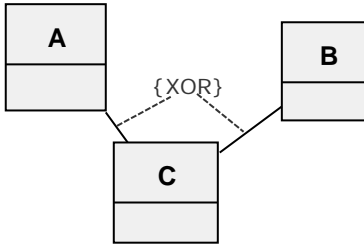


## Basiselemente (1/3)

Name	Syntax	Beschreibung
Klasse	<div style="border: 1px solid black; padding: 5px; width: fit-content;"> <p style="text-align: center;"><b>Klassenname</b></p> <p>- Attribut1: Typ - Attribut2: Typ</p> <p>+ Operation1() : void + Operation2() : void</p> </div>	Beschreibung der Struktur und des Verhaltens einer Menge von Objekten
abstrakte Klasse	<div style="display: flex; align-items: center; gap: 20px;"> <div style="border: 1px solid black; padding: 5px; width: 80px; height: 40px; display: flex; flex-direction: column; justify-content: center;"> <p style="font-weight: bold; margin: 0;"><i>Klassenname</i></p> </div> <p style="font-size: 1.2em;">oder</p> <div style="border: 1px solid black; padding: 5px; width: 80px; height: 40px; display: flex; flex-direction: column; justify-content: center;"> <p style="font-weight: bold; margin: 0;">{abstract} Klassenname</p> </div> </div>	Klasse, die nicht instanziiert werden kann
Assoziation	<div style="text-align: center; margin-bottom: 10px;"> <p>_____</p> <p>←—————→</p> <p>✕—————→</p> </div>	Beziehung zwischen Klassen: keine Angabe über Nav.-r.; mit Navigationsrichtung; in eine Richtung nicht navigierbar.






## Basiselemente (2/3)

Name	Syntax	Beschreibung
n-äre Assoziation		Beziehung zwischen n Klassen
Assoziations- klasse		nähere Beschreibung einer Assoziation
xor-Beziehung		Entweder steht Klasse A oder Klasse B in Beziehung zu C, nicht aber beide

## Basiselemente (3/3)

---

Name	Syntax	Beschreibung
schwache Aggregation		"Teil-Ganzes"-Beziehung
starke Aggregation = Komposition		exklusive "Teil-Ganzes"- Beziehung
Generalisierung		Vererbungsbeziehung zwischen Klassen

# Zusammenfassung

---

- Sie haben diese Lektion verstanden, wenn Sie wissen ...
- was der Unterschied zwischen einer Klasse und einem Objekt ist.
- was der Unterschied zwischen abstrakten und konkreten Klassen ist.
- was Attribute und Operationen einer Klasse sind und welche Eigenschaften diese haben können.
- dass Klassen durch Assoziationen miteinander verbunden werden.
- warum Assoziationen mit einer Multiplizität versehen werden.
- was Generalisierung ist und wann diese eingesetzt wird.
- was der Unterschied zwischen starker und schwacher Aggregation ist.
- wie Sie aus einer textuellen Angabe ein UML-Klassendiagramm erstellen.
- wie Sie die wichtigsten Elemente des Klassendiagramms in Java umsetzen können.
- wozu ein Meta-Modell benötigt wird.
- wie der Zusammenhang zwischen Klassen- und Objektdiagramm ist.
- was Interfaces sind.
- wozu Pakete eingesetzt werden.
- wie Abhängigkeiten in UML dargestellt werden können.



Anhang

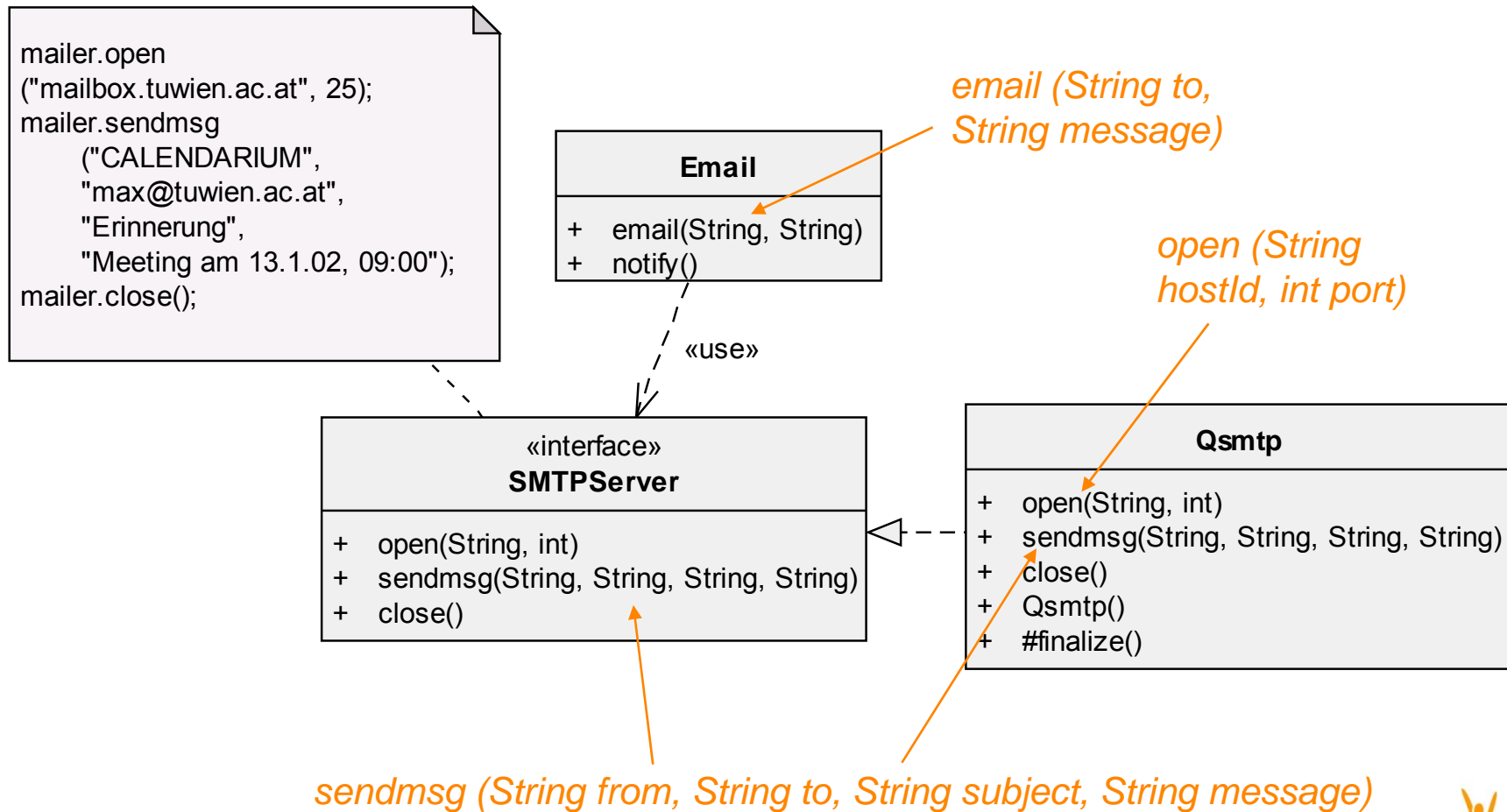
# Interface

---

- Ein Interface spezifiziert **gewünschtes Verhalten** durch **Zusammenfassung der Operationen**
  - einer Klasse
  - einer Komponente
  - eines Paketsund kann auch Attribute aufweisen
- **Unterschied zur abstrakten Klasse**
  - Abstrakte Klasse kann nur durch Subklassen realisiert werden, Interfaces durch beliebige Klassen
- Klassen, die ein **Interface realisieren - Anbieter** - können noch zusätzliche Operationen aufweisen
- Klassen, die ein **Interface benutzen - Klienten** - müssen nicht alle angebotenen Operationen tatsächlich nutzen

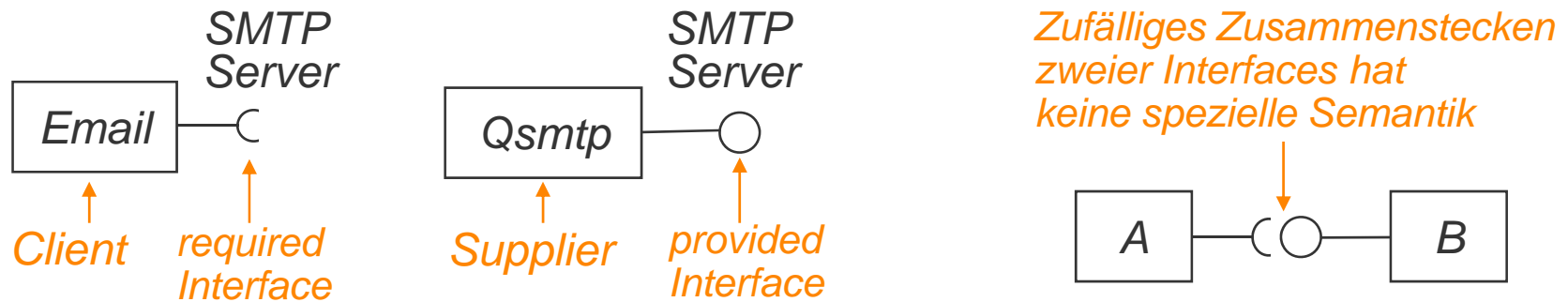
# Interface: Beispiel CALENDARIUM (1/2)

- Notationsvariante 1:

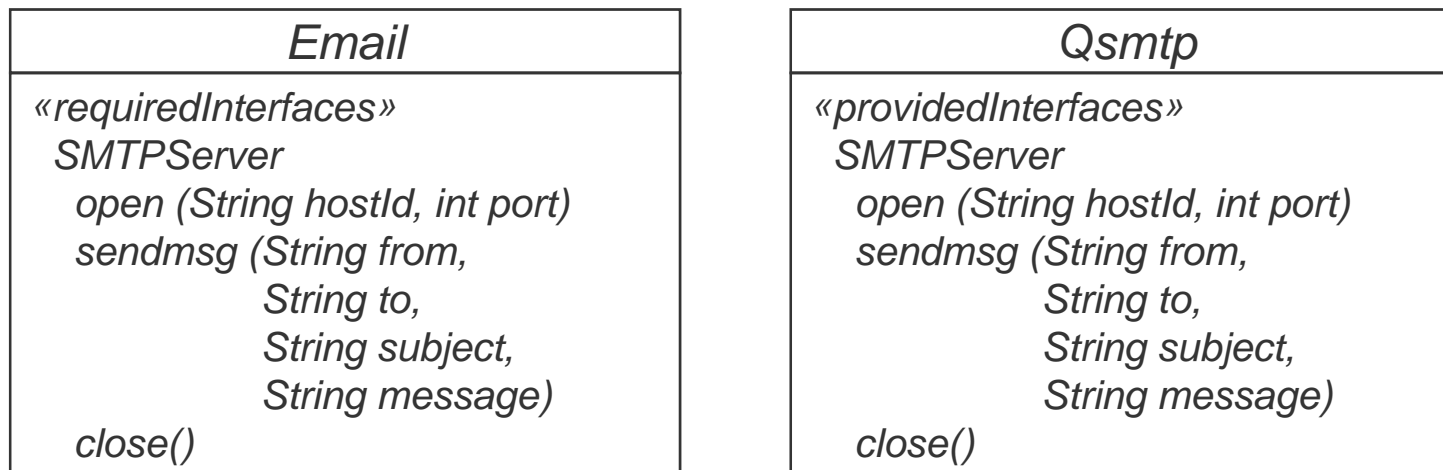


# Interface: Beispiel CALENDARIUM (2/2)

- Notationsvariante 2:



- Notationsvariante 3:



## Interface: Vorteile

---

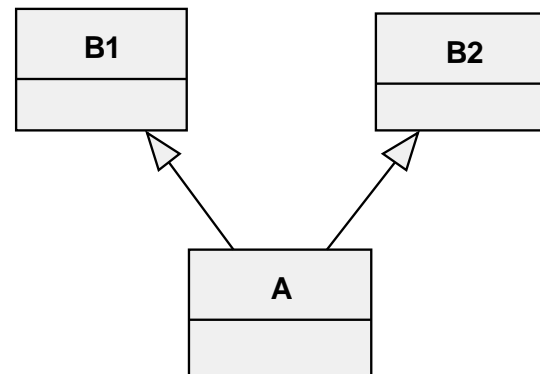
- Durch die Verwendung von Interfaces wird die **Vererbung von Implementierungen** von der **Vererbung von Interfaces** getrennt
  - Insbesondere **Frameworks** können größtenteils auf der Basis von Interfaces gebaut werden
- Eine **Klasse** kann als **Menge von Rollen** angesehen werden
  - Jedes **Interface** repräsentiert eine **Rolle**, die die Klasse spielt
  - Verschiedene **Klienten** verwenden nur jene Rollen, die für sie interessant sind
  - Mit Interfaces können **Sichten** auf eine Klasse für verschiedene Klienten realisiert werden
  - **Kopplung** wird reduziert, Flexibilität in Bezug auf Wartbarkeit und Erweiterbarkeit steigt



# Übersetzung nach Java: Generalisierung – Mehrfachvererbung (1/3)

---

- Bei Simulation von Mehrfachvererbung zu beachten:
  - Spezifikationsvererbung („Muss“):  
A erbt die nichtprivaten Schnittstellen von B1 und B2
  - Implementationsvererbung („Soll“):  
A erbt die Implementation von B1 und B2  
(Code-Wiederverwendung)

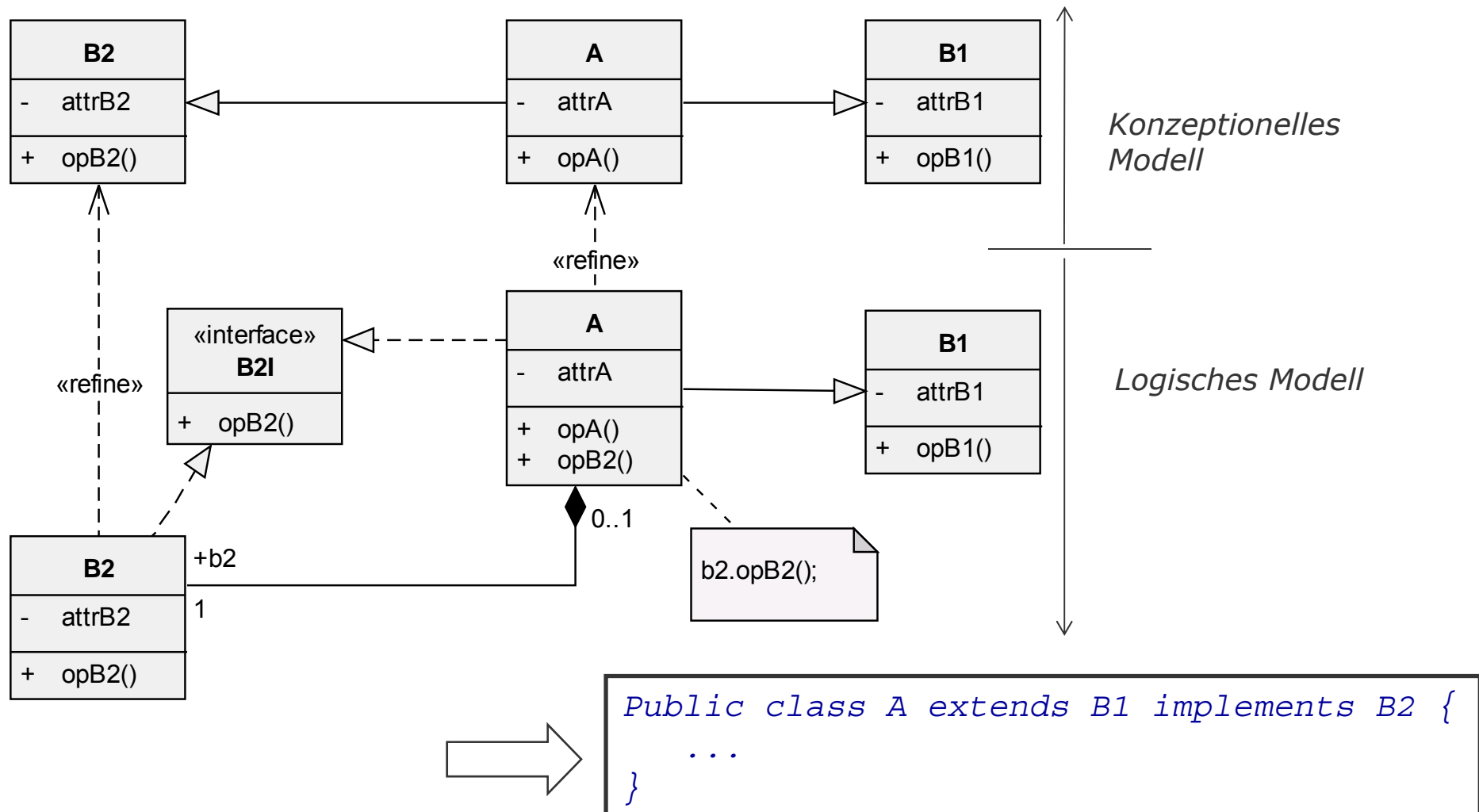


# Übersetzung nach Java: Generalisierung – Mehrfachvererbung (2/3)

---

- Vorgangsweise bei n-fach-Vererbung:
  - 1 Mal: „echte“ Vererbung von Basisklasse B1
  - (n-1) Mal: simulierte Vererbung durch
    - Interface-Realisierung (Spezifikationsvererbung); für die Basisklassen B2 ... Bn sind entsprechende Interfaces vorzusehen (im Beispiel B2)
    - Komposition („Implementierungsvererbung“) der „ehemaligen“ Basisklassen B2 ... Bn (im Beispiel B2I)

# Übersetzung nach Java: Generalisierung – Mehrfachvererbung (3/3)



# Metamodell

---

- Aufgabe: Entwicklung eines Tools zum Erstellen von Modellen (z.B. von Klassen- und Anwendungsfalldiagrammen)
- Problem: Wie stellen wir die Modelle generisch dar? (z.B. wie beschreiben wir, dass Klassen durch Assoziationen mit einander in Beziehung stehen können?)

Wir brauchen ein **Modell von den Modellen**  
**= METAMODELL**

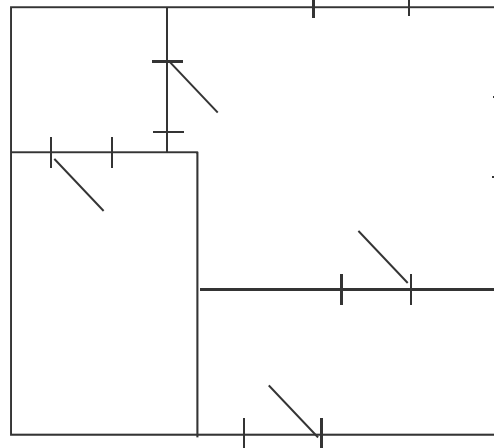
- auch UML besitzt ein Metamodell: die **Superstruktur**
  - Definition von Sprachkonzepten und deren Grammatiken für die Spezifikation von Modellen
  - UML kann mit einer Teilmenge von UML spezifiziert werden

# Beispiel: Metamodell der Architektur

*Reales Objekt*



*Modell (Plan)*



*Metamodell*

*Konstrukte (Objekttypen):*

————— *Wand*

—/— *Tür*

—+—+ *Fenster*

*Attribute:*

- *Ein Fenster hat Breite und Höhe*
- *Eine Wand hat eine Stärke*

*Regeln:*

- *Eine Tür grenzt links & rechts an eine Wand*
- *Fenster sind an Außenwänden*

# Erweiterung von UML

---

- Manchmal sind die von UML zur Verfügung gestellten Sprachkonstrukte unzureichend
  - ⇒ Erweiterung von UML ist notwendig

Varianten:

- 1.) Neues Metamodell
- 2.) Erweiterung des UML-Metamodells
  - a) Unkontrolliert
  - b) Mit vorgegebenen Erweiterungsmechanismen:

Stereotype	}	Profile
Tagged Value		
Constraint		

- Profile stellen den Hauptmechanismus dar, um UML zu erweitern
  - Technologische Erweiterungen (z.B. J2EE, .NET)
  - Domänenerweiterungen (z.B. Echtzeitanwendungen, EAI, Telekommunikation, Luftfahrt)

# Stereotype

---

- Jedes Modellelement kann mit einem Stereotyp weiter klassifiziert werden («...» in der Nähe des Namens)
- Anstelle einer Erweiterung der Metaklassenhierarchie:
  - Repräsentieren zur Modellierungszeit eingeführte Erweiterungen von Metamodellelementen
  - Dienen einem spezifischen Zweck (z.B. «hardware», «software»)
  - Stellen eine Brücke zwischen Modell und Metamodell dar (befinden sich im Modell, gehören zum Metamodell)
- Selbstdefinierte Symbole sind zulässig
- Beispiel:



- Vordefinierte Stereotype: «file», «executable», «send», «entity», ...

# Tagged Values

---

- Schlüssel/Wert-Paar (tagged value)
- Belegung der Attribute eines Stereotyps mit Werten
- Notation: {tag = value}
  - Vordefinierte Paare dienen i.A. zur Angabe von Metamodellattributen, z.B.
    - {persistence=transitory/persistent}
    - {isAbstract=true/false}
  - Statt {isX=true} kann einfach {X} geschrieben werden
  - Benutzerdefinierte Paare werden innerhalb von UML nicht interpretiert, z.B.
    - {author="Martin"}