# UML – Strukturdiagramme

by

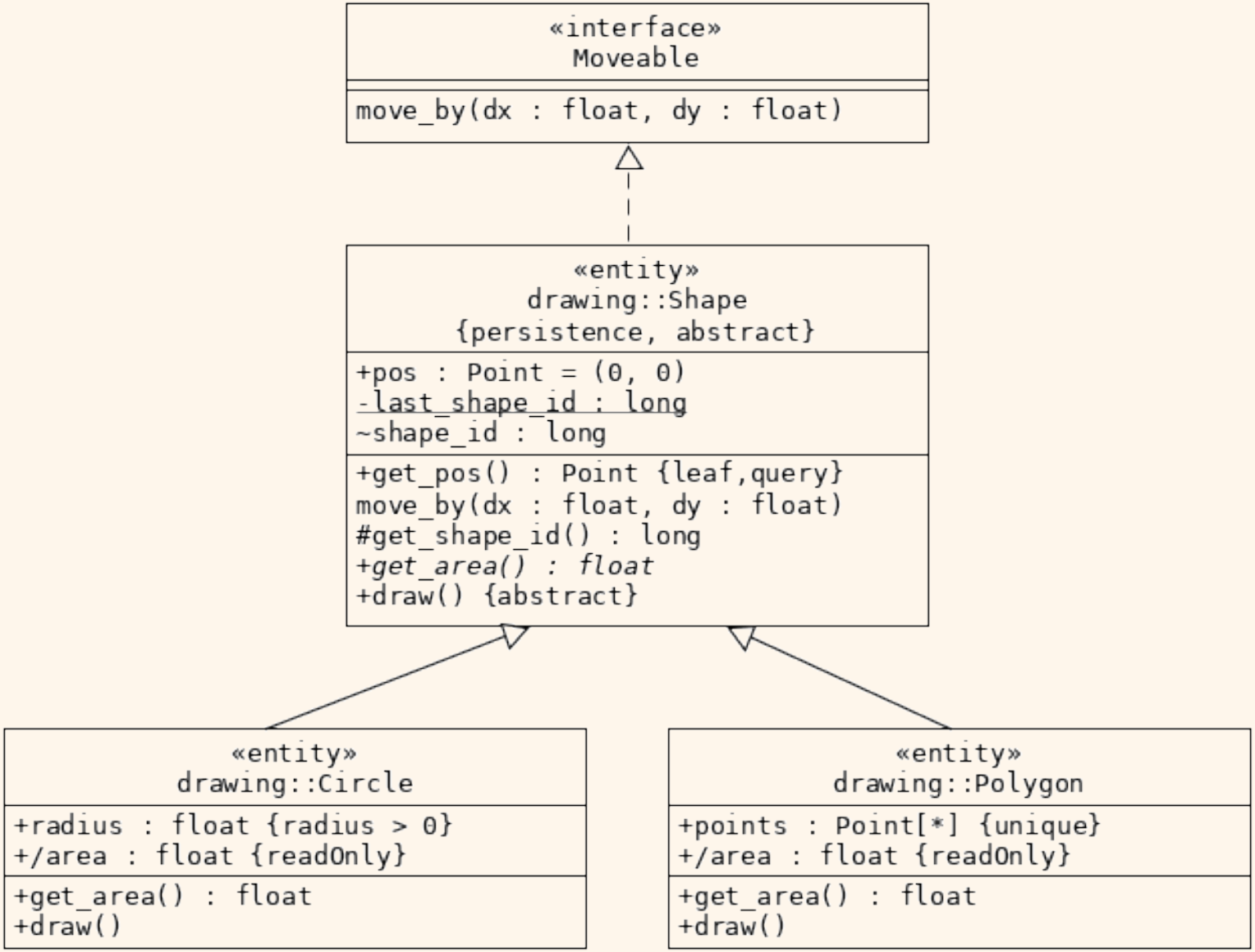
Dr. Günter Kolousek

# Klassendiagramm

- stellt Klassen (samt Aufbau) und Zusammenhänge dar
- werden verwendet um
  - konzepitionelle Sicht (WAS)
  - Realisierung (WIE)

  darzustellen
- Klasse $\equiv$ Rechteck
  1. Stereotyp, Klassenname, Eigenschaften (obligatorisch)
  2. Attribute (optional)
  3. Methoden (optional)
  4. meist für Invarianten (optional)

# Klasse


«interface»
Moveable
move_by(dx : float, dy : float)
«entity»
drawing::Shape
{persistence, abstract}
+pos : Point = (0, 0)
-last_shape_id : long
~shape_id : long
+get_pos() : Point {leaf,query}
move_by(dx : float, dy : float)
#get_shape_id() : long
+get_area() : float
+draw() {abstract}
«entity»
drawing::Circle
+radius : float {radius > 0}
+/area : float {readOnly}
+get_area() : float
+draw()
«entity»
drawing::Polygon
+points : Point[*] {unique}
+/area : float {readOnly}
+get_area() : float
+draw()

# Stereotypen & Eigenschaften

- vordefinierte Schlüsselwerte & Stereotypen für Klassen
  - `interface`
  - `type` ... Typ, d.h. keine Instanzen
  - `implementationClass` ... implementiert einen Typ
  - `metaClass` ... Metaklasse
  - `entity`: persistentes Objekt, das ein Businesskonzept darstellt
  - `focus`: Objekt, das Teil der ”core logic” ist
  - `auxiliary:` Objekt, das ein `focus` Objekt unterstützt
  - `control`: Steuerungsklassen: Ablauf-, Steuerungs-, Berechnungsvorgänge
  - `utility`: keine Instanzen, nur statische Methoden und Attribute
  - `enumeration`
- vordefinierte Eigenschaften:
  - `abstract` ($\rightarrow$ kursiv!): Klasse und Operation
  - `query`: Operation ohne Nebeneffekt
  - `leaf`: nicht überschreibbare Operation

# Property

- *property*: *attribute* oder *association end*
- property modifier
  - `readOnly`
  - `id` ... Teil des Identifiers
  - `ordered`
    - vs. `unordered`
  - `nonunique` ... doppelte Elemente erlaubt
    - in mehrwertigen Properties
    - vs. `unique`
  - `subsets <property-name>`
  - `redefines <property-name>` ... überschreibt vererbtes Attribut
  - `seq` ... `nonunique` $\wedge$ `ordered`
  - oder ”property constraint”

# `unique` vs. `ordered`

| Eindeutigkeit | Ordnung | Kombination | Beschreibung |
|---|---|---|---|
| `unique` | `unordered` | `set` | Menge (default) |
| `nonunique` | `unordered` | `bag` | Multimenge |
| `unique` | `ordered` | `ordered` | Geordnete Menge |
| `nonunique` | `ordered` | `sequence`, `seq` | geordnet mit Duplikaten |

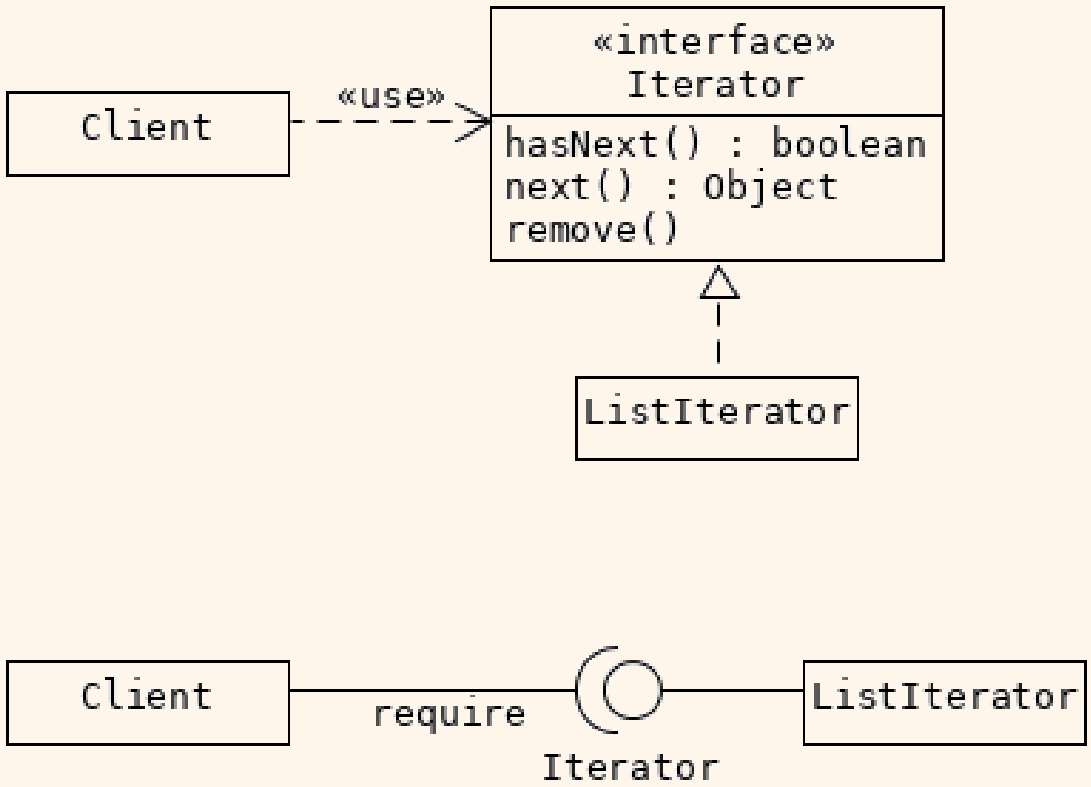
# Multiplizitäten

| | |
|---|---|
| `1` | exakte Angabe, d.h. genau 1 |
| `1,2` | Aufzählung, d.h. 1 oder 2 |
| `0..1` | Bereich, d.h. `0,1` bzw. optional |
| `*` | beliebig, d.h. `0..*` bzw. von 0 bis $\infty$ |

keine Angabe dann

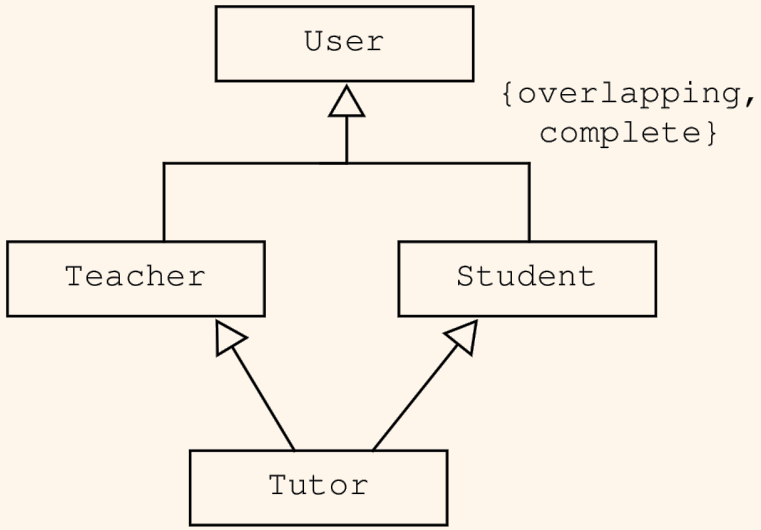
- bei Attributen … 1
- bei Assoziationsenden abhängig von Tätigkeit
  - Analyse: wird meist als unbestimmt betrachtet
  - Design: 1

# Interface

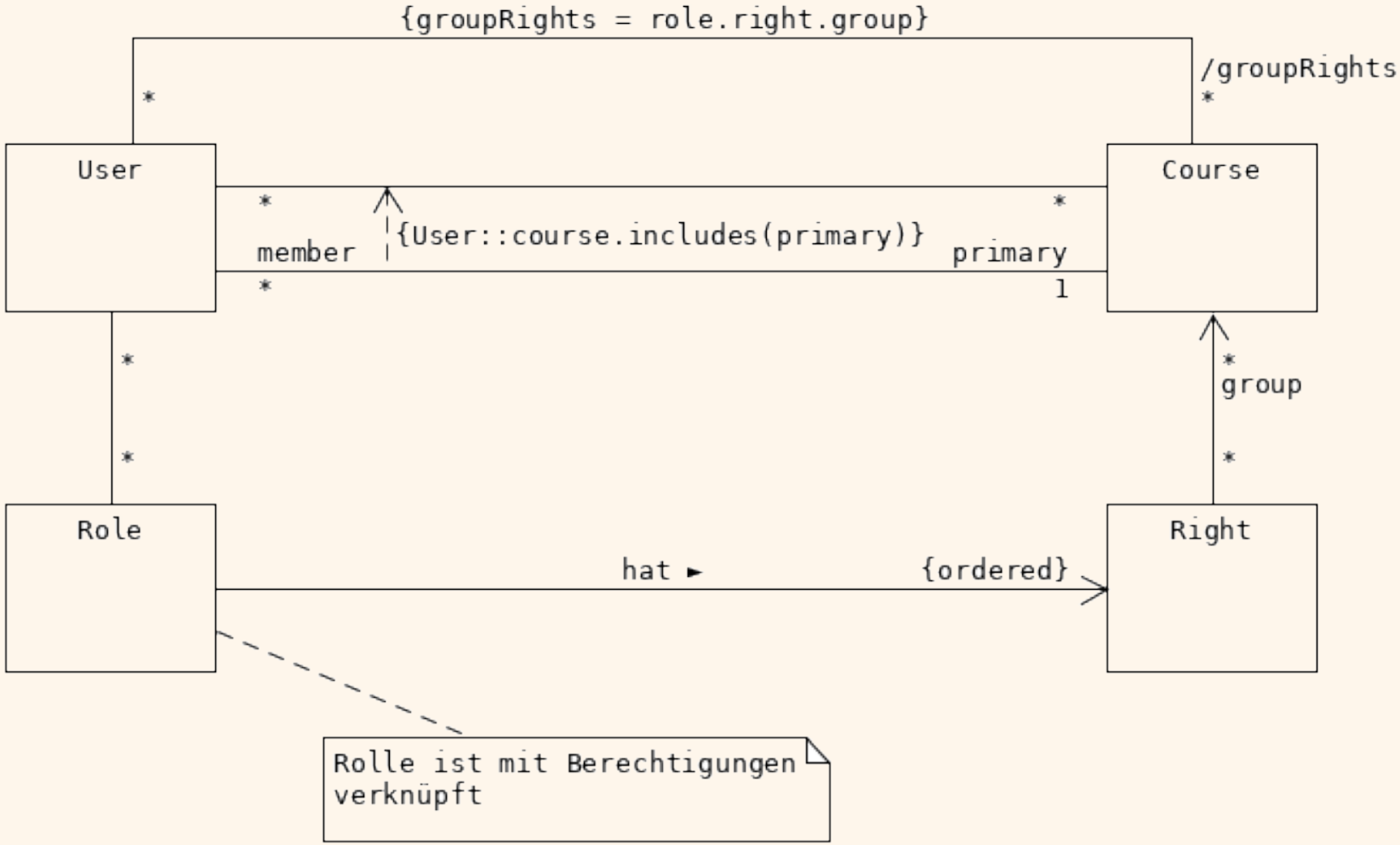

«interface»
Iterator
hasNext() : boolean
next() : Object
remove()
Client
«use»
ListIterator
Client
require
Iterator
ListIterator

# Vererbung

- `complete` vs. `incomplete`
  - `complete`: alle Unterklassen sind schon definiert
- `disjoint` vs. `overlapping`
  - `disjoint`: Klassen haben keine gemeinsame Instanzen
- default: `{incomplete,overlapping}`

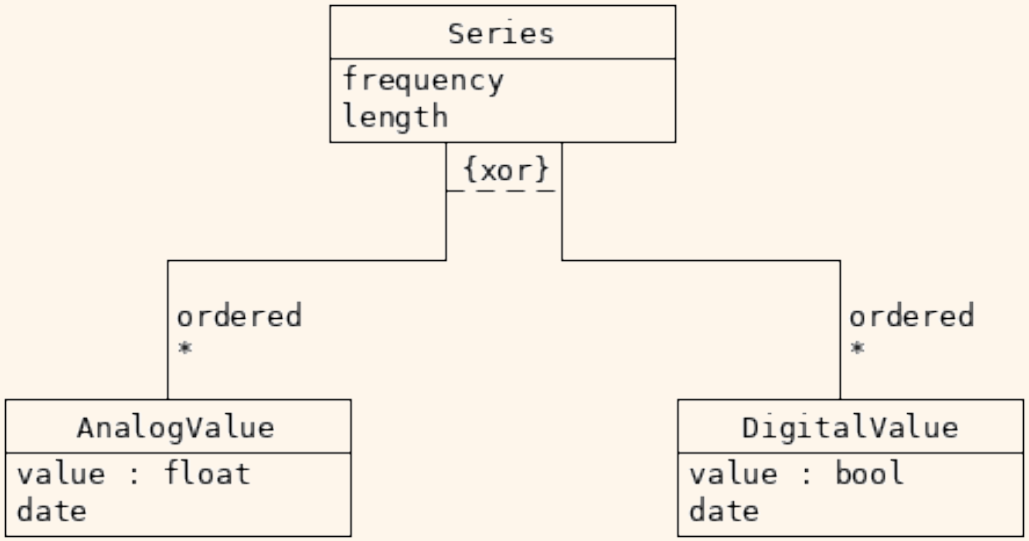


# Assoziationen und Rollen



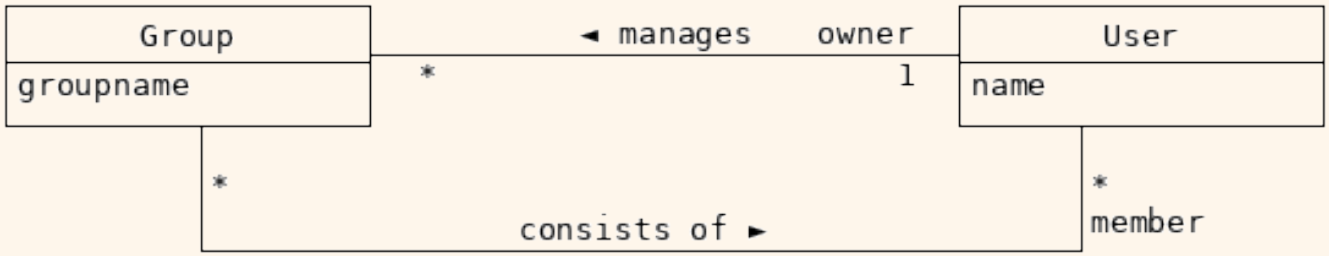
# Einschränkungen bei Assoziationen

- Benutzerdefinierte Einschränkungen
  - z.B. `User::course.includes(primary)`
    - aber auch: $\forall u \in \text{User} : \text{primary} \in u.\text{course}$
- XOR-Einschränkung



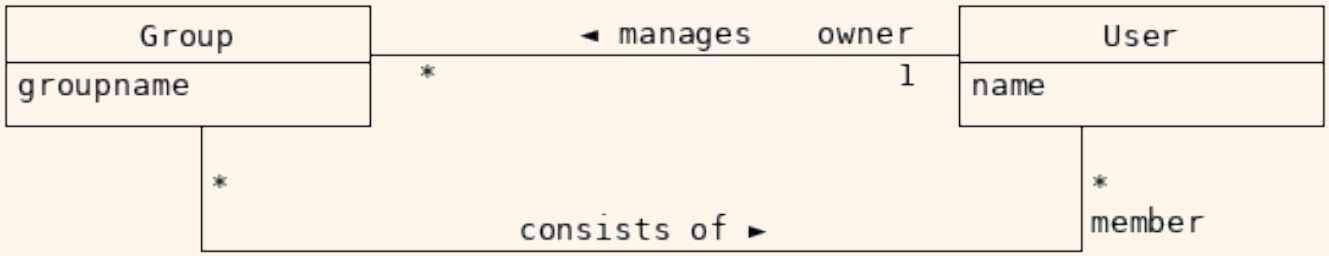
# Qualifizierte Assoziationen

- `user` verwaltet eigene Gruppen
  - `groupname` in Assoziation `manages` eindeutig
    - aber nicht über alle Gruppen
    - kann so nicht ausgedrückt werden

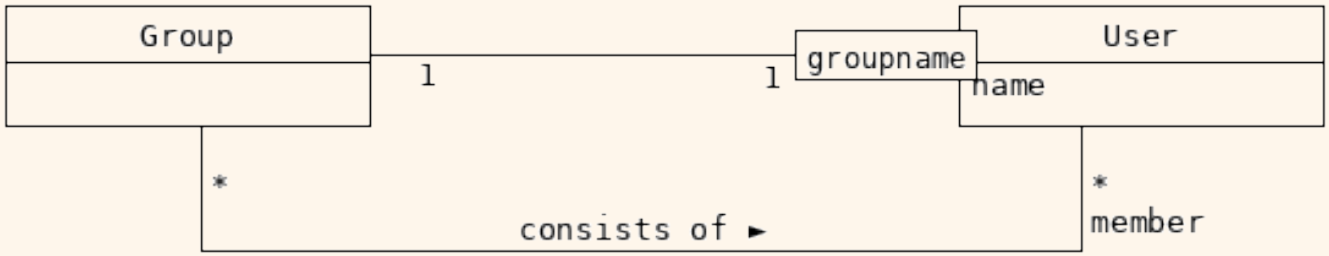


# Qualifizierte Assoziationen

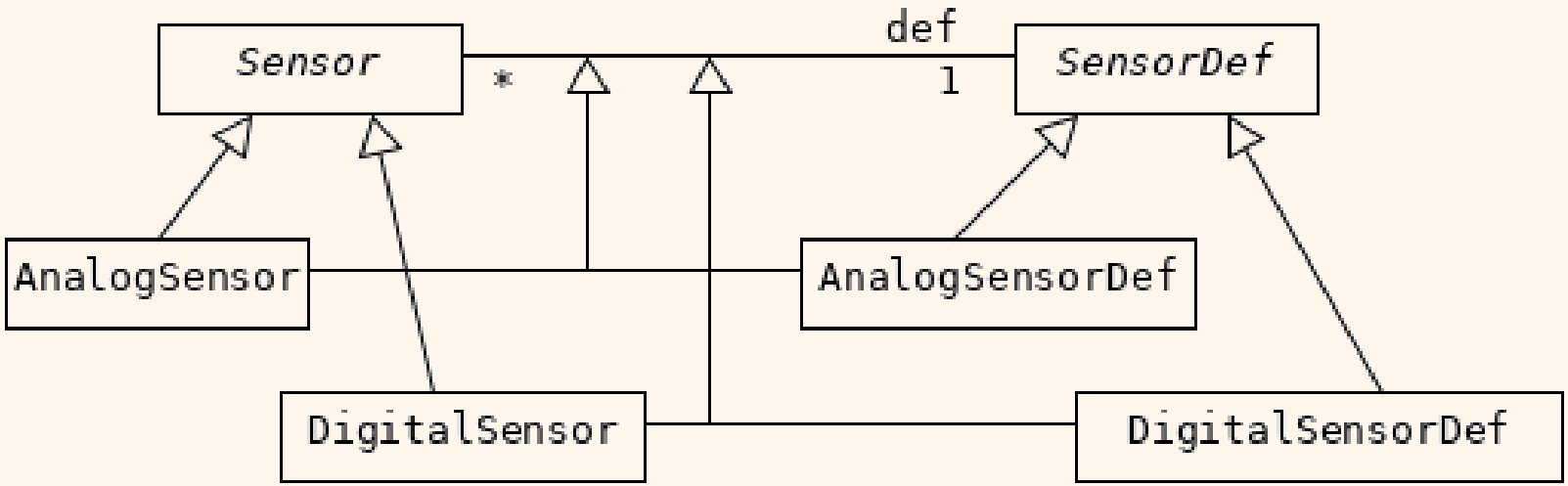
- `user` verwaltet eigene Gruppen
  - `groupname` in Assoziation `manages` eindeutig
    - aber nicht über alle Gruppen
    - kann so nicht ausgedrückt werden



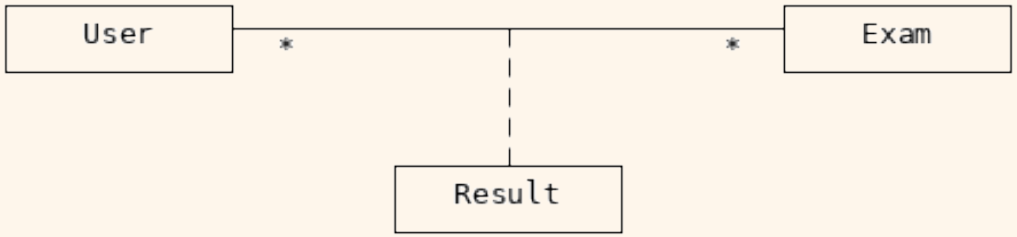

- $\rightarrow$ qualifizierte Assoziation

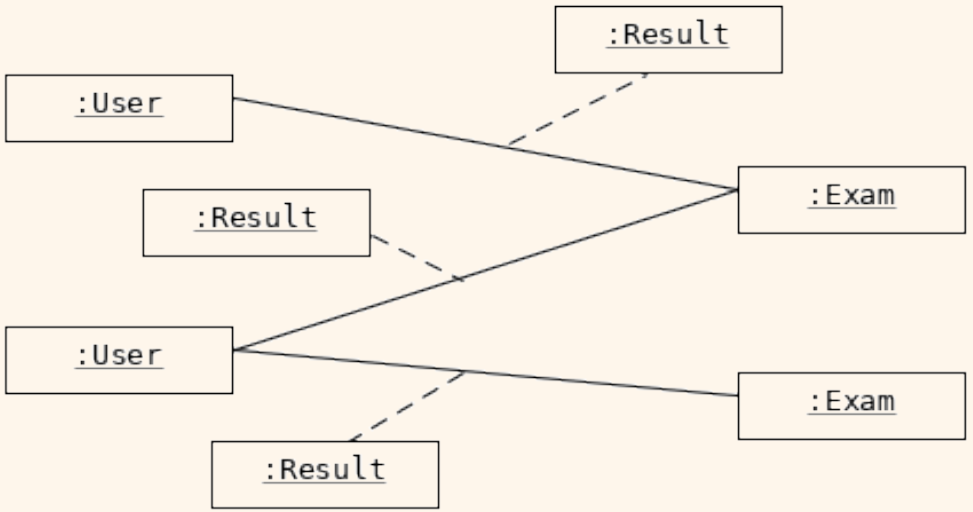


# Vererbung und Assoziationen



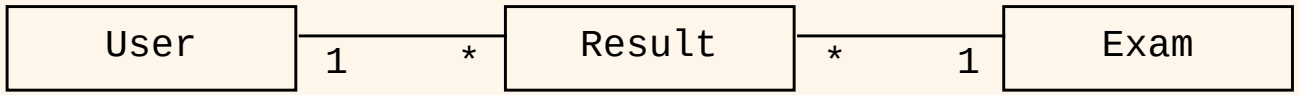
# Assoziationsklasse



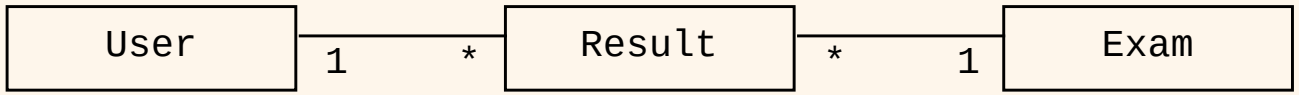
# Assoziationsklasse – 2

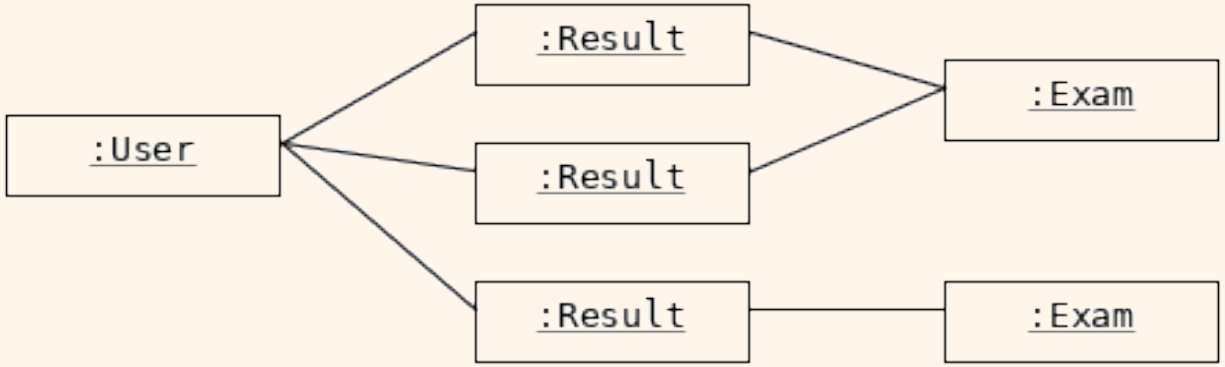
Alternative Modellierung mittels ”normaler” Klassen?



# Assoziationsklasse – 2

Alternative Modellierung mittels ”normaler” Klassen?



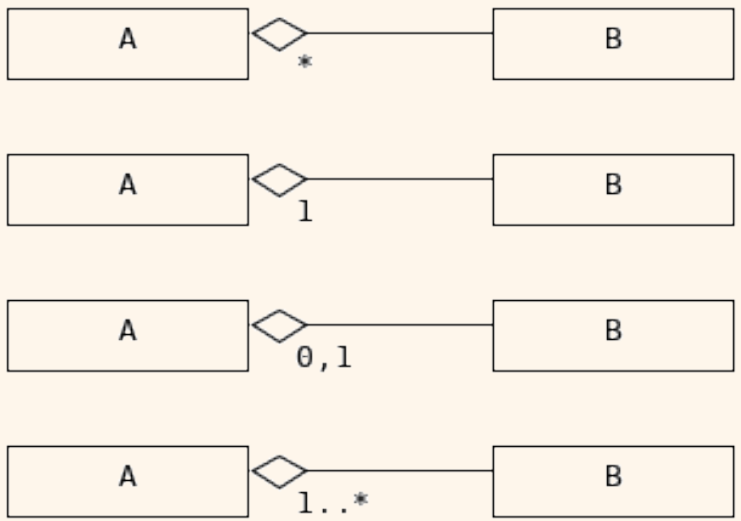



# Aggregation

- Spezialfall der Assoziation
  - $\rightarrow$ ”part-of” Beziehung
  - Definition: *Komponentenobjekt* part-of *komplexes Objekt*
- keine Zyklen auf Instanzebene!
- Semantik der Aggregation bzw. Komposition

| | |
|---|---|
| abhängig | abhängiges Komponentenbjekt ist existenzabhängig von komplexen Objekt |
| unabhängig | unabhängige Objekte können losgelöst existieren |
| exklusiv | Komponentenobjekt hat genau part-of-Beziehung komplexen Objekt |
| nicht exklusiv | Komponentobjekt kann mehrere part-of-Beziehungen zu unterschiedlichen komplexen Objekten haben |

# Aggregation – 2




unabhängig, nicht exklusiv

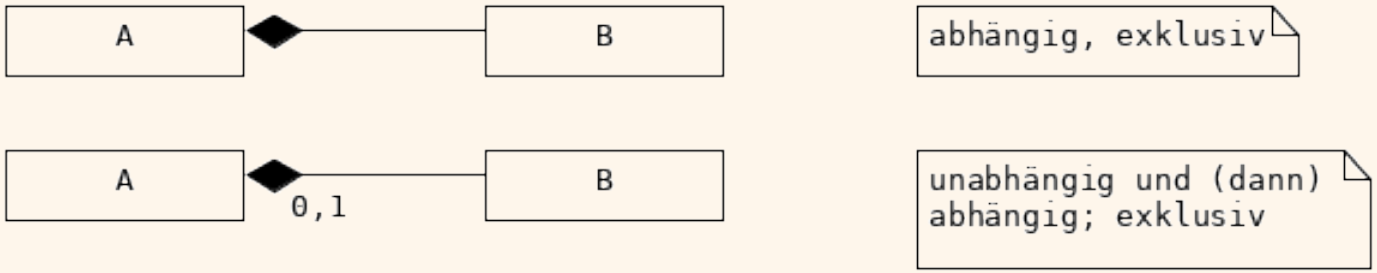
abhängig, exklusiv,
aber keine propagierende
Löschsemantik

unabhängig, exklusiv

abhängig, nicht exklusiv,
+ benutzerdefinierte
Einschränkungen bzgl. Lösch-
und Propagierungssemantik

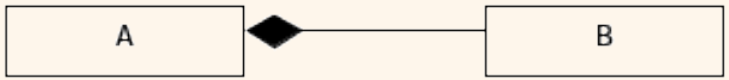
# Komposition

- Spezialfall der Assoziation
- zusätzliche Semantik:
  - Kardinalität: max. 1 auf Seite der Komposition
  - kaskadierende Operationen: Löschen,…

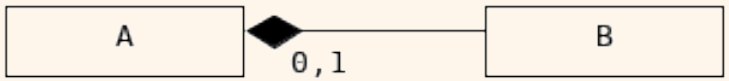


# Komposition

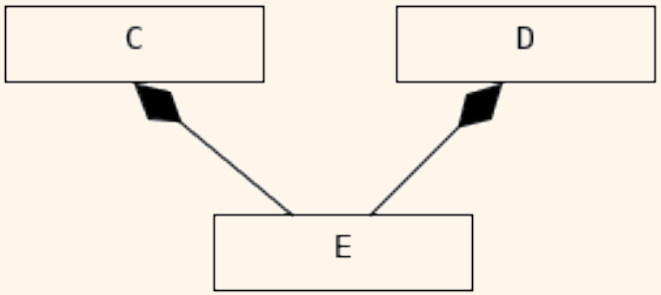
- Spezialfall der Assoziation
- zusätzliche Semantik:
  - Kardinalität: max. 1 auf Seite der Komposition
  - kaskadierende Operationen: Löschen,…




abhängig, exklusiv

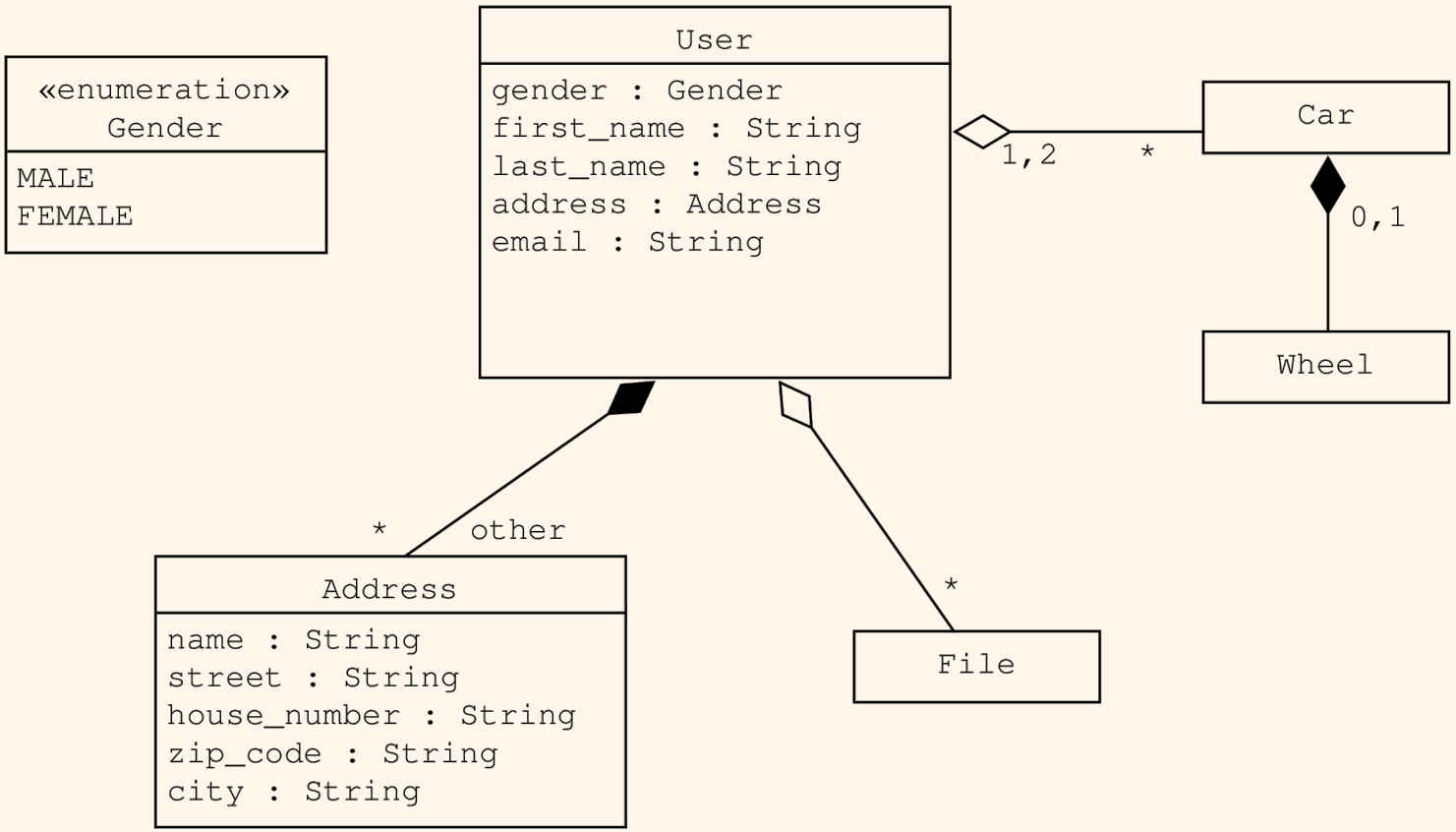



unabhängig und (dann)
abhängig; exklusiv

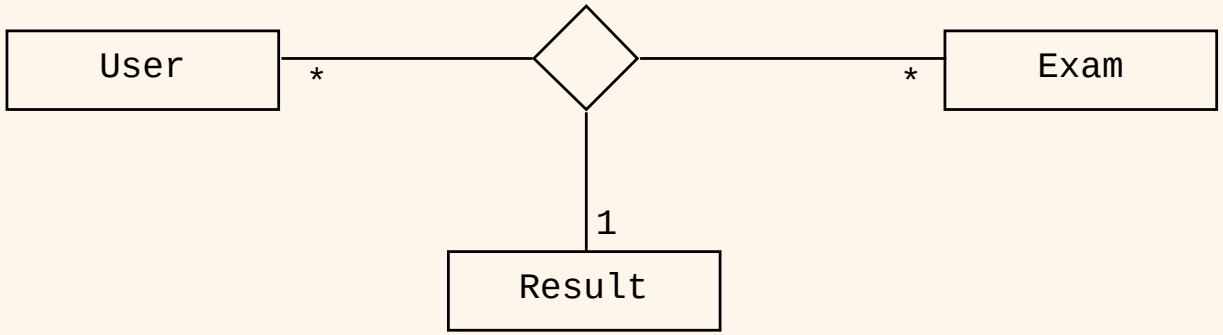


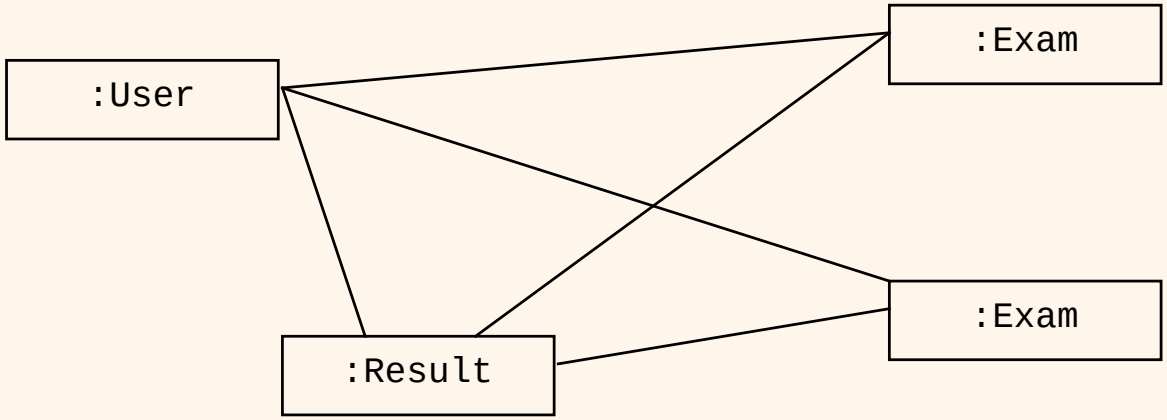

ungültig!

# Aggregation, Komposition



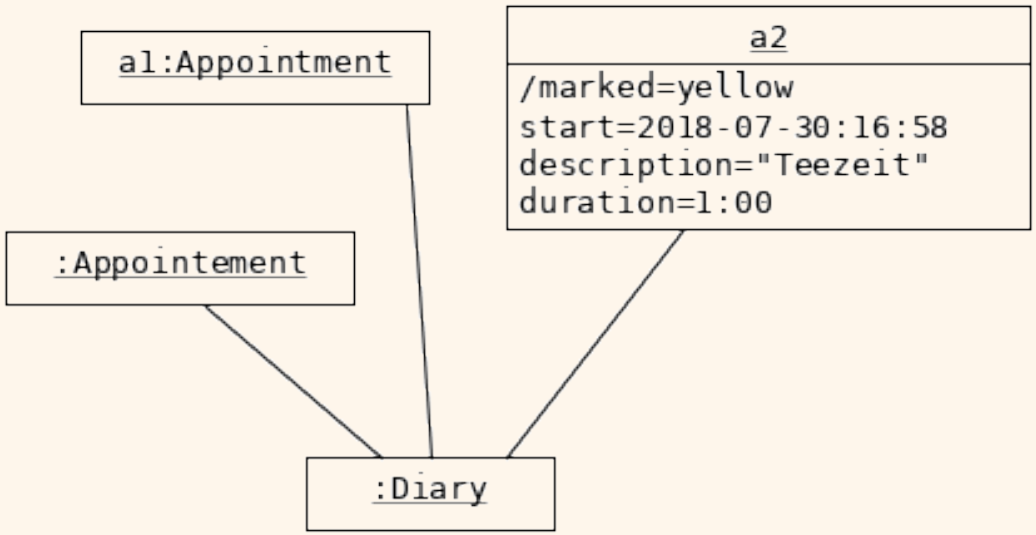
# N-wertige Assoziationen



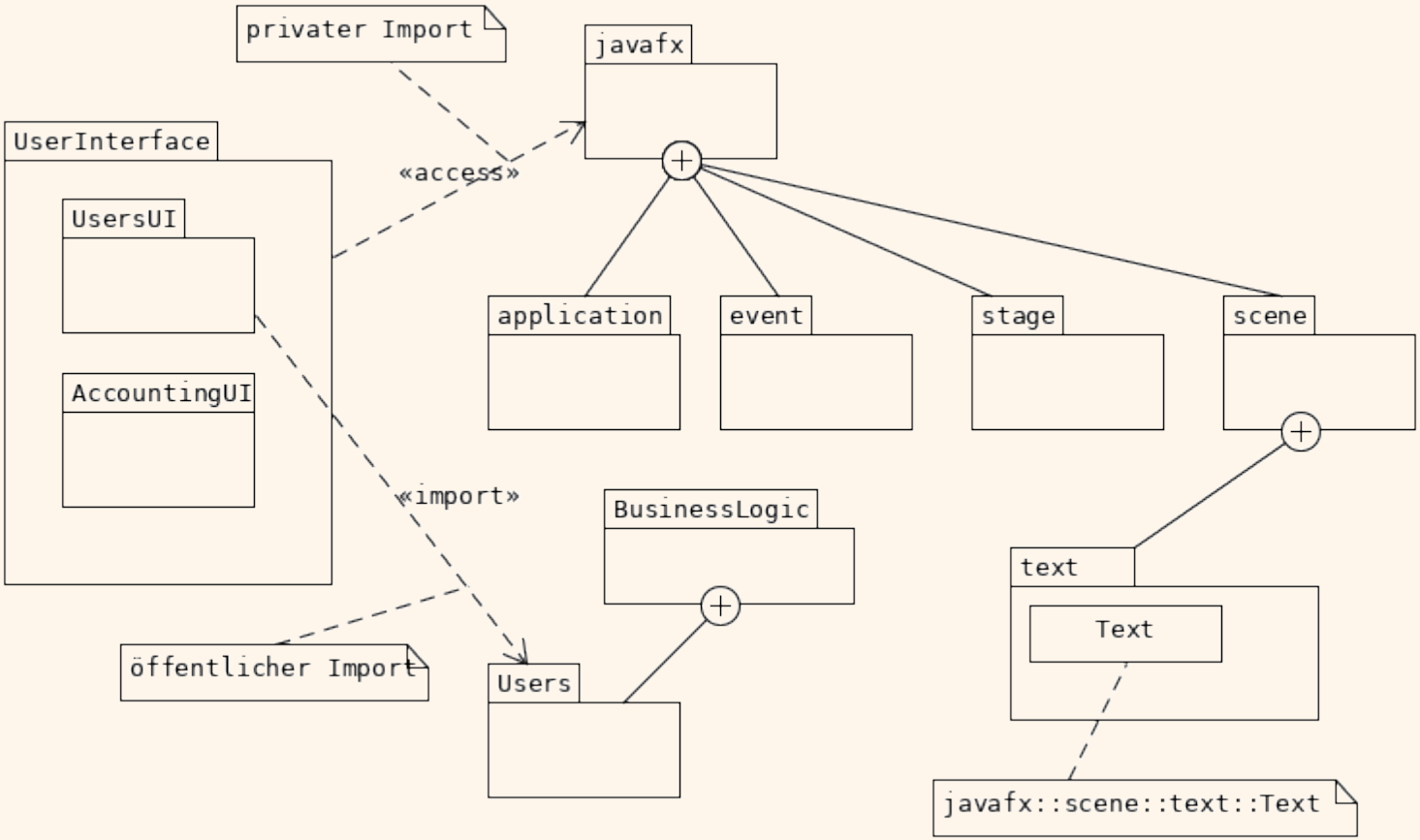
# Objektdiagramm

- Zeigt
  - Objekte (d.h. Instanzen von Klassen)
  - Links (d.h. Instanzen von Assoziationen)



# Paketdiagramm

- Zeigt Pakete mit Klassen,…



# Verteilungsdiagramm

- Zeigt eingesetzte Hard- und Softwaretopologie und das eingesetzte Laufzeitsystem