Objektorientierte Datenbanken

- Die regelmäßige tabellen-strukturierte Form relationaler Daten impliziert eine Reihe von Vorteilen.

  Dies betrifft auch die Implementierung:

  - Tabellen (und Records) lassen sich einfach auf den block-strukturierte Sekundärspeicher abbilden und typische Operationen auf Tabellen führen zu effizient unterstützten Zugriffsmustern (z.B. *table scans*) auf solchen Speichern.

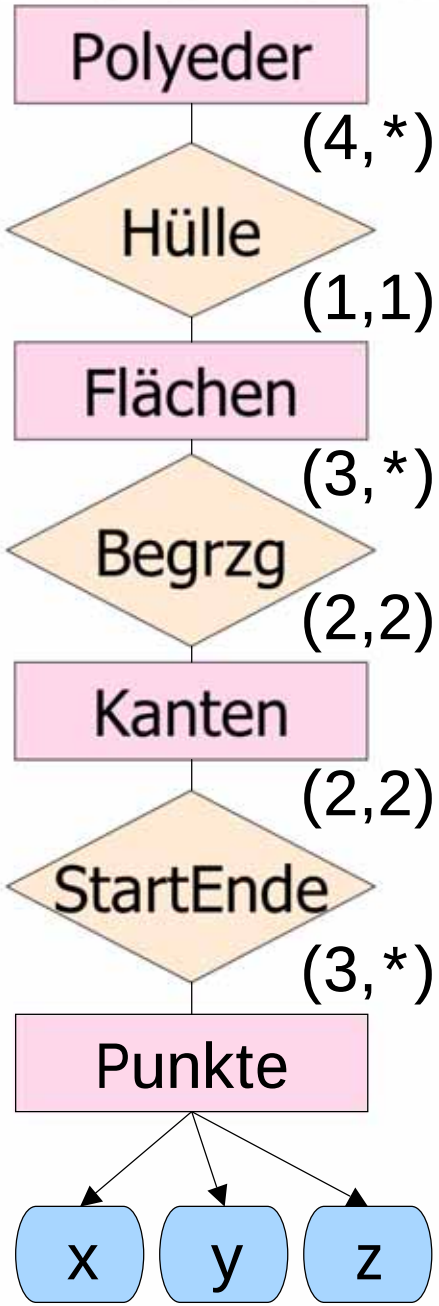
- **Komplexe Datentypen** lassen sich jedoch nicht immer einfach auf Tabellen abbilden (Multimedia-Daten, ingenieurs-wissenschaftliche Anwendungen, CAD-Objekte, ...).

  - Relationale Modellierung dieser Daten verteilt **ein Objekt** typischerweise auf **viele Tabellen** oder würde eine Erweiterung des relationalen Modelles benötigen (NF^2).

Objektorientierte Datenbanken

- Dies führte zu Beginn der 80er-Jahre zur Entwicklung von **Objekt(-orientierten) Datenbanken** (**OODB**), in denen Objekte die Rolle der Relationen als zentrales Konzept einnehmen.
  - Objekte in OODBs können komplexe interne Strukturen besitzen. OODBs unterstützen Objekt**beziehungen** (*is-part-of*), **Vererbung** (*is-a*) und speichern **Methoden** zusammen mit den Objektdaten.
  - OODBs bevölkern lediglich eine Nische im Datenbank-Markt (ca. 10 kommerzielle Produkte).
  - Aber: OODB-Konzepte haben ihren Weg in RDBMS gefunden (Ü objekt-relationale DBMS, nächstes Kapitel).

Nachteile relationaler Modellierung



Polyeder			
PolyID	Gewicht	Material	. . .
cubo#5	25.765	Eisen	. . .
tetra#7	37.985	Glas	. . .
. . .	. . .	. . .	. . .

Flächen		
FlächenID	PolyID	Oberfläche
f1	cubo#5	. . .
f2	cubo#2	. . .
. . .	. . .	. . .
f6	cubo#5	. . .
f7	tetra#7	. . .

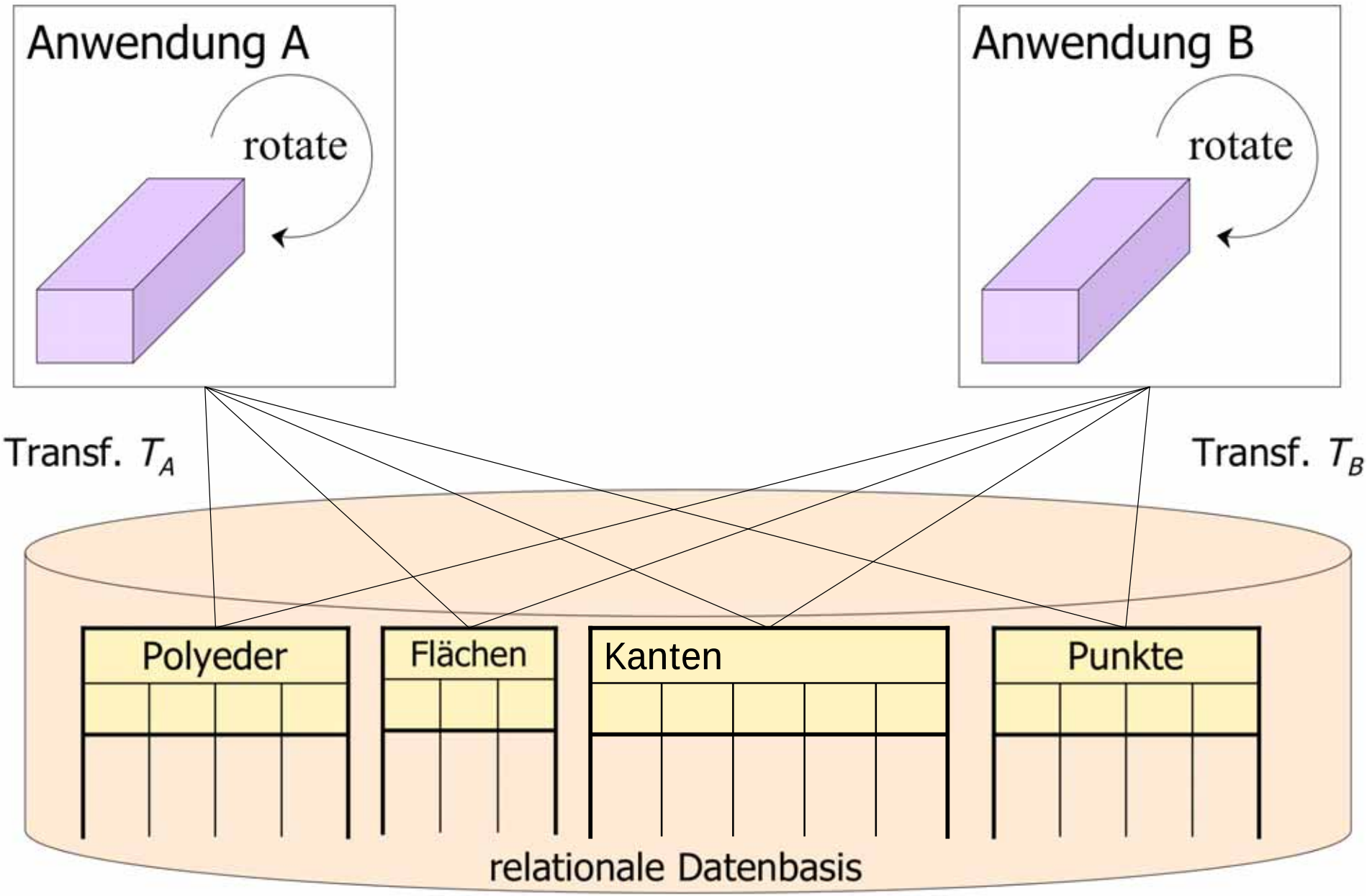
Kanten				
KantenID	F1	F2	P1	P2
k1	f1	f4	p1	p4
k2	f1	f2	p2	p3
. . .	. . .	. . .		

Kanten			
PunktID	X	Y	Z
p1	0.0	0.0	0.0
p2	1.0	0.0	0.0
. . .	. . .	. . .	

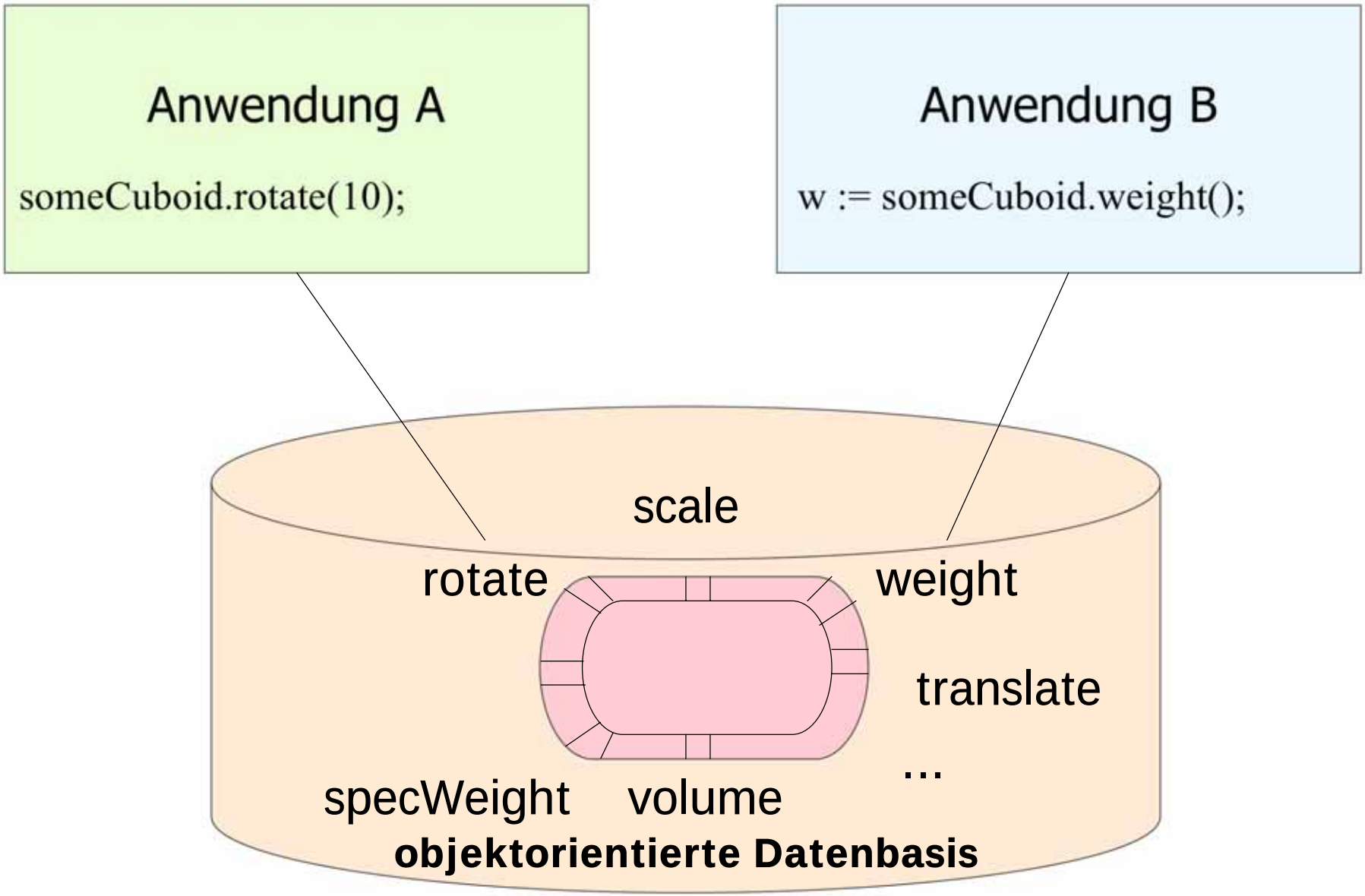
Nachteile relationaler Modellierung

- **Segmentierung**
  (ein Objekt, viele Tabellen)
- **Künstliche Schlüsselattribute**
  (Fremdschlüsselbeziehungen)
- **Fehlendes Verhalten**
  (lediglich Tupeloperationen sind direkt anwendbar)
- **Externe Programmierschnittstelle**
  (*PL Embedding* notwendig, um komplexe Operationen zu implementieren)

Visualisierung des „Impedance Mismatch“



Vorteile objektorientierter Datenmodellierung



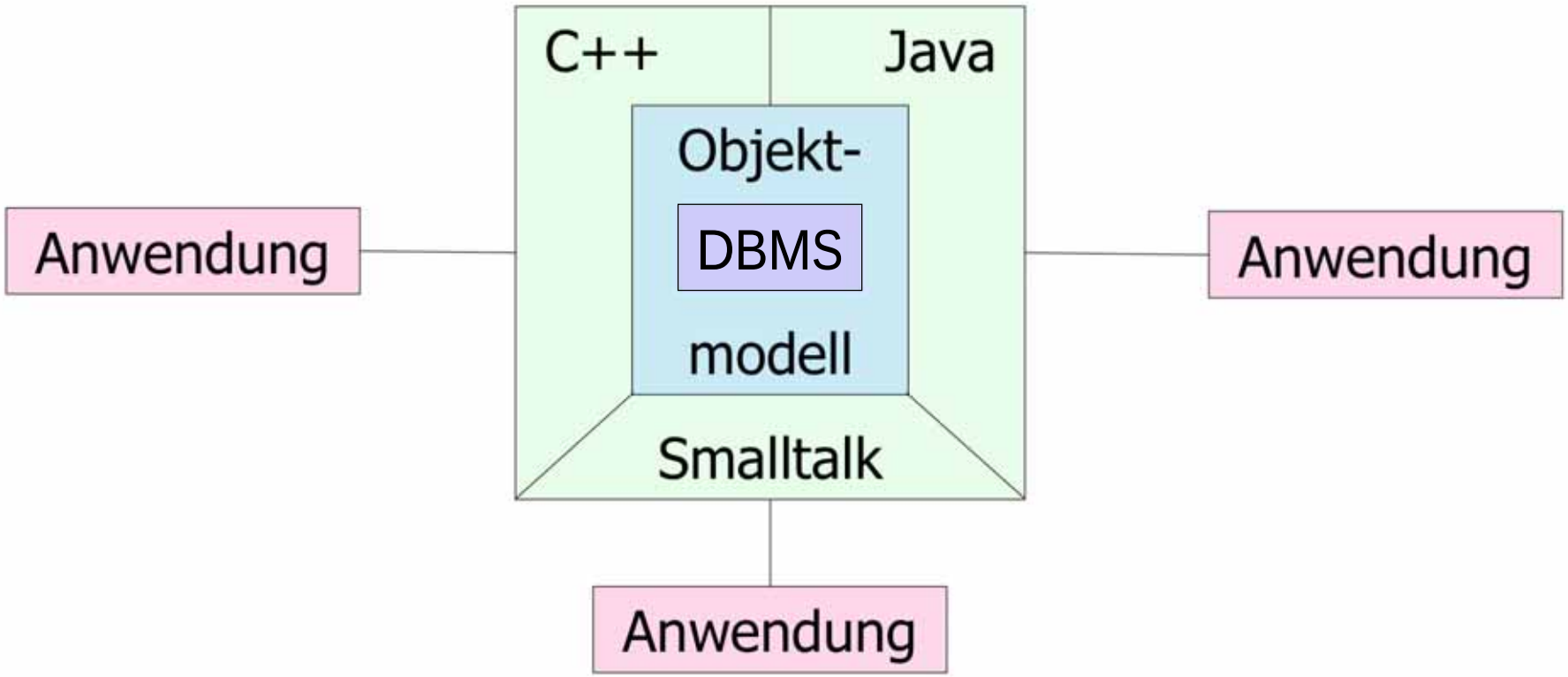
Vorteile objektorientierter Datenmodellierung

- In OODBs wird ein komplexes Objekt als integrierte eingekapselte Einheit deklariert, angefragt, und manipuliert.
  - Die Einkapselung verbirgt die strukturelle Repräsentation des Objektes (***information hiding***)
  - Operationen (**Methoden**) werden in einer Sprache formuliert, die die Konzepte des Objektmodells versteht
  - Die Signaturen der Methoden bildet das einzige externe Interface der Objekte
  - Die notwendige Transformation zwischen Primär- und Sekundärspeicherrepräsentation der Objekt ist transparent
  - Die OODB verwaltet eine system-weite **Objektidentität**, die zur Referenzierung und damit zum Aufbau komplexer Objektnetzwerke genutzt werden kann

Der ODMG-Standard

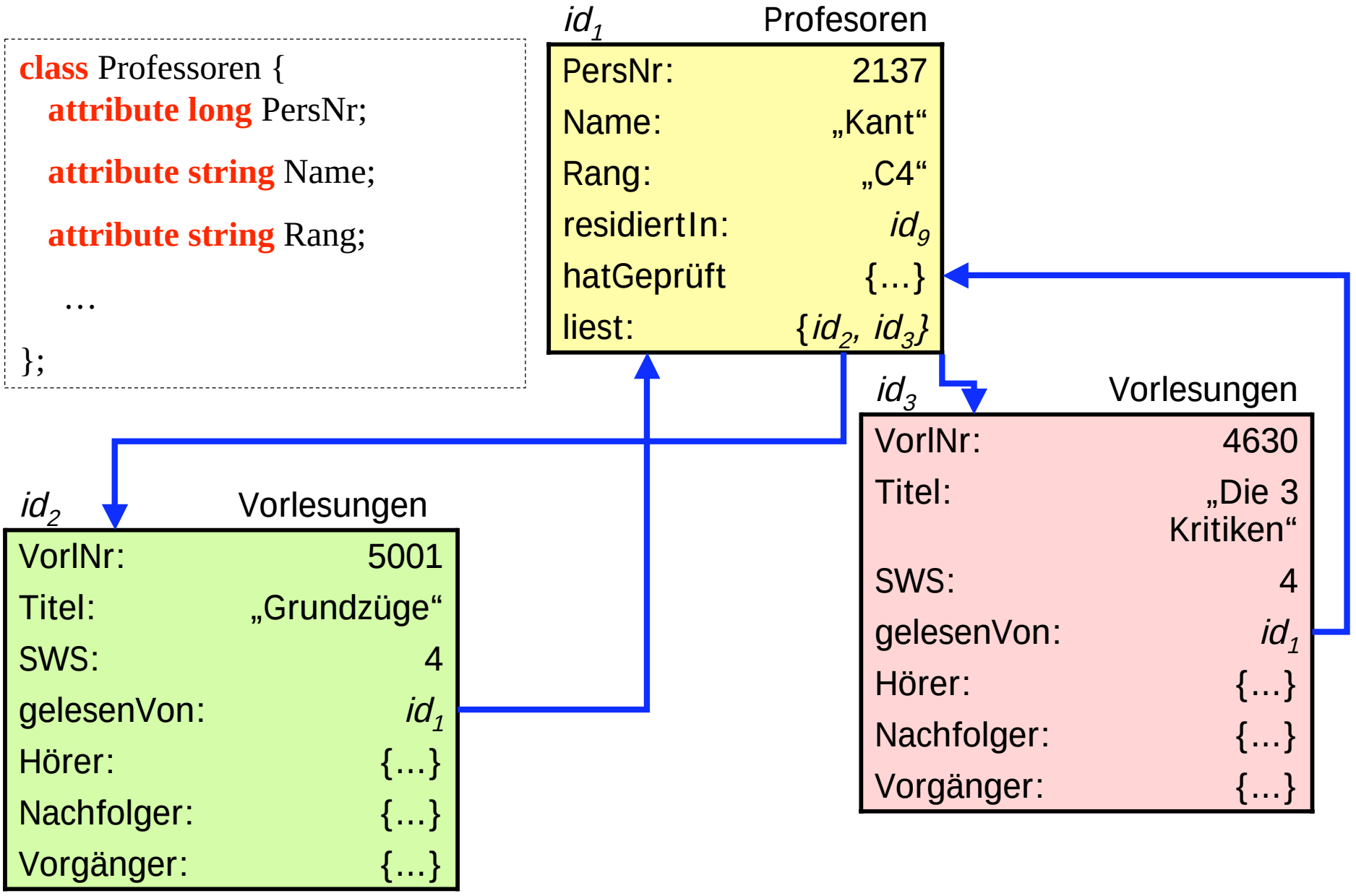
- 1993 beginnt die ***Object Database Management Group*** (**ODMG**), einen Standard zu definieren, der ein Objektmodell (OM) und die damit assoziierten Sprachen definiert.
  - ODMG besteht aus einer Reihe von DBMS-Herstellern und einer Gruppe von sog. *Reviewers.*
  - Komponenten des ODMG-Standards:
    - Ein Kern-Objektmodell (Objektidentität, Vererbung, ...)
    - *Object Definition Language* (ODL)
    - *Object Query Language* (OQL)
    - Java/C++/Smalltalk-Bindings für das OM

Integration des ODMG-Objektmodells



Einige Objekte aus der Universitätswelt

```
class Professoren {
   attribute long PersNr;
   attribute string Name;
   attribute string Rang;
     …
};
```



Objektidentität

- Relationales Modell: Identifikation über **Werte** der Schlüsselattribute (*identity through contents*)
  - Objekte gleichen Wertes müssen nicht identisch sein
  - è Einführung künstlicher Schlüsselattribute (z.B.: KantenID) ohne Bedeutung in der Anwendung
  - Schlüssel müssen unveränderbar sein (*dangling references, object "rebirth"*)
- Programmiersprachen: Identifikation von Objekten durch **Speicheradressen** (*pointer*)
  - Physisches Bewegen des Objektes unmöglich, Probleme bei persistenten Objekten
  - *Object rebirth* durch Wiederbenutzung von Speicherplatz (nach *garbage collection*)

Objektidentität

- Objektidentität in OODB:
  - Systemseitig verwaltete OIDs, OODB verwaltet Pool von verfügbaren OIDs
  - OID invariant während Lebenszeit seines Objektes
  - Stabile Referenzierung über OIDs möglich
  - OIDs in diesen Folien: id_1, id_2, ...

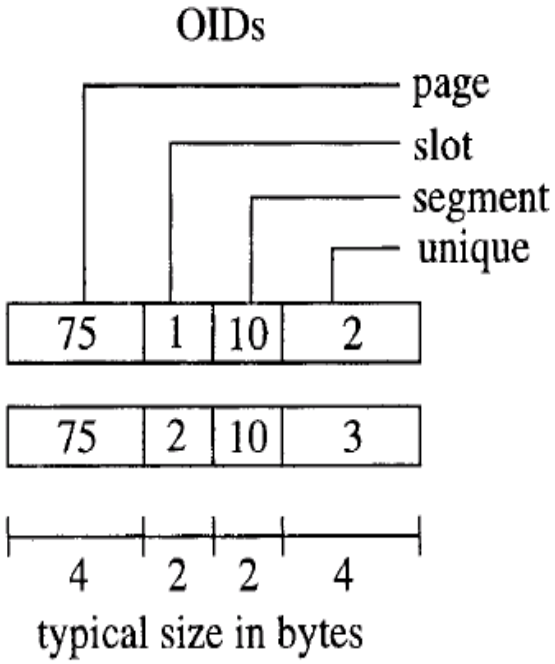
Objektidentität - Realisierung

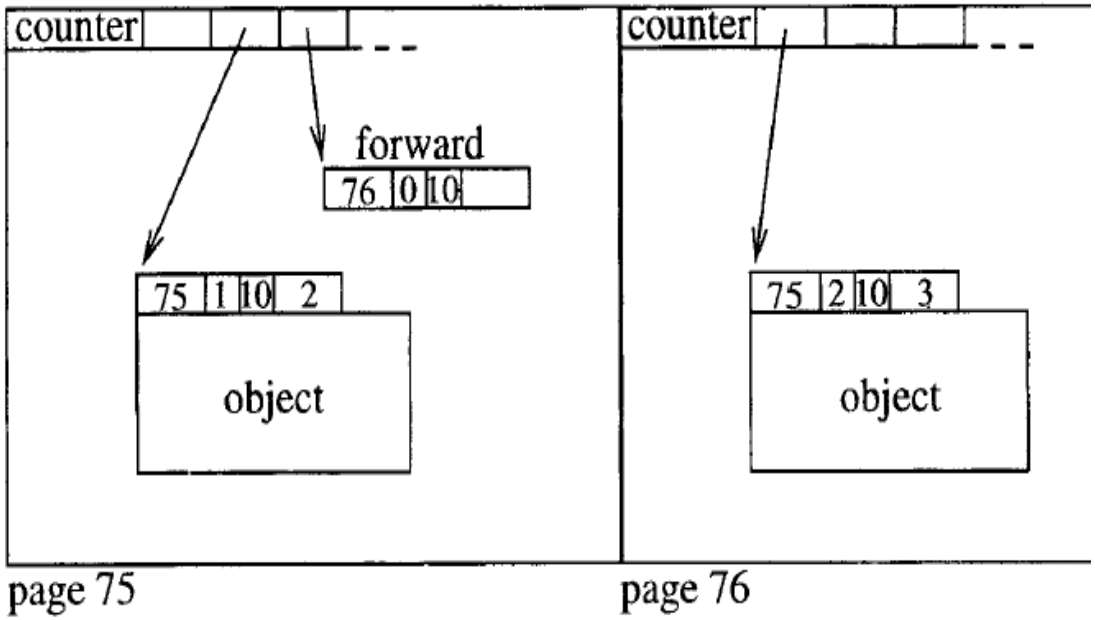
- Im wesentlichen zwei Realisierungen für OIDs in OODB:

1. **Physische** OIDs
   - Enthalten den Speicherort des Objektes
   - Im wesentlichen entsprechen diese den Tupel-Identifikatoren (TIDs) der relationalen Welt

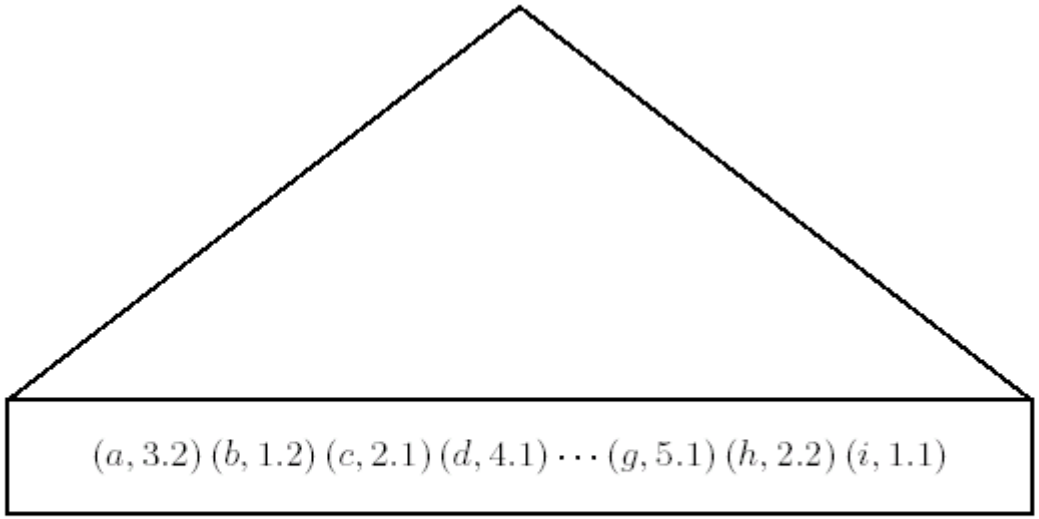
2. **Logische** OIDs
   - Unabhängig vom Speicherort der Objekte
   - Objekte können verschoben/geclustered werden
   - Indirektion über eine *Mapping*-Struktur:
     - B-Baum
     - Hash-Tabelle
     - *Direct Mapping*
   - ***Swizzling****:* Übersetzung logische OIDs Referenzen ⇔ Primärspeicher

Realisierung physischer OIDs





Realisierung logischer OIDs



(a) B^+-tree

$(g, 5.1)$
$(e, 3.1)$
$(b, 1.2)$
$(f, 4.2)$
$(d, 4.1)$
$(c, 2.1)$
$(a, 3.2)$
$(h, 2.2)$
$(i, 1.1)$

(b) Hash Table

a	3.2
b	1.2
c	2.1
d	4.1
e	3.1
f	4.2
g	5.1
h	2.2
i	1.1

(c) Direct Mapping

Fig. 2. Mapping techniques

Definition von Objekttypen

- Eine Objekttyp-Definition (formuliert in ODL)
  - legt die **strukturelle Beschreibung** (Attribute, Beziehungen) der Objekte eines Typs fest,
  - beschreibt das **Verhalten** der Objekte des Typs durch eine Menge von Operationen,
  - setzt den Typ in Beziehung zu anderen Objekttypen des Systems (**Vererbung**: Generalisierung, Spezialisierung)

class *objecttype* extends *objecttype* {

attribute *type* a_1;

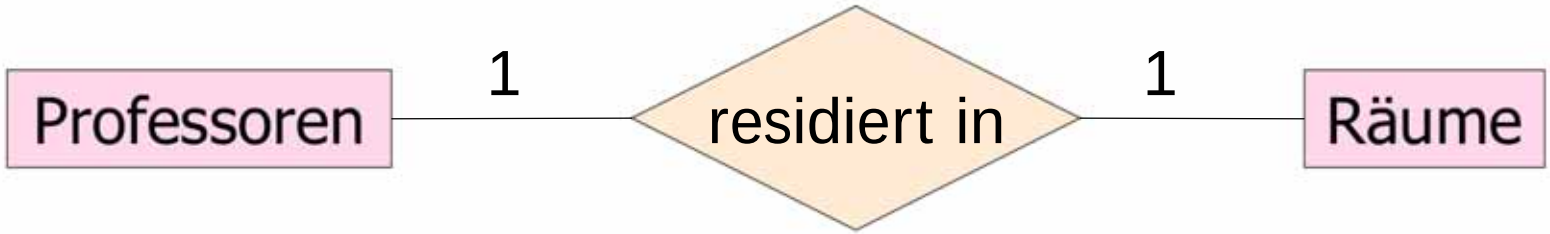
attribute *type* a_2;

…

relationship *objecttype* r;

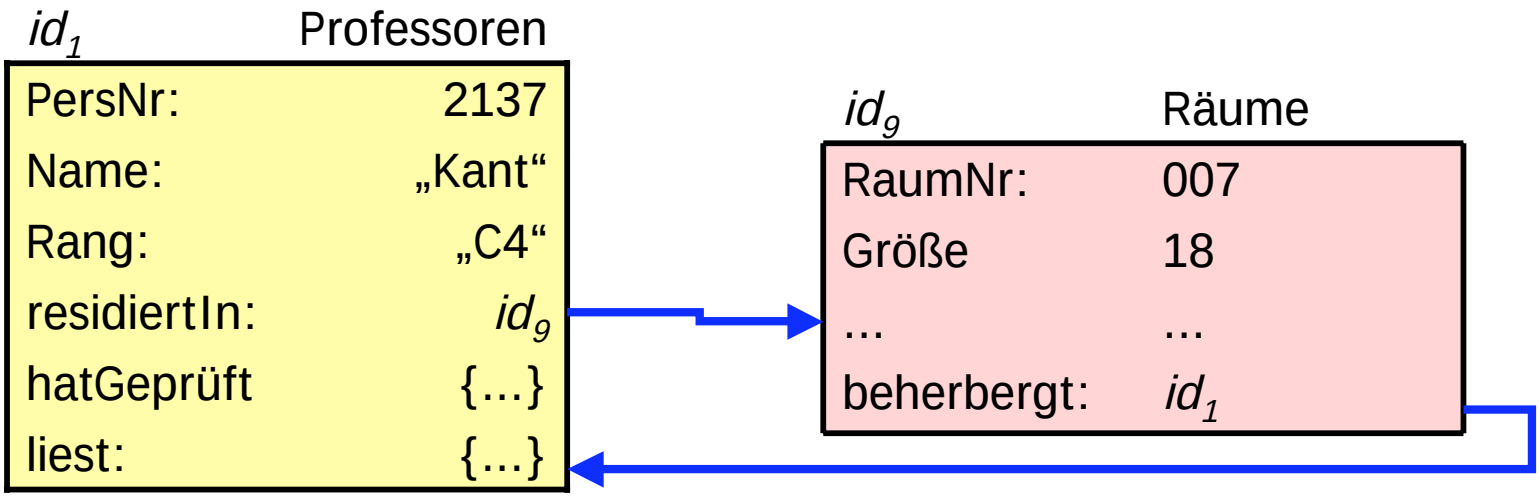
}

Objekttypen: 1 : 1-Beziehungen

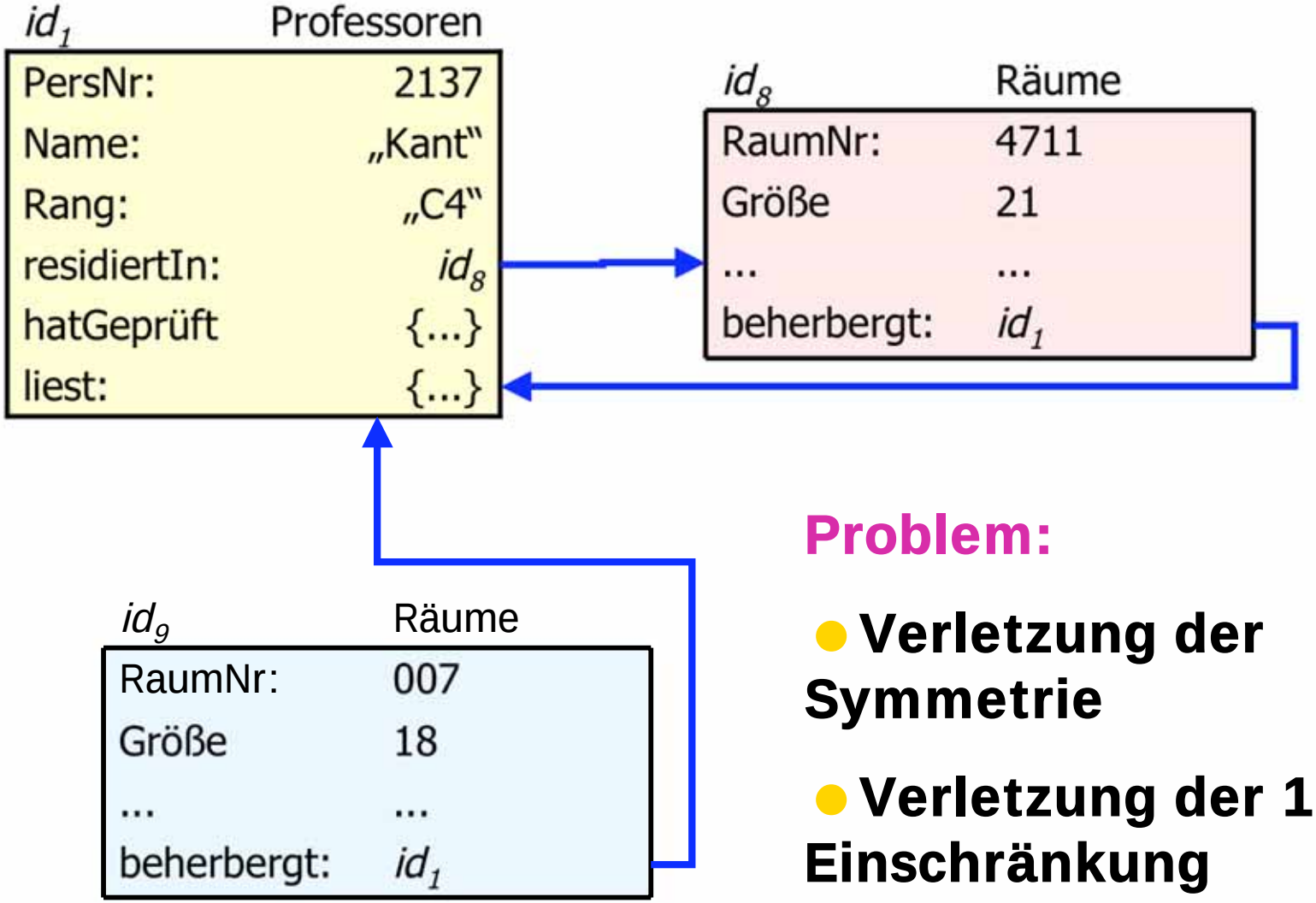



```
class Professoren {
    attribute long PersNr;
    ...
    relationship Räume residiertIn;
};
class Räume {
    attribute long RaumNr;
    attribute short Größe;
    ...
    relationship Professoren beherbergt;
};
```

Beispielausprägungen (Instanzen)



Beispielausprägungen



Problem:

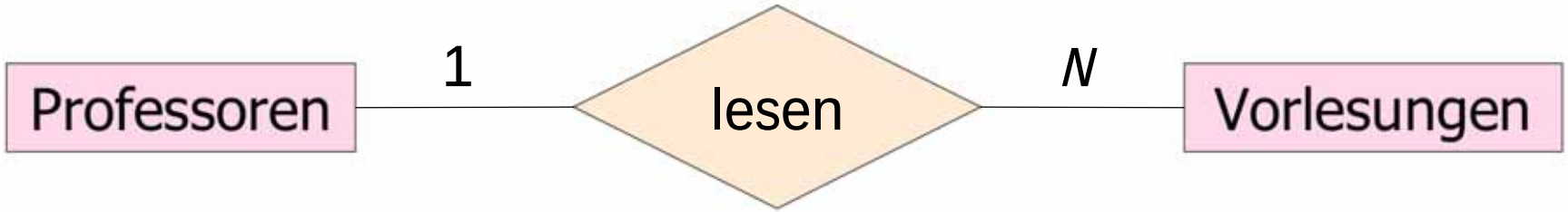
- **Verletzung der Symmetrie**
- **Verletzung der 1:1-Einschränkung**

Bessere Modellierung mittels inverse

```
class Professoren {
        attribute long PersNr;
        ...
        relationship Räume residiertIn inverse Räume::beherbergt;
};
class Räume {
        attribute long RaumNr;
        attribute short Größe;
        ...
        relationship Professoren beherbergt inverse Professoren::residiertIn
};
```

$$p.\text{residiertIn} = r \Leftrightarrow r.\text{beherbergt} = p$$

1 : *N*-Beziehungen



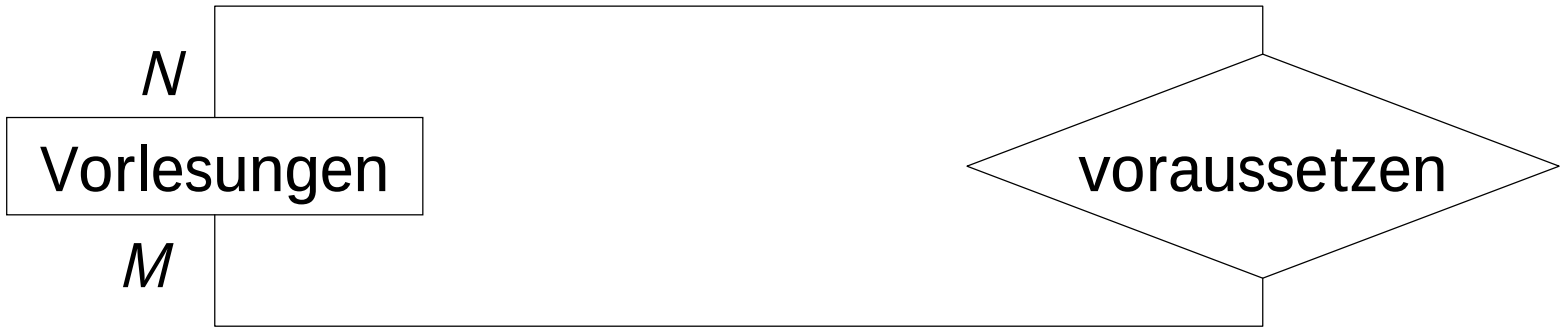

```
class Professoren {
    ...
    relationship set(Vorlesungen) liest inverse
    Vorlesungen::gelesenVon;
};
class Vorlesungen {
    ...
    relationship Professoren gelesenVon inverse
    Professoren::liest;
};
```

N : *M*-Beziehungen


```
class Studenten {
  ...
  relationship set(Vorlesungen) hört inverse Vorlesungen::Hörer;
};
class Vorlesungen {
  ...
  relationship set(Studenten) Hörer inverse Studenten::hört;
};
```

$$s \in v.\text{Hörer} \Leftrightarrow v \in s.\text{hört}$$

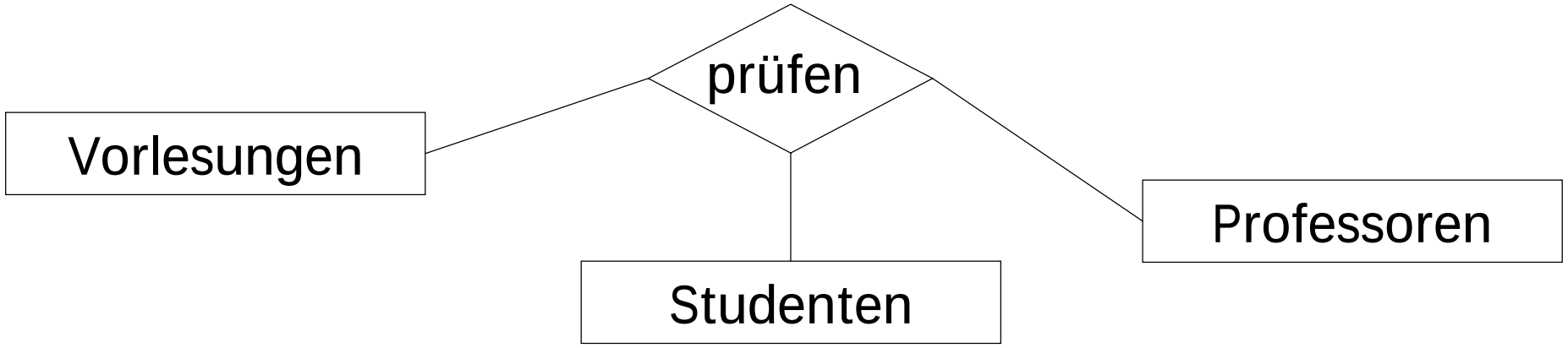
Rekursive *N* : *M*-Beziehungen




```
class Vorlesungen {
  ...
  relationship set(Vorlesungen) Vorgänger inverse
  Vorlesungen::Nachfolger;

  relationship set(Vorlesungen) Nachfolger inverse
  Vorlesungen::Vorgänger;
};
```

Ternäre Beziehungen

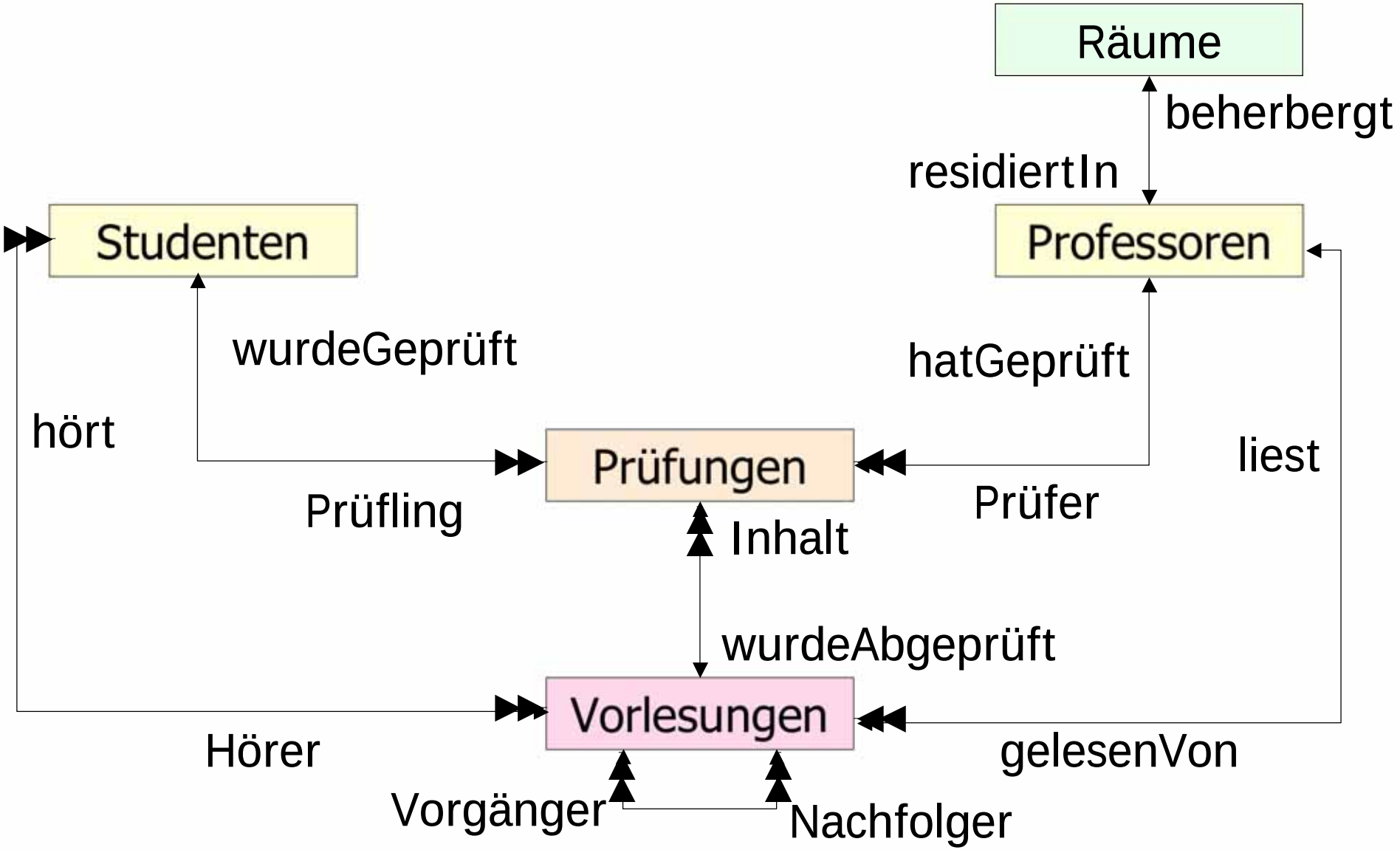



```
class Prüfungen {
    attribute struct Datum
        { short Tag; short Monat; short Jahr } Prüfdatum;
    attribute float Note;
    relationship Professoren Prüfer inverse
    Professoren::hatGeprüft;
    relationship Studenten Prüfling inverse Studenten::wurdeGeprüft;
    relationship Vorlesungen Inhalt inverse
    Vorlesungen::wurdeAbgeprüft;
};
```

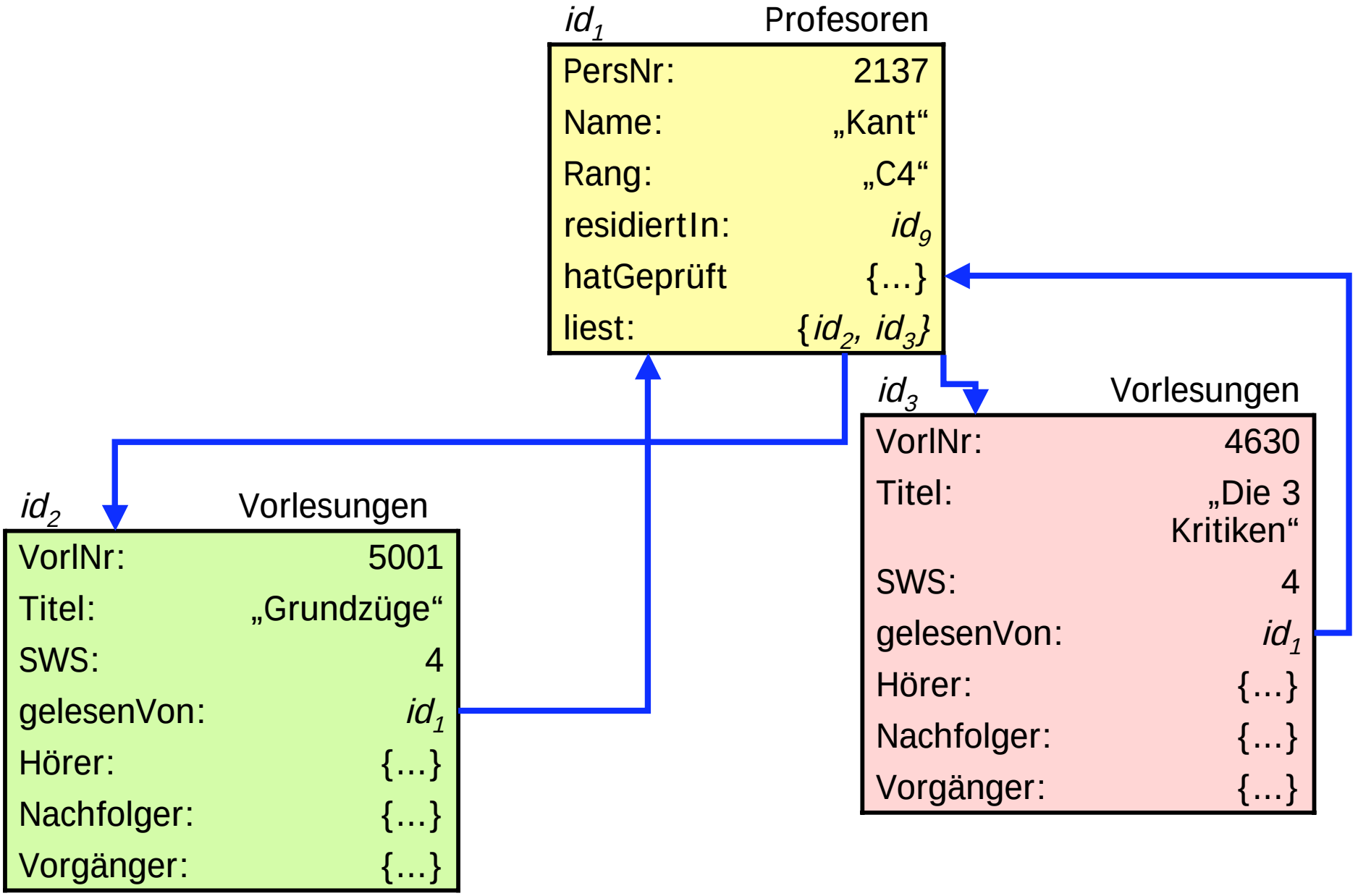
Vervollständigtes Universitäts-Schema

```
class Professoren {
        attribute long PersNr;
        attribute string Name;
        attribute string Rang;
        relationship Räume residiertIn inverse Räume::beherbergt;
        relationship set(Vorlesungen) liest inverse Vorlesungen::gelesenVon
        relationship set(Prüfungen) hatGeprüft inverse Prüfungen::Prüfer;
};
class Vorlesungen {
        attribute long VorlNr;
        attribute string Titel;
        attribute short SWS;
        relationship Professoren gelesenVon inverse Professoren::liest;
        relationship set(Studenten) Hörer inverse Studenten::hört;
        relationship set(Vorlesungen) Nachfolger inverse Vorlesungen::Vorgänger;
        relationship set(Vorlesungen) Vorgänger inverse Vorlesungen::Nachfolger;
        relationship set(Prüfungen) wurdeAbgeprüft inverse Prüfungen::Inhalt;
};
class Studenten {
        ...
        relationship set(Prüfungen) wurdeGeprüft inverse Prüfungen::Prüfling;
};
```

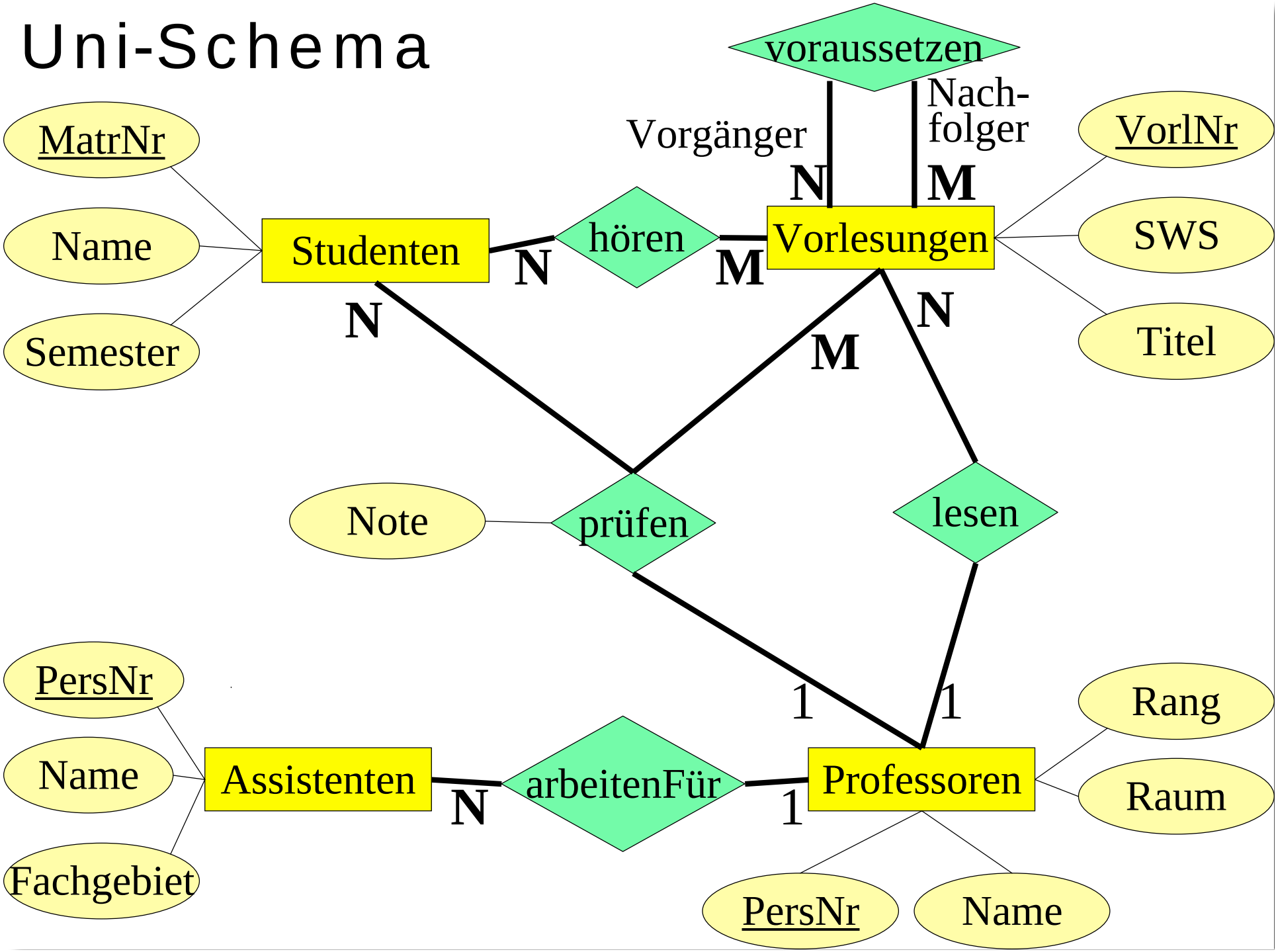
Modellierung von Beziehungen im Objektmodell



Einige Objekte aus der Universitätswelt



Uni-Schema


voraussetzen
Vorgänger
Nach-
folger
N
M
MatrNr
Name
Semester
Studenten
N
hören
M
Vorlesungen
VorlNr
SWS
Titel
N
M
N
Note
prüfen
lesen
1
1
PersNr
Name
Fachgebiet
Assistenten
N
arbeitenFür
1
Professoren
Rang
Raum
PersNr
Name

Professoren

PersNr	Name	Rang	Raum
2125	Sokrates	C4	226
2126	Russel	C4	232
2127	Kopernikus	C3	310
2133	Popper	C3	52
2134	Augustinus	C3	309
2136	Curie	C4	36
2137	Kant	C4	7

Studenten

MatrNr	Name	Semester
24002	Xenokrates	18
25403	Jonas	12
26120	Fichte	10
26830	Aristoxenos	8
27550	Schopenhauer	6
28106	Carnap	3
29120	Theophrastos	2
29555	Feuerbach	2

Vorlesungen

VorlNr	Titel	SWS	gelesen Von
5001	Grundzüge	4	2137
5041	Ethik	4	2125
5043	Erkenntnistheorie	3	2126
5049	Mäeutik	2	2125
4052	Logik	4	2125
5052	Wissenschaftstheorie	3	2126
5216	Bioethik	2	2126
5259	Der Wiener Kreis	2	2133
5022	Glaube und Wissen	2	2134
4630	Die 3 Kritiken	4	2137

voraussetzen

Vorgänger	Nachfolger
5001	5041
5001	5043
5001	5049
5041	5216
5043	5052
5041	5052
5052	5259

hören

MatrNr	VorlNr
26120	5001
27550	5001
27550	4052
28106	5041
28106	5052
28106	5216
28106	5259
29120	5001
29120	5041
29120	5049
29555	5022
25403	5022

prüfen

MatrNr	VorlNr	PersNr	Note
28106	5001	2126	1
25403	5041	2125	2
27550	4630	2137	2

Assistenten

PerslNr	Name	Fachgebiet	Boss
3002	Platon	Ideenlehre	2125
3003	Aristoteles	Syllogistik	2125
3004	Wittgenstein	Sprachtheorie	2126
3005	Rhetikus	Planetenbewegung	2127
3006	Newton	Keplersche Gesetze	2127
3007	Spinoza	Gott und Natur	2126

Typeigenschaften: Extensionen und Schlüssel

```
class Studenten (extent AlleStudenten key MatrNr) {
    attribute long MatrNr;
    attribute string Name;
    attribute short Semester;
    relationship set(Vorlesungen) hört inverse Vorlesungen::Hörer;
    relationship set(Prüfungen) wurdeGeprüft inverse
        Prüfungen::Prüfling;
};
```

- Extension (extent) dient als **Container** für alle Objekte eines Objekttyps und wird typischerweise in Anfragen referenziert
- Konstrukt key führt zusätzlich wert-basierten Schlüssel ein (eindeutig innerhalb des *extents*)

Modellierung des Verhaltens: Operationen

Operationen um ...

- Objekte zu erzeugen (instanziieren) und zu initialisieren,
- die für Klienten interessanten Teile des Zustands der Objekte zu erfragen,
- legale und konsistenzerhaltende Operationen auf diesen Objekten auszuführen und letztendlich
- die Objekte wieder zu zerstören.

Modellierung des Verhaltens: Operationen II

Drei Klassen von Operationen:

- *Beobachter* (*observer*)*:*
  - oft auch Funktionen genannt
  - Erfragen den Objektzustand
  - Beobachter-Operationen haben keinerlei objektändernde Seiteneffekte

- *Mutatoren:*
  - Änderungen am Zustand der Objekte.
  - Einen Objekttyp mit mindestens einer Mutator-Operation bezeichnet man als *mutierbar*
  - Objekte eines Typs ohne jegliche Mutatoren sind unveränderbar (engl. *immutable*)
  - Unveränderbare Typen bezeichnet man oft als **Literale** oder **Wertetypen**

Modellierung des Verhaltens: Operationen III

- *Konstruktoren und Destruktoren:*
  - Konstruktoren werden verwendet, um neue Objekte eines bestimmten Objekttyps zu erzeugen (Instantiierung)
  - Der Destruktor wird dazu verwendet, ein existierendes Objekt auf Dauer zu zerstören (OID ist *dauerhaft* zu invalidieren!)
  - Konstruktoren werden auf einen Objekttyp angewandt, um ein neues Objekt zu erzeugen
  - Destruktoren werden hingegen auf existierende Objekte angewandt

Klassen-Definition von Operationen in ODL

Man spezifiziert

- den Namen der Operation;
- die Anzahl und die Typen der Parameter;
- den Typ des Rückgabewerts der Operation;
- eine eventuell durch die Operationsausführung ausgelöste *Ausnahmebehandlung* (*exception handling*).

Klassen-Definition von Operationen in ODL

Beispiel-Operationen

```
class Professoren {
    exception hatNochNichtGeprüft {};
    exception schonHöchsteStufe {};
    ...
    float wieHartAlsPrüfer() raises (hatNochNichtGeprüft);
    void befördert() raises (schonHöchsteStufe);
};
```

Aufruf der Operationen

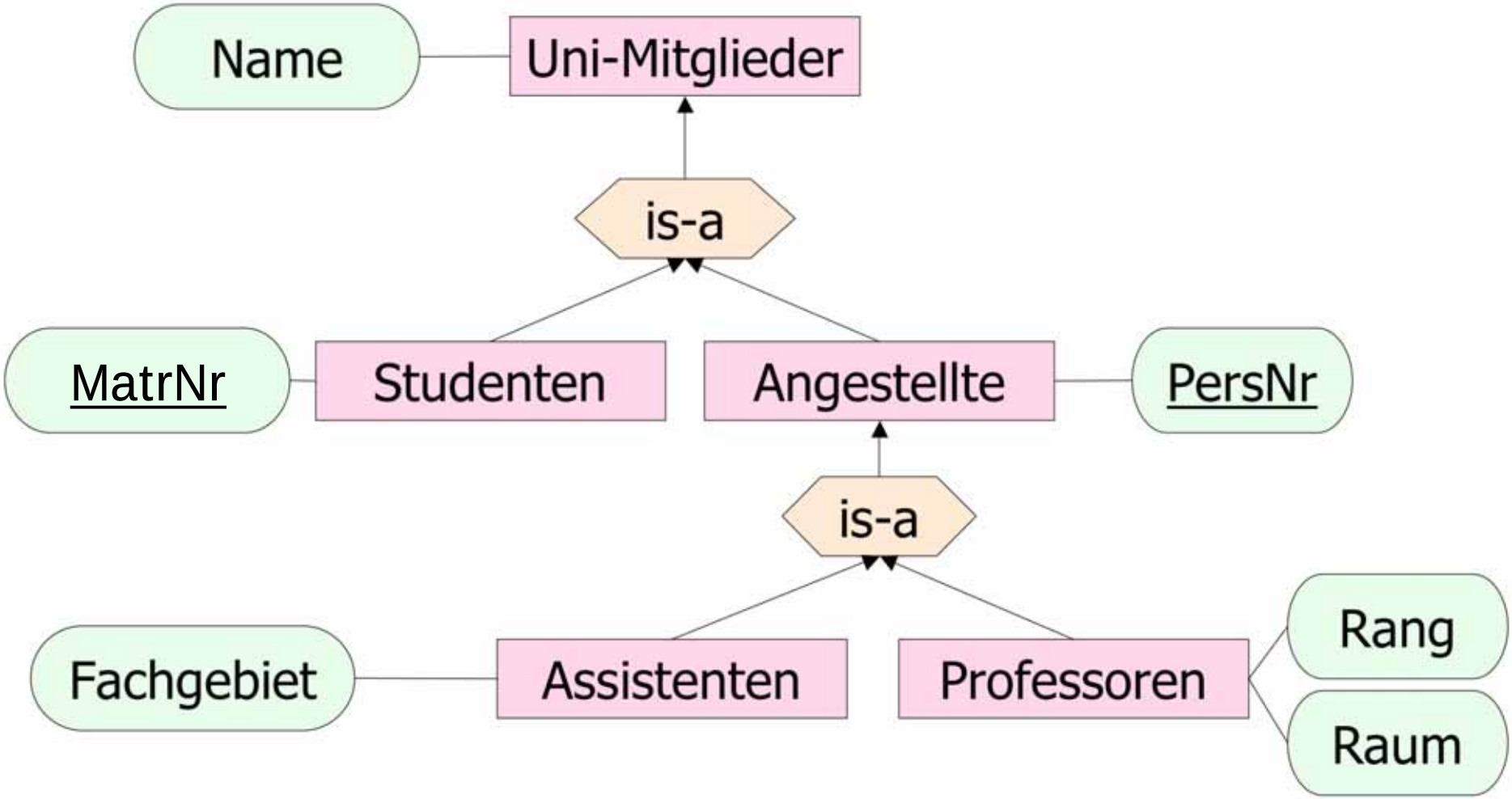
In einer OOPL:

```
meinLieblingsProf.befördert();
```

In OQL:

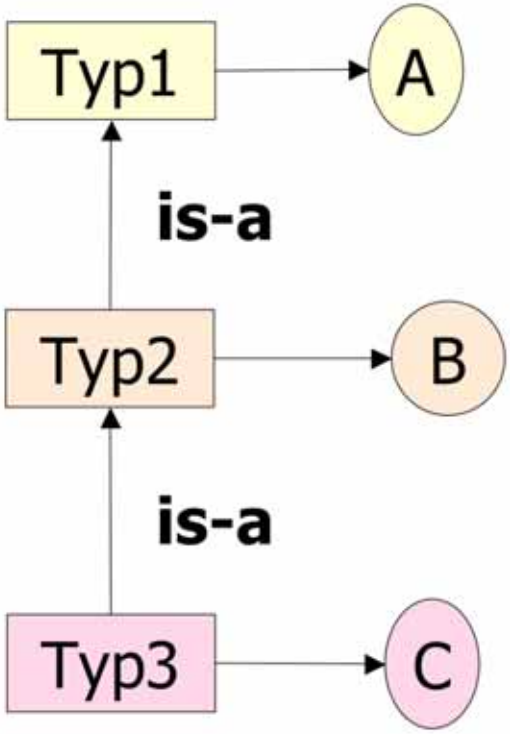
```
select p.wieHartAlsPrüfer()
from p in AlleProfessoren
where p.Name = „Curie“;
```

Vererbung und Subtypisierung (hier: ER)

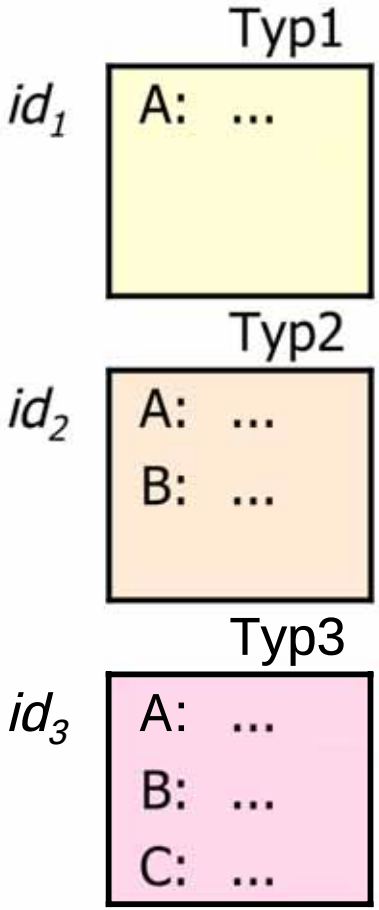


Terminologie

Objekttypen

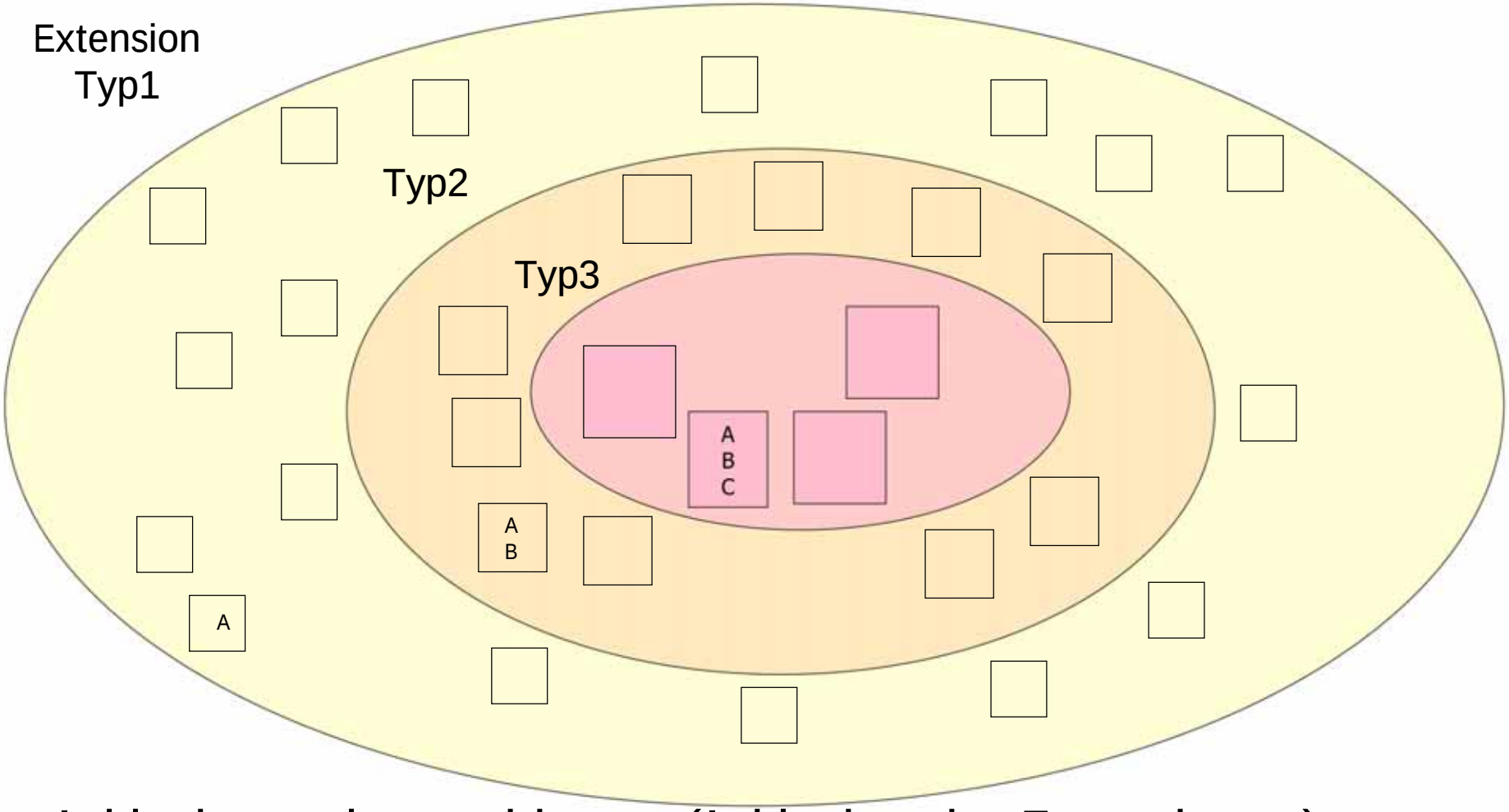


Instanzen



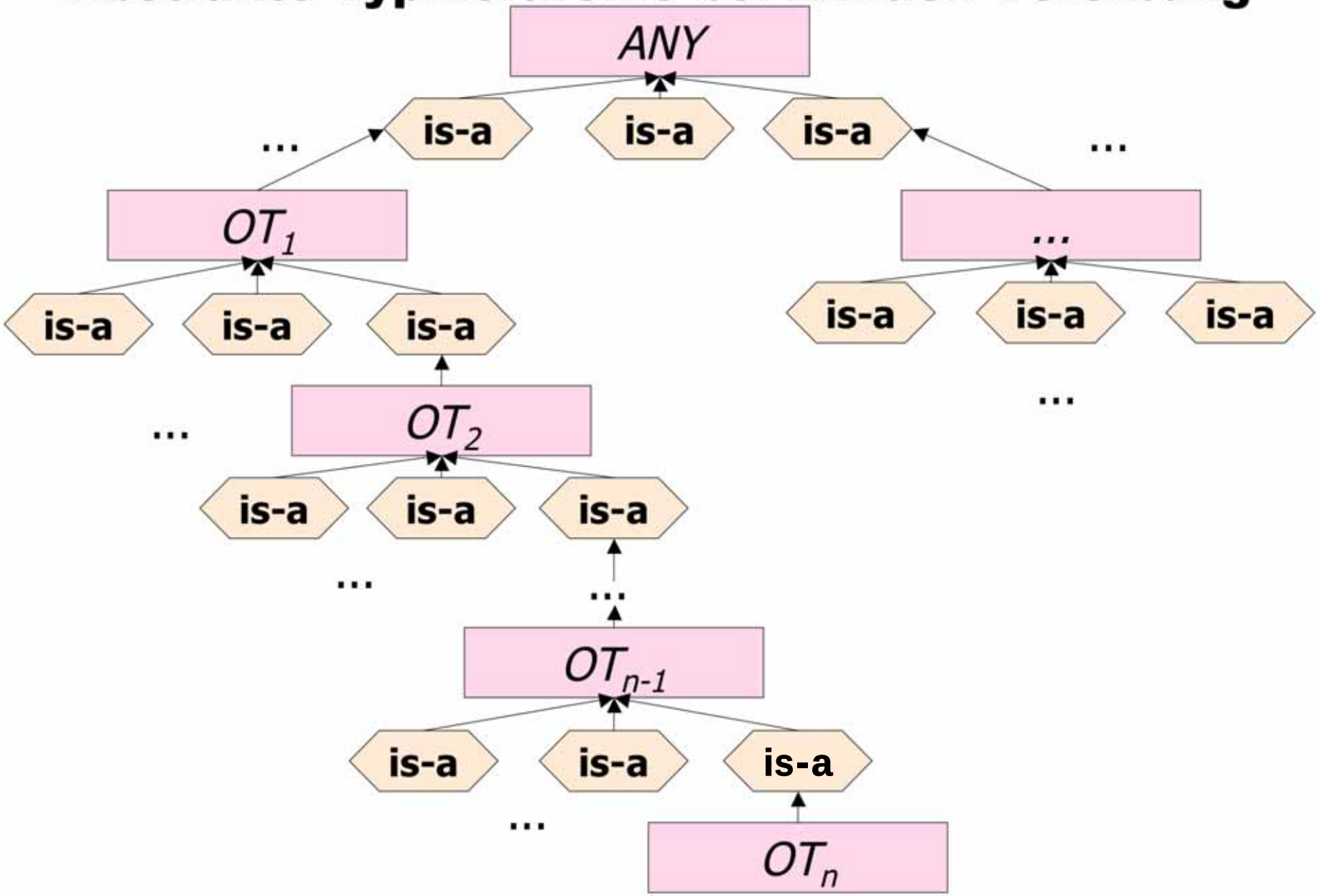
- Untertyp (Subtyp) / Obertyp (Supertyp)
- Instanz eines Untertyps gehört auch zur Extension des Obertyps
- Vererbung der Eigenschaften eines Obertyps an den Untertyp

Darstellung der Subtypisierung



- Inklusionspolymorphismus (Inklusion der Extensionen)
- Subtituierbarkeit
  - Eine Untertyp-Instanz ist überall dort einsetzbar, wo eine Obertyp-Instanz gefordert ist.

Abstrakte Typhierarchie bei Einfach-Vererbung

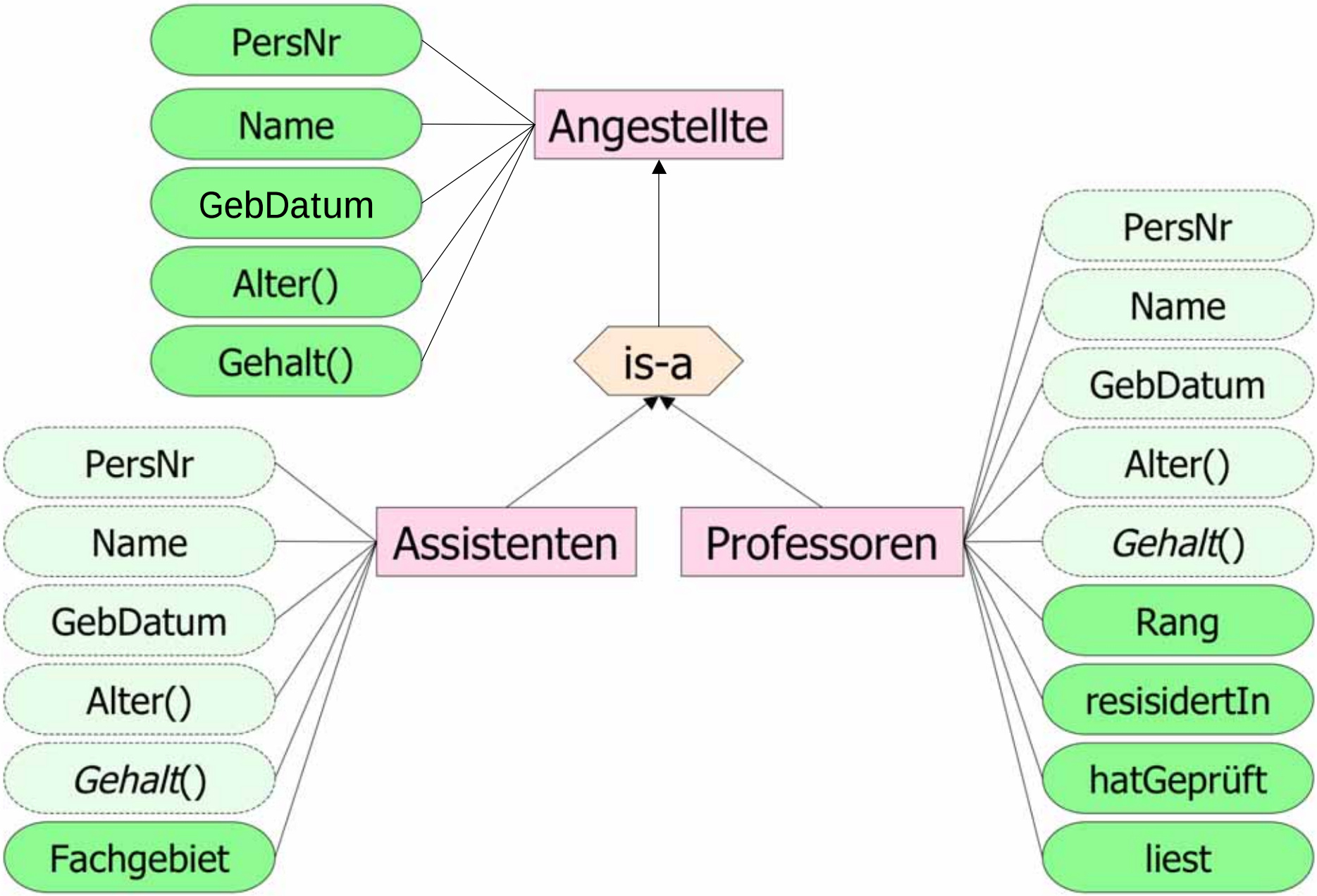


Eindeutiger Pfad: $OT_n \rightarrow OT_{n-1} \rightarrow ... \rightarrow OT_2 \rightarrow OT_1 \rightarrow ANY$

Einfachvererbung

- Objekttyp OT_n erbt alle Eigenschaften der Typen OT_{n-1}, ..., OT_1, *ANY*
- Pfad OT_n zu *ANY* eindeutig; damit sind die durch OT_n ererbten Eigenschaften eindeutig zu benennen
- Jeder Objekttyp ist auch Subtyp des (abstrakten) Obertyps *ANY*
  - *ANY* selbst trägt keine Eigenschaften (außer Objektidentität)
  - Im C++-Embedding: *ANY* ≡ d_Object

Vererbung von Eigenschaften



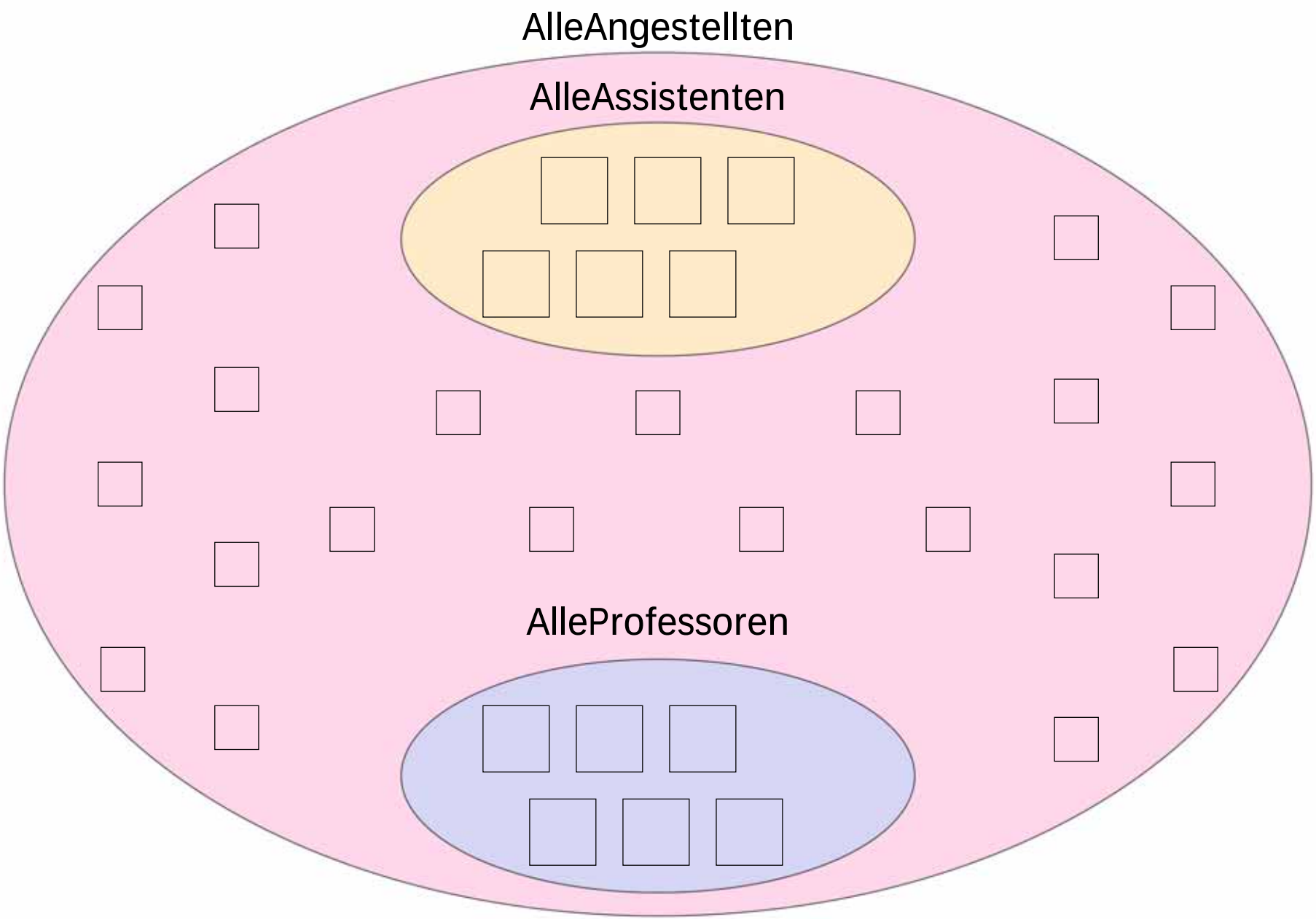
Interface-Definition in ODL

```
class Angestellte (extent AlleAngestellte) {
        attribute long PersNr;
        attribute string Name;
        attribute date GebDatum;
        short Alter();
        long Gehalt();
};

class Assistenten extends Angestellte (extent AlleAssistenten) {
        attribute string Fachgebiet;
};

class Professoren extends Angestellte (extent AlleProfessoren) {
        attribute string Rang;
        relationship Räume residiertIn inverse Räume::beherbergt;
        relationship set(Vorlesungen) liest inverse Vorlesungen::gelesenVon;
        relationship set(Prüfungen) hatGeprüft inverse Prüfungen::Prüfer;
};
```

Darstellung der Extensionen



Verfeinerung und spätes Binden

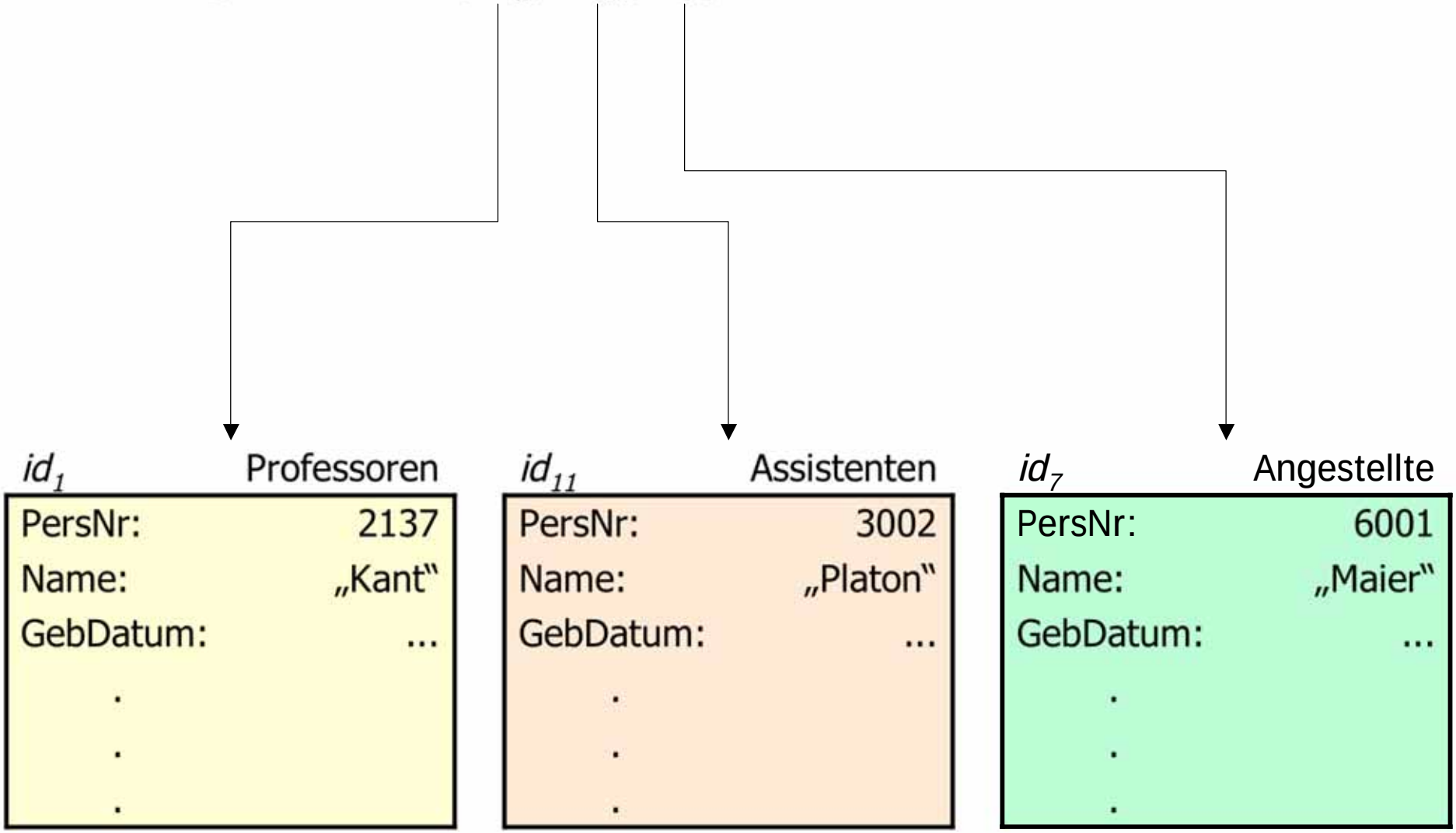
- Operationen werden (genau wie Attribute) von Obertypen an ihre Untertypen vererbt
  - Untertypen besitzen die Option, diese Operationen an ihre (speziellere) Situation anzupassen und damit zu **verfeinern**
  - Bezeichnet *o* ein Objekt des Objekttyps OT_n und *m* den Namen einer Methode, dann wird mittels

$$o.m(...);$$

    die erste Methode *m* aufgerufen, die auf dem Vererbungspfad von OT_n zu *ANY* gefunden wird.
  - Da im allg. der Typ von *o* statisch nicht genauer bestimmt werden kann als OT_i ($i \in \{1...n\}$), kann ein konkretes *m* erst zur Laufzeit bestimmt werden (***late binding***).

Verfeinerung und spätes Binden

- Die Extension *AlleAngestellten* mit (nur) drei Objekten
  AlleAngestellten: $\{id_1, id_{11}, id_7\}$



Verfeinerung (Spezialisierung) der Operation Gehalt()

- Angestellte erhalten: 2000 €+ (Alter() – 21) * 100 €
- Assistenten erhalten: 2500 €+ (Alter() – 21) * 125 €
- Professoren erhalten: 3000 € + (Alter() – 21) * 150 €

OQL:

```
select sum(a.Gehalt())
from a in AlleAngestellten
```

- für das Objekt id_1 wird die Professoren-spezifische Gehalt()-Methode durchgeführt,
- für das Objekt id_{11} die Assistenten-spezifische und
- für das Objekt id_7 die allgemeinste, also Angestellten-spezifische Realisierung der Operation Gehalt() gebunden.

Die Anfragesprache OQL

- Zentraler Bestandteil des ODMG-Standard ist die *Object Query Language* (**OQL**), die eine Brücke zwischen SQL und dem Objektmodell schlägt
  - Syntax und Semantik von OQL folgt dem **SQL-SFW-Block**
  - **Orthogonalität**: immer dort, wo ein Wert des Typs *T* erlaubt ist, kann auch eine OQL-(Unter-)Anfrage mit Ergebnistyp *T* eingesetzt werden, Schachtelung beliebig
  - Das Konzept der Tupelvariablen in SQL ist in OQL verallgemeinert worden. Eine OQL-Variable kann an
    1. **Werte** (auch mittels struct(…) konstruierte Strukturen)
    2. **Objekte**

    gebunden werden.
  - **Operator '.' (*dot*)** dient zum Attributzugriff, Verfolgen von Beziehungen und Aufruf von Methoden (*encapsulation*).

Die Anfragesprache OQL

Einfache Anfragen

- „Finde die Namen der C4-Professoren“

```
select p.Name
from p in AlleProfessoren
where p.Rang = „C4“;
```

```
select p
from p in AlleProfessoren
where p.Rang = „C4“;
```

- „Generiere Namen- und Rang-Tupel der C4-Professoren“

```
select struct(n: p.Name, r: p.Rang)
from p in AlleProfessoren
where p.Rang = „C4“;
```

Geschachtelte Anfragen und Strukturen, Aggregate

```
select struct(n: p.Name, a: sum(select v.SWS from v in p.liest))
from p in Alle Professoren
where avg(select v.SWS from v in p.liest) > 2;
```

Pfadausdrücke in OQL-Anfragen

```
select s.Name
from s in AlleStudenten, v in s.hört
where v.gelesenVon.Name = „Sokrates“;
```

- Visualisierung des Pfadausdruckes:

- Ein längerer Pfadausdruck:

eineVorlesung.gelesenVon.residiertIn.Größe

Vorlesungen

Professoren

Räume

float

Erzeugung von Objekten

- Erzeugung von Objekten: Aufruf des *OT*-Konstruktors:

```
Vorlesungen(VorlNr: 5555, Titel: "Ethik II", SWS: 4,
            gelesenVon: (select p
                         from p in AlleProfessoren
                         where p.Name = "Sokrates" ));
```

- Methodenaufruf innerhalb einer Anfrage:

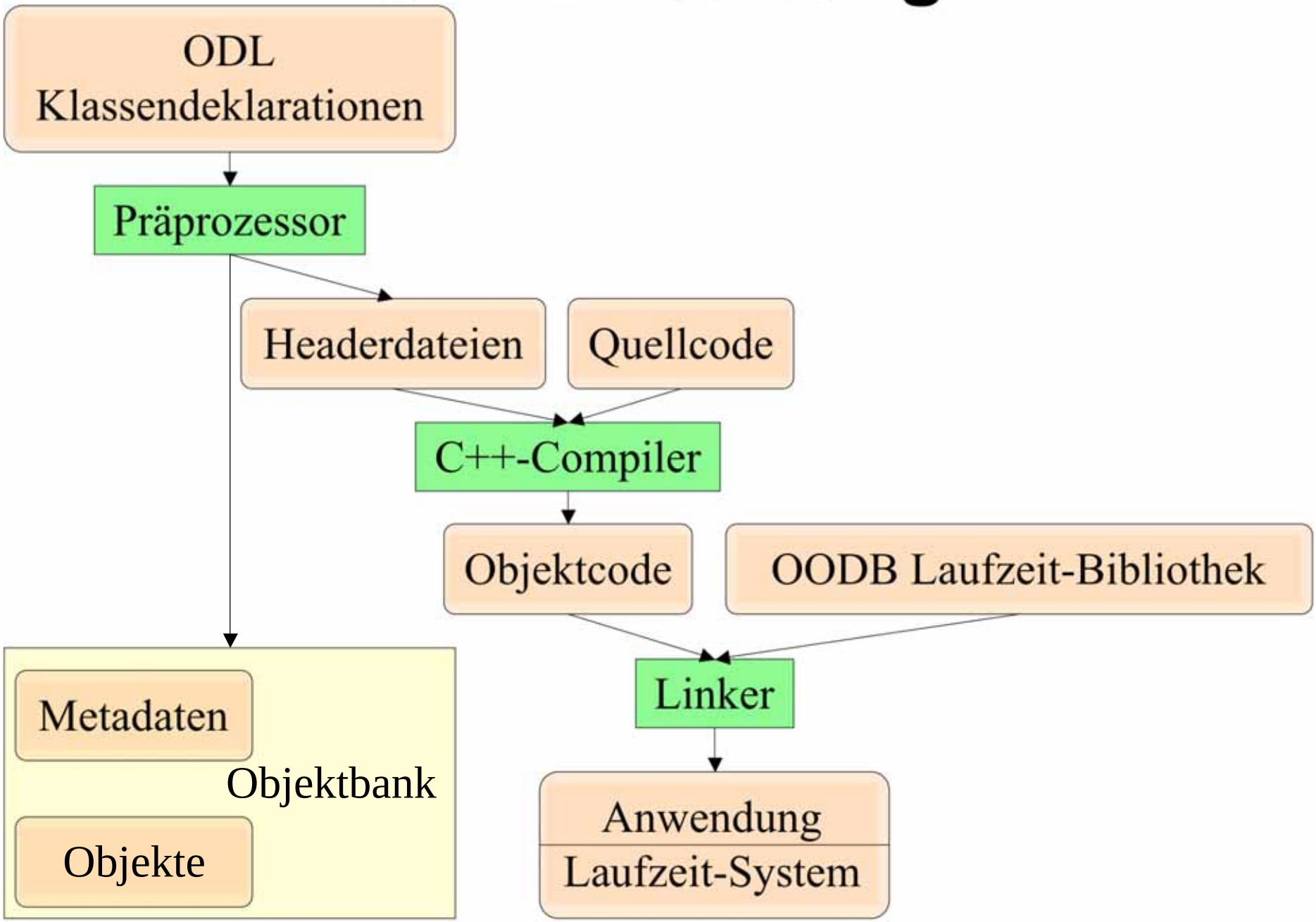
```
select a.Name
from a in AlleAngestellte
where a.Gehalt() > 100000;
```

Programmiersprachen-Anbindung

Entwurfsentscheidung

- Entwurf einer neuen Sprache
  - eleganteste Methode,
  - hoher Realisierungsaufwand
  - Benutzer müssen eine neue Programmiersprache lernen
- Erweiterung einer bestehenden Sprache
  - Benutzer müssen keine neue Sprache lernen
  - manchmal unnatürlich wirkende Erweiterungen der Basissprache
- Datenbankfähigkeit durch Typbibliothek
  - einfachste Möglichkeit für das Erreichen von Persistenz
  - mit den höchsten „Reibungsverlusten“
  - evtl. Probleme mit der Transparenz der Einbindung und der Typüberprüfung der Programmiersprache
  - ODMG-Ansatz

C++-Einbindung



C++-Einbindung

- Eine C++-Klasse deren Objekte persistent in der OODB gespeichert werden sollen, wird von der abstrakten Klasse d_Object abgeleitet:

```
class Vorlesung : public d_Object {   // C++-Code
        d_String Titel;
        d_Short  SWS;
        …
        d_Ref ⟨Professoren⟩ gelesenVon;
};
```

  - Typen d_String, d_Short, ... sind die C++-Varianten der ODL-Typen string, short, ...
  - d_Ref⟨*OT*⟩ implementiert eine Referenz auf ein persistentes Objekt des Objekttyps *OT* (¬ C++-Pointer!)

Instantiierung, Persistenz, Clustering

- Objekterzeugung (Instantiierung eines *OT*) wird in C++ mittels new() — eine Methode von d_Object — durchgeführt
  - Persistent wird ein so erzeugtes Objekt dann, falls dem new()-Operator eine **Platzierung** mitgegeben wird:
    1. Platzierung innerhalb einer Datenbank
    2. Platzierung "in der Nähe" eines existierenden Objektes (Clustering)

```
Ref ⟨Professoren⟩ Russel =
    new(UniDB) Professoren(2126, "Russel", "C4",...);

Ref ⟨Professoren⟩ Popper =
    new(Russel) Professoren(2133, "Popper", "C3",...);
```

Transaktionen

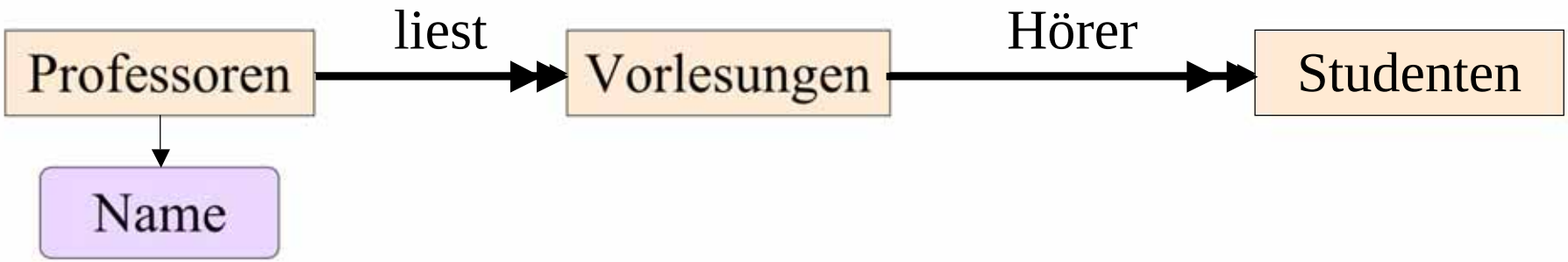
- Schachtelung von Transaktionen
- notwendig um Operationen, die TAs repräsentieren, geschachtelt aufrufen zu können.

```
void Professoren::Umziehen(Ref ⟨Räume⟩ neuerRaum) {
    Transaction TAumziehen;
    TAumziehen.start();
    ...
    if ( /*Fehler? */ )
        TAumziehen.abort();
    ...
    TAumziehen.commit();
};
```

C++: Einbettung von Anfragen

- Lose Kopplung: einige Typfehler sind erst zur Laufzeit des Programmes zu entdecken:

```
d_Bag ⟨Studenten⟩ Schüler;
char* profname = "…";
d_OQL_Query anfrage("select s
                     from p in AlleProfessoren, v in p.liest, s in v.Hörer
                     where p.Name = $1");
anfrage << profname;                    // Parameter $1 binden
d_oql_execute(anfrage, Schüler);        // Anfrage ausführen
```



Objektrelationale Datenbanken

- Mengenwertige Attribute
- Typdeklarationen
- Referenzen
- Objektidentität
- Pfadausdrücke
- Vererbung
- Operationen