

## Kapitel 16

# Objektorientierte Datenbanken

Relationale Datenbanksysteme sind derzeit in administrativen Anwendungsbereichen marktbeherrschend, da sich die sehr einfache Strukturierung der Daten in flachen Tabellen als recht benutzerfreundlich erwiesen hat. Unzulänglichkeiten traten jedoch zutage bei komplexeren, ingenieurwissenschaftlichen Anwendungen, z.B. in den Bereichen CAD, Architektur und Multimedia.

Daraus ergaben sich zwei unterschiedliche Ansätze der Weiterentwicklung:

- Der *evolutionäre* Ansatz: Das relationale Modell wird um komplexe Typen erweitert zum sogenannten *geschachtelten* relationalen Modell.
- Der *revolutionäre* Ansatz: In Analogie zur Objektorientierten Programmierung wird in einem Objekttyp die *strukturelle* Information zusammen mit der *verhaltensmäßigen* Information integriert.

### 16.1 Schwächen relationaler Systeme

Die Relation

Buch : {[ ISBN, Verlag, Titel, Autor, Version, Stichwort ]}

erfordert bei 2 Autoren, 5 Versionen, 6 Stichworten für jedes Buch  $2 \times 5 \times 6 = 60$  Einträge.

Eine Aufsplittung auf mehrere Tabellen ergibt

Buch : {[ ISBN, Titel, Verlag ]}  
Autor : {[ ISBN, Name, Vorname ]}  
Version : {[ ISBN, Auflage, Jahr ]}  
Stichwort : {[ ISBN, Stichwort ]}

Nun sind die Informationen zu einem Buch auf vier Tabellen verteilt. Beim Einfügen eines neuen Buches muß mehrmals dieselbe ISBN eingegeben werden. Die referentielle Integrität muß selbst überwacht werden. Eine Query der Form “*Liste Bücher mit den Autoren Meier und Schmidt*“ ist nur sehr umständlich zu formulieren.

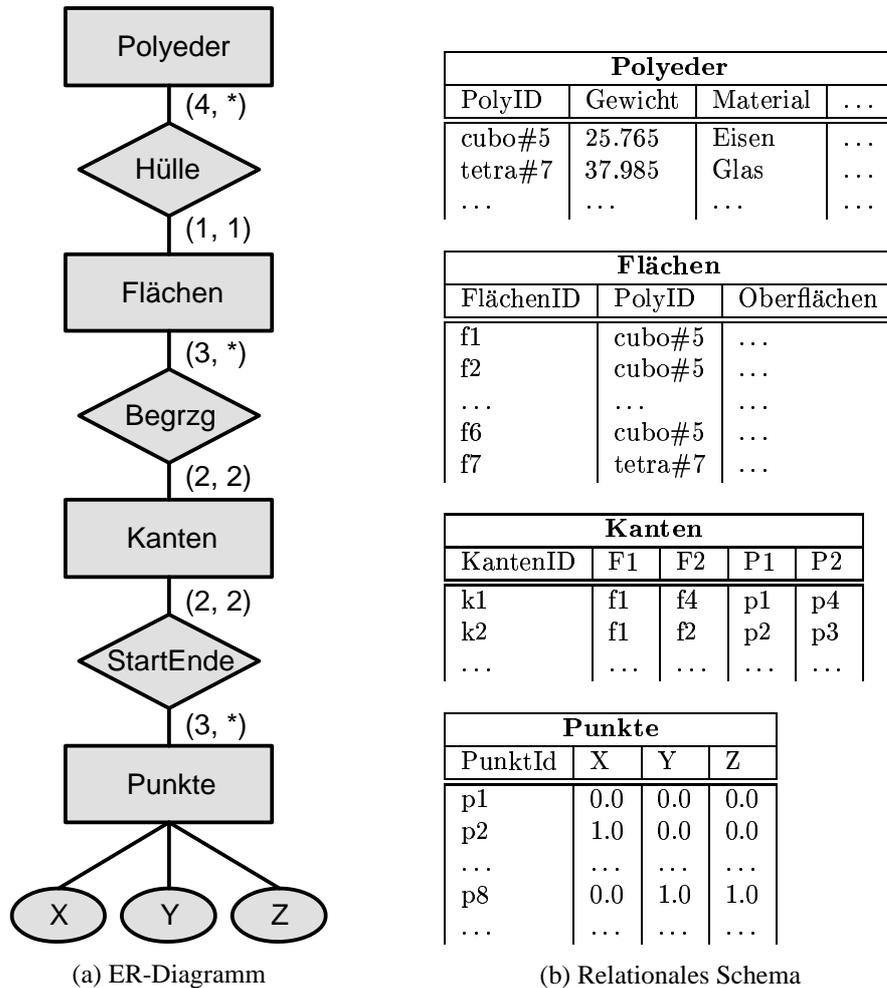


Abbildung 16.1: Modellierung von Polyedern

Abbildung 16.1a zeigt die Modellierung von Polyedern nach dem Begrenzungsflächenmodell, d. h. ein Polyeder wird beschrieben durch seine begrenzenden Flächen, diese wiederum durch ihre beteiligten Kanten und diese wiederum durch ihre beiden Eckpunkte. Abbildung 16.1b zeigt eine mögliche Umsetzung in ein relationales Schema, wobei die Beziehungen *Hülle*, *Begrzg* und *StartEnde* aufgrund der Kardinalitäten in die Entity-Typen integriert wurden.

Die relationale Modellierung hat etliche Schwachpunkte:

- **Segmentierung:** Ein Anwendungsobjekt wird über mehrere Relationen verteilt, die immer wieder durch einen Verbund zusammengefügt werden müssen.
- **Künstliche Schlüsselattribute:** Zur Identifikation von Tupeln müssen vom Benutzer relationenweit eindeutige Schlüssel vergeben werden.

- **Fehlendes Verhalten:** Das anwendungsspezifische Verhalten von Objekten, z.B. die Rotation eines Polyeders, findet im relationalen Schema keine Berücksichtigung.
- **Externe Programmierschnittstelle:** Die Manipulation von Objekten erfordert eine Programmierschnittstelle in Form einer Einbettung der (mengenorientierten) Datenbanksprache in eine (satzorientierte) Programmiersprache.

## 16.2 Vorteile der objektorientierten Modellierung

In einem objektorientierten Datenbanksystem werden *Verhaltens-* und *Struktur-*Beschreibungen in einem Objekt-Typ integriert. Das anwendungsspezifische Verhalten wird integraler Bestandteil der Datenbank. Dadurch können die umständlichen Transformationen zwischen Datenbank und Programmiersprache vermieden werden. Vielmehr sind die den Objekten zugeordneten Operationen direkt ausführbar, ohne detaillierte Kenntnis der strukturellen Repräsentation der Objekte. Dies wird durch das *Geheimnisprinzip* (engl.: *information hiding*) unterstützt, wonach an der Schnittstelle des Objekttyps eine Kollektion von Operatoren angeboten wird, für deren Ausführung man lediglich die *Signatur* (Aufrufstruktur) kennen muß.

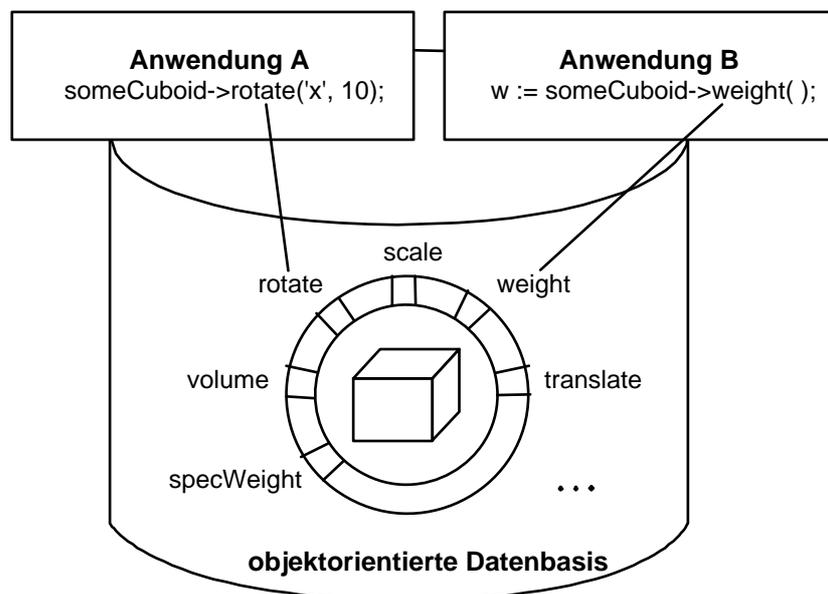


Abbildung 16.2: Visualisierung der Vorteile der objektorientierten Datenmodellierung

Abbildung 16.2 visualisiert den objektorientierten Ansatz bei der Datenmodellierung. Ein Quader wird zusammen mit einer Reihe von Datenfeldern und Operatoren zur Verfügung gestellt. Unter Verwendung dieser Schnittstelle rotiert Anwendung A einen Quader und bestimmt Anwendung B das Gewicht.

## 16.3 Der ODMG-Standard

Im Gegensatz zum relationalen Modell ist die Standardisierung bei objektorientierten Datenbanksystemen noch nicht so weit fortgeschritten. Ein (de-facto) Standard wurde von der *Object Database Management Group* entworfen. Das ODMG-Modell umfaßt das objektorientierte Datenbanksystem und eine einheitliche Anbindung an bestehende Programmiersprachen. Bisher wurden Schnittstellen für C++ und Smalltalk vorgesehen. Außerdem wurde eine an SQL angelehnte deklarative Abfragesprache namens *OQL* (Object Query Language) entworfen.

## 16.4 Eigenschaften von Objekten

Im relationalen Modell werden Entitäten durch Tupel dargestellt, die aus atomaren *Literalen* bestehen.

Im objektorientierten Modell hat ein Objekt drei Bestandteile:

- **Identität:** Jedes Objekt hat eine systemweit eindeutige Objektidentität, die sich während seiner Lebenszeit nicht verändert.
- **Typ:** Der Objekttyp, auch *Klasse* genannt, legt die Struktur und das Verhalten des Objekts fest. Individuelle Objekte werden durch die *Instanzierung* eines Objekttyps erzeugt und heißen *Instanzen*. Die Menge aller Objekte (Instanzen) eines Typs wird als (Typ-) *Extension* (eng. *extent*) bezeichnet.
- **Wert bzw. Zustand:** Ein Objekt hat zu jedem Zeitpunkt seiner Lebenszeit einen bestimmten Zustand, auch Wert genannt, der sich aus der momentanen Ausprägung seiner Attribute ergibt.

Abbildung 16.3 zeigt einige Objekte aus der Universitätswelt. Dabei wird zum Beispiel der Identifikator  $id_1$  als Wert des Attributs *gelesenVon* in der Vorlesung mit dem Titel *Grundzüge* verwendet, um auf die Person mit dem Namen *Kant* zu verweisen. Wertebereiche bestehen nicht nur aus atomaren Literalen, sondern auch aus Mengen. Zum Beispiel liest *Kant* zwei Vorlesungen, identifiziert durch  $id_2$  und  $id_3$ .

Im relationalen Modell wurden Tupel anhand der Werte der Schlüsselattribute identifiziert (*identity through content*). Dieser Ansatz hat verschiedene Nachteile:

- Objekte mit gleichem Wert müssen nicht unbedingt identisch sein. Zum Beispiel könnte es zwei Studenten mit Namen "Willy Wacker" im 3. Semester geben.
- Aus diesem Grund müssen künstliche Schlüsselattribute ohne Anwendungsemantik (siehe Polyedermodellierung) eingeführt werden.
- Schlüssel dürfen während der Lebenszeit eines Objekts nicht verändert werden, da ansonsten alle Bezugnahmen auf das Objekt ungültig werden.

In Programmiersprachen wie Pascal oder C verwendet man Zeiger, um Objekte zu referenzieren. Dies ist für kurzlebige (transiente) Hauptspeicherobjekte akzeptabel, allerdings nicht für persistente Objekte.

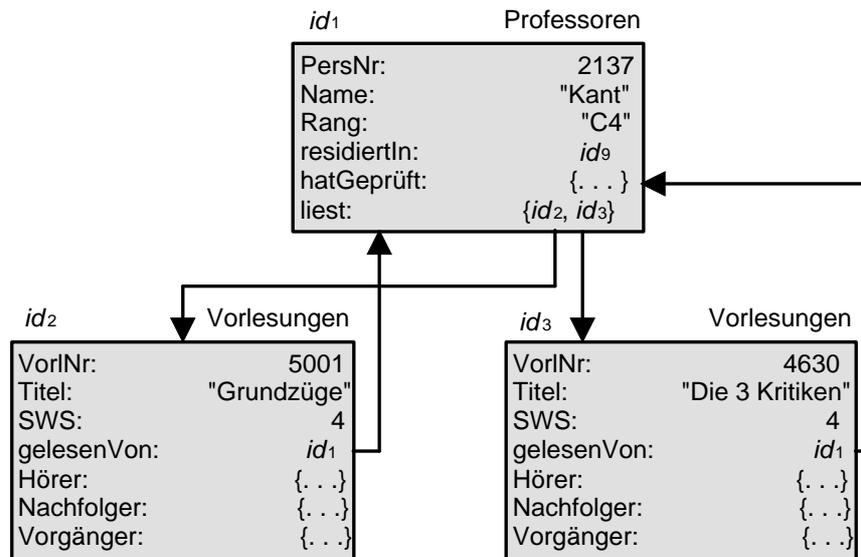


Abbildung 16.3: Einige Objekte aus der Universitätswelt

Objektorientierte Datenbanksysteme verwenden daher zustands- und speicherort-unabhängige *Objektidentifikatoren* (OIDs). Ein OID wird vom Datenbanksystem systemweit eindeutig generiert, sobald ein neues Objekt erzeugt wird. Der OID bleibt dem Anwender verborgen, er ist unveränderlich und unabhängig vom momentanen Objekt-Zustand. Die momentane physikalische Adresse ergibt sich aus dem Inhalt einer Tabelle, die mit dem OID referiert wird.

Die Objekttyp-Definition enthält folgende Bestandteile:

- die Strukturbeschreibung der Instanzen, bestehend aus Attributen und Beziehungen zu anderen Objekten,
- die Verhaltensbeschreibung der Instanzen, bestehend aus einer Menge von Operationen,
- die Typeigenschaften, z.B. Generalisierungs- und Spezialisierungsbeziehungen.

## 16.5 Definition von Attributen

Die Definition des Objekttyps *Professoren* könnte wie folgt aussehen:

```

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

Attribute können strukturiert werden mit Hilfe des Tupelkonstruktors `struct{...}`:

```
class Person {
  attribute string Name;
  attribute struct Datum {
    short Tag;
    short Monat;
    short Jahr;
  } GebDatum;
};
```

## 16.6 Definition von Beziehungen

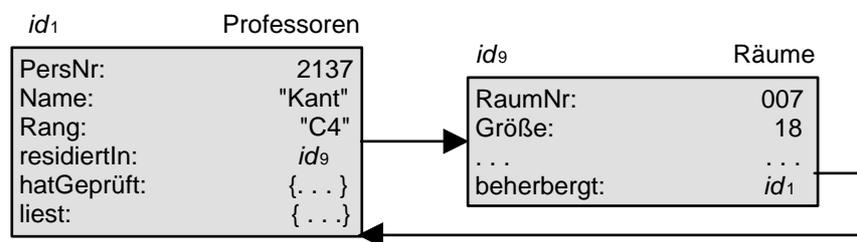


Abbildung 16.4: Ausprägung einer 1:1-Beziehung

Eine 1 : 1-Beziehung wird symmetrisch in beiden beteiligten Objekt-Typen modelliert:

```
class Professoren {
  attribute long PersNr;
  ...
  relationship Raeume residiertIn;
};

class Raeume {
  attribute long RaumNr;
  attribute short Groesse;
  ...
  relationship Professoren beherbergt;
};
```

Abbildung 16.4 zeigt eine mögliche Ausprägung der Beziehungen *residiertIn* und *beherbergt*.

Allerdings wird durch die gegebene Klassenspezifikation weder die Symmetrie noch die 1:1-Einschränkung garantiert. Abbildung 16.5 zeigt einen inkonsistenten Zustand des Beziehungspaares *residiertIn* und *beherbergt*.

Um Inkonsistenzen dieser Art zu vermeiden, wurde im ODMG-Objektmodell das **inverse**-Konstrukt integriert:

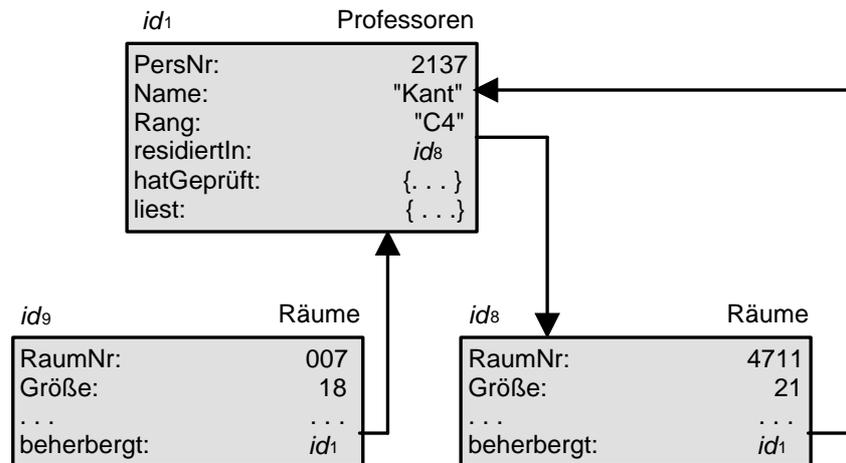


Abbildung 16.5: Inkonsistenter Zustand einer Beziehung

```

class Professoren {
    attribute long  PersNr;
    ...
    relationship Raeume residiertIn inverse Raeume::beherbergt;
};

class Raeume {
    attribute long  RaumNr;
    attribute short Groesse;
    ...
    relationship Professoren beherbergt inverse Professoren::residiertIn;
};

```

Damit wird sichergestellt, daß immer gilt:

$$p = r.beherbergt \Leftrightarrow r = p.residiertIn$$

Binäre  $1:N$  - Beziehungen werden modelliert mit Hilfe des Mengenkonstruktors **set**, der im nächsten Beispiel einem Professor eine Menge von Referenzen auf *Vorlesungen*-Objekte zuordnet:

```

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

class Vorlesungen {
    ...
    relationship Professoren gelesenVon inverse Professoren::liest;
};

```

Man beachte, daß im relationalen Modell die Einführung eines Attributs *liest* im Entity-Typ *Professoren* die Verletzung der 3. Normalform verursacht hätte.

Binäre  $N:M$  - Beziehungen werden unter Verwendung von zwei **set**-Konstruktoren modelliert:

```
class Studenten {
    ...
    relationship set (Vorlesungen) hoert inverse Vorlesungen::Hoerer;
};

class Vorlesungen {
    ...
    relationship set (Studenten) Hoerer inverse Studenten::hoert;
};
```

Durch die **inverse**-Spezifikation wird sichergestellt, daß gilt:

$$s \in v.Hoerer \Leftrightarrow v \in s.hoert$$

Analog lassen sich rekursive  $N : M$  - Beziehungen beschreiben:

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

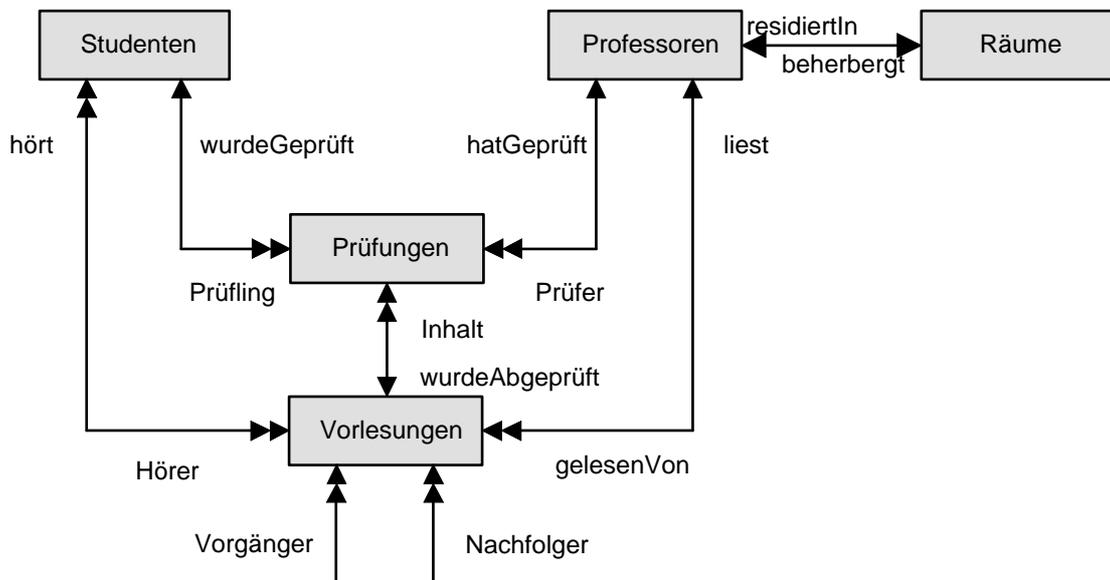


Abbildung 16.6: Modellierungen von Beziehungen im Objektmodell

Ternäre oder  $n \geq 3$  stellige Beziehungen benötigen einen eigenständigen Objekttyp, der die Beziehung repräsentiert. Zum Beispiel wird die ternäre Beziehung

pruefen : {[ MatrNr, VorlNr, PersNr, Note ]}

zwischen den *Studenten*, *Vorlesungen* und *Professoren* wie folgt modelliert:

```

class Pruefungen {
  attribute float      Note;
  relationship Professoren Pruefer  inverse Professoren::hatgeprueft;
  relationship Studenten  Pruefling inverse Studenten::wurdegeprueft;
  relationship Vorlesungen Inhalt   inverse Vorlesungen::wurdeAbgeprueft;
};

class Professoren {
  attribute long      PersNr;
  attribute string   Name;
  attribute string   Rang;
  relationship Raeume  residiertIn inverse Raeume::beherbergt;
  relationship set<Vorlesungen> liest      inverse Vorlesungen::gelesenVon;
  relationship set<Pruefungen> hatgeprueft inverse Pruefungen::Pruefer;
};

class Vorlesungen {
  attribute long      VorlNr;
  attribute string   Titel;
  attribute short    SWS;
  relationship Professoren  gelesenVon inverse Professoren::liest;
  relationship set<Studenten> Hoerer      inverse Studenten::hoert;
  relationship set<Vorlesungen> Nachfolger inverse Vorlesungen::Vorgaenger;
  relationship set<Vorlesungen> Vorgaenger inverse Vorlesungen::Nachfolger;
  relationship set<Pruefungen> wurdeAbgeprueft inverse Pruefungen::Inhalt;
};

class Studenten {
  attribute long      MatrNr;
  attribute string   Name;
  attribute short    Semester;
  relationship set<Pruefungen> wurdeGeprueft inverse Pruefungen::Pruefling;
  relationship set<Vorlesungen> hoert      inverse Vorlesungen::Hoerer;
};

```

Abbildung 16.6 visualisiert die bislang eingeführten Beziehungen. Die Anzahl der Pfeilspitzen gibt die Wertigkeit der Beziehung an:

<p>↔ bezeichnet eine 1 : 1-Beziehung  ↔↔ bezeichnet eine N : 1-Beziehung</p>	<p>↔↔↔ bezeichnet eine 1 : N-Beziehung  ↔↔↔↔ bezeichnet eine N : M-Beziehung</p>
--	--

## 16.7 Extensionen und Schlüssel

Eine *Extension* ist die Menge aller Instanzen eines Objekt-Typs incl. seiner spezialisierten Untertypen (siehe später). Sie kann verwendet werden für Anfragen der Art “*Suche alle Objekte eines Typs, die eine bestimmte Bedingung erfüllen*“. Man kann zu einem Objekttyp auch *Schlüssel* definieren, deren Eindeutigkeit innerhalb der Extension gewährleistet wird. Diese Schlüsselinformation wird jedoch nur als Integritätsbedingung verwendet und nicht zur Referenzierung von Objekten:

```
class Studenten (extent AlleStudenten key MatrNr) {
  attribute    long           MatrNr;
  attribute    string        Name;
  attribute    short         Semester;
  relationship set(Vorlesungen) hoert           inverse Vorlesungen::Hoerer;
  relationship set(Pruefungen) wurdeGeprueft   inverse Pruefungen::Pruefiling;
};
```

## 16.8 Modellierung des Verhaltens

Der Zugriff auf den Objektzustand und die Manipulation des Zustands geschieht über eine *Schnittstelle*. Die Schnittstellenoperationen können

- ein Objekt erzeugen (instanzieren) und initialisieren mit Hilfe eines *Konstruktors*,
- freigegebene Teile des Zustands erfragen mit Hilfe eines *Observers*,
- konsistenzhaltende Operationen auf einem Objekt ausführen mit Hilfe eines *Mutators*,
- das Objekt zerstören mit Hilfe eines *Destruktors*.

Die Aufrufstruktur der Operation, genannt *Signatur*, legt folgendes fest:

- Name der Operation,
- Anzahl und Typ der Parameter,
- Typ des Rückgabewerts, falls vorhanden, sonst **void**,
- ggf. die durch die Operation ausgelöste *Ausnahme* (engl. *exception*).

Beispiel:

```
class Professoren {
  exception hatNochNichtGeprueft { };
  exception schonHoechsteStufe   { };
  ...
  float wieHartAlsPruefer() raises (hatNochNichtgeprueft);
  void befoerdert() raises (schonHoechsteStufe);
};
```

Hierdurch wird der Objekttyp *Professoren* um zwei Signaturen erweitert:

- Der Observer *wieHartalsPruefer* liefert die Durchschnittsnote und stößt die Ausnahmebehandlung *hatNochNichtGepueft* an, wenn keine Prüfungen vorliegen.
- Der Mutator *befoerdert* erhöht den Rang um eine Stufe und stößt die Ausnahmebehandlung *schonHoechsteStufe* an, wenn bereits die Gehaltsstufe C4 vorliegt.

Man bezeichnet den Objekttyp, auf dem die Operationen definiert wurden, als *Empfängertyp* (engl. *receiver type*) und das Objekt, auf dem die Operation aufgerufen wird, als *Empfängerobjekt*.

Die Aufrufstruktur hängt von der Sprachanbindung ab. Innerhalb von C++ würde befördert aufgerufen als

```
meinLieblingsProf->befoerdert();
```

In der deklarativen Anfragesprache OQL (siehe Abschnitt 16.13) ist der Aufruf wahlweise mit Pfeil (->) oder mit einem Punkt (.) durchzuführen:

```
select p.wieHartAlsPruefer()
from p in AlleProfessoren
where p.Name = "Kant";
```

## 16.9 Vererbung

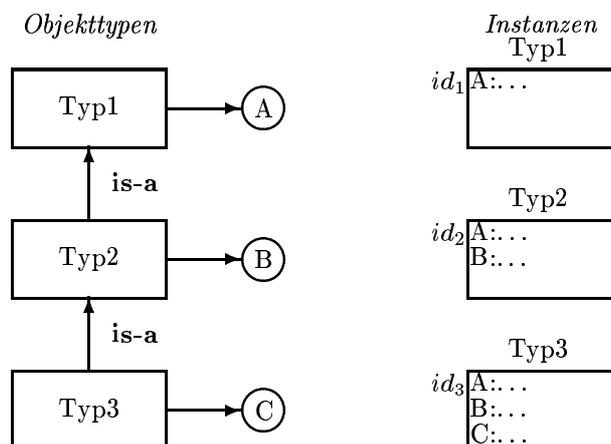


Abbildung 16.7: Schematische Darstellung einer abstrakten Typhierarchie

Das in Kapitel 2 eingeführte Konzept der Generalisierung bzw. Spezialisierung läßt sich bei objektorientierten Datenbanksystemen mit Hilfe der *Vererbung* lösen. Hierbei erbt der Untertyp nicht nur die Struktur, sondern auch das Verhalten des Obertyps. Außerdem sind Instanzen des Untertyps überall dort einsetzbar (*substituierbar*), wo Instanzen des Obertyps erforderlich sind.

Abbildung 16.7 zeigt eine Typhierarchie, bei der die Untertypen *Typ2* und *Typ3* jeweils ein weiteres Attribut, nämlich *B* bzw. *C* aufweisen. Operationen sind hier gänzlich außer Acht gelassen. Instanzen des Typs *Typ3* gehören auch zur Extension von Typ *Typ2* und von Typ *Typ1*.

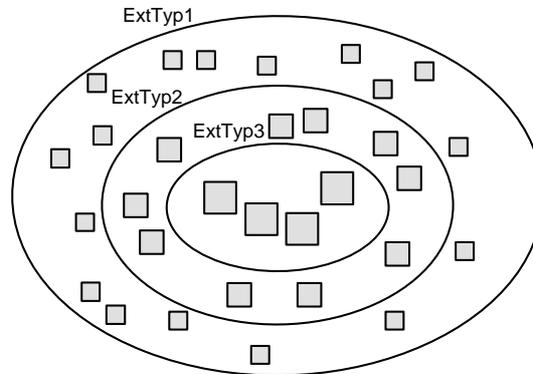


Abbildung 16.8: Darstellung der Subtypisierung

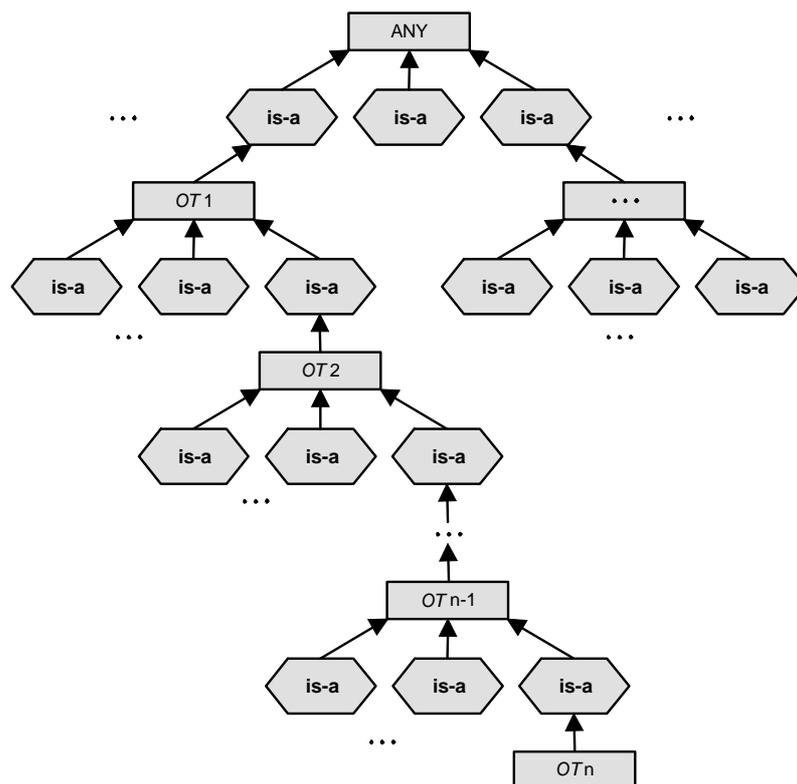


Abbildung 16.9: Abstrakte Typhierarchie bei einfacher Vererbung

Abbildung 16.8 zeigt die geschachtelte Anordnung der drei Extensionen der Typen *Typ1*, *Typ2* und *Typ3*. Durch die unterschiedliche Kästchengröße soll angedeutet werden, daß Untertyp-Instanzen mehr Eigenschaften haben und daher mehr wissen als die direkten Instanzen eines Obertyps.

Man unterscheidet zwei unterschiedliche Arten der Vererbung:

- *Einfachvererbung (single inheritance)*:  
Jeder Objekttyp hat höchstens einen direkten Obertyp.
- *Mehrfachvererbung (multiple inheritance)*:  
Jeder Objekttyp kann mehrere direkte Obertypen haben.

Abbildung 16.9 zeigt eine abstrakte Typhierarchie mit Einfachvererbung. Der Vorteil der Einfachvererbung gegenüber der Mehrfachvererbung besteht darin, daß es für jeden Typ einen eindeutigen Pfad zur Wurzel der Typhierarchie (hier: ANY) gibt. Ein derartiger Super-Obertyp findet sich in vielen Objektmodellen, manchmal wird er *object* genannt, in der ODMG C++-Einbindung heißt er *d\_Object*.

## 16.10 Beispiel einer Typhierarchie

Wir betrachten eine Typhierarchie aus dem Universitätsbereich. *Angestellte* werden spezialisiert zu *Professoren* und *Assistenten*:

```
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 Raeume          residiertIn inverse Raeume::beherbergt;
    relationship set(Vorlesungen) liest      inverse Vorlesungen::gelesenVon;
    relationship set(Pruefungen) hatgeprueft inverse Pruefungen::Pruefer;
};
```

Abbildung 16.10 zeigt die drei Objekttypen *Angestellte*, *Professoren* und *Assistenten*, wobei die ererbten Eigenschaften in den gepunkteten Ovalen angegeben ist.

Abbildung 16.11 zeigt schematisch die aus der Ober-/Untertyp-Beziehung resultierende Inklusion der Extensionen *AlleProfessoren* und *AlleAssistenten* in der Extension *AlleAngestellte*.

## 16.11 Verfeinerung und spätes Binden

Genau wie Attribute werden auch Operationen vom Obertyp an alle Untertypen vererbt. Zum Beispiel steht der bei *Angestellte* definierte Observer *Gehalt()* auch bei den Objekttypen *Professoren* und

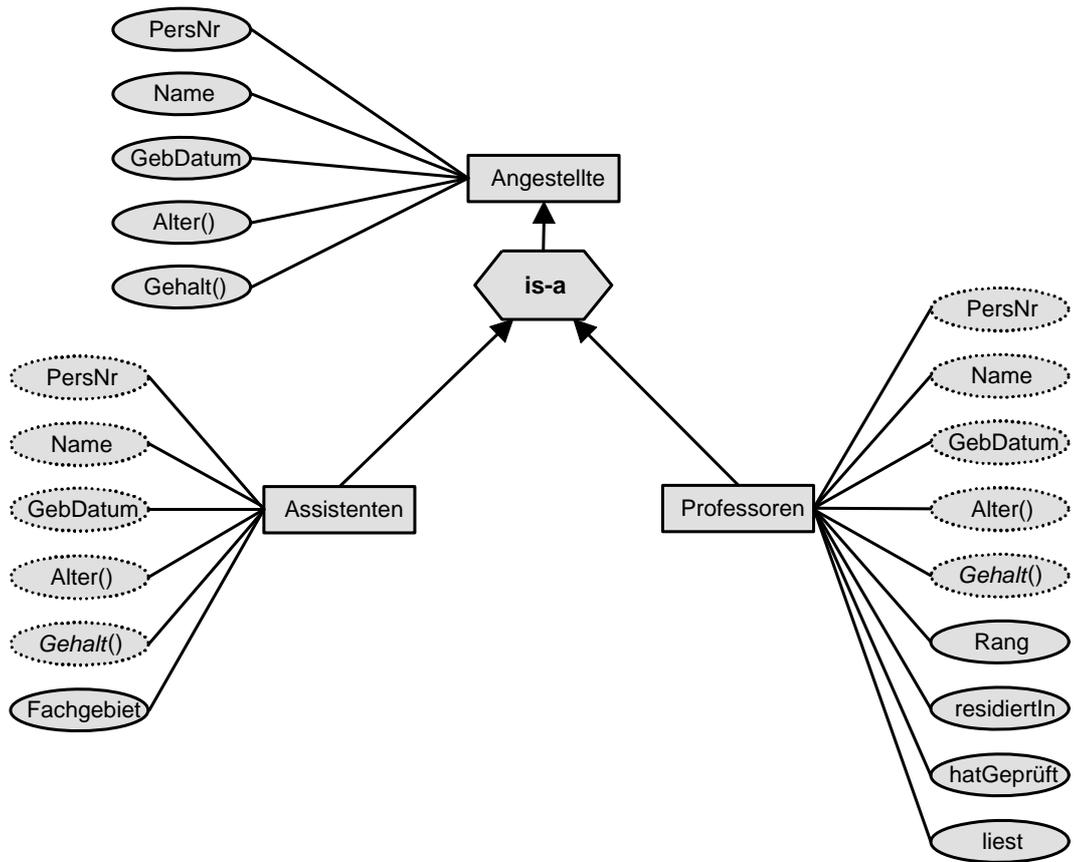


Abbildung 16.10: Vererbung von Eigenschaften

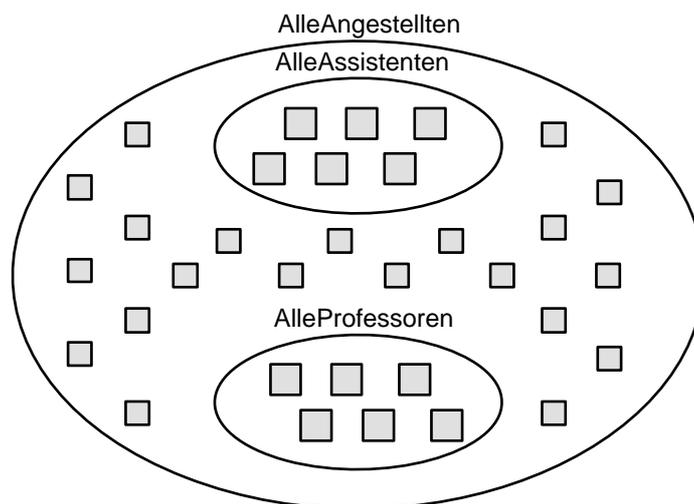


Abbildung 16.11: Visualisierung der Extensionen

Assistenten zur Verfügung.

Wir hätten auch eine *Verfeinerung* bzw. *Spezialisierung* (engl. *Refinement*) vornehmen können. Das Gehalt würde danach für jeden Objekttyp unterschiedlich berechnet:

- Angestellte erhalten  $2000 + (\text{Alter}() - 21) * 100$  Euro,
- Assistenten erhalten  $2500 + (\text{Alter}() - 21) * 125$  Euro,
- Professoren erhalten  $3000 + (\text{Alter}() - 21) * 150$  Euro.

In Abbildung 16.10 ist dies durch den kursiven Schrifttyp der geerbten *Gehalt*-Eigenschaft gekennzeichnet.

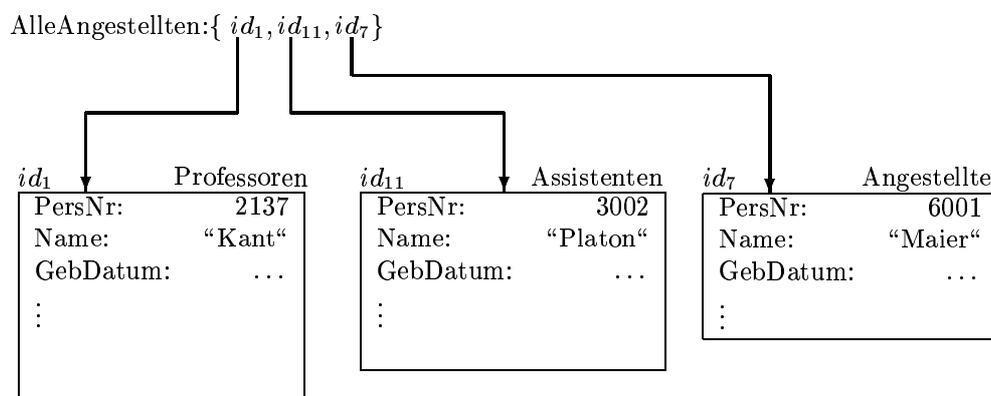


Abbildung 16.12: Die Extension *AlleAngestellten* mit drei Objekten

Abbildung 16.12 zeigt die Extension *AlleAngestellten* mit drei Elementen:

- Objekt *id<sub>1</sub>* ist eine direkte *Professoren*-Instanz,
- Objekt *id<sub>11</sub>* ist eine direkte *Assistenten*-Instanz,
- Objekt *id<sub>7</sub>* ist eine direkte *Angestellte*-Instanz.

Es werde nun die folgende Query abgesetzt:

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

Offensichtlich kann erst zur Laufzeit die jeweils spezialisierteste Version von *Gehalt* ermittelt werden. Diesen Vorgang bezeichnet man als *spätes Binden* (engl. *late binding*).

## 16.12 Mehrfachvererbung

Bei der Mehrfachvererbung erbt ein Objekttyp die Eigenschaften von mehreren Obertypen. Abbildung 16.13 zeigt ein Beispiel dafür. Der Objekttyp *Hiwi* erbt

- von *Angestellte* die Attribute *PersNr*, *Name* und *GebDatum* sowie die Operationen *Gehalt()* und *Alter()*,
- von *Studenten* die Attribute *MatrNr*, *Name*, *Semester*, *hört* und *wurdeGeprüft*.

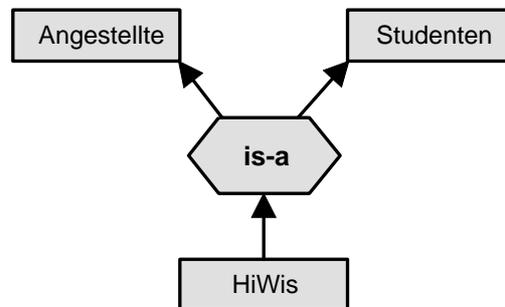


Abbildung 16.13: Beispiel für Mehrfachvererbung

Die Syntax könnte lauten:

```
class HiWis extends Studenten, Angestellte (extent AlleHiWis) {
    attribute short Arbeitsstunden;
    ...
};
```

Nun wird allerdings das Attribut *Name* sowohl von *Angestellte* als auch von *Studenten* geerbt. Um solchen Mehrdeutigkeiten und den damit verbundenen Implementationsproblemen aus dem Weg zu gehen, wurde in der Version 2.0 von ODL das Schnittstellen-Konzept (engl *interface*) eingeführt, das es in ähnlicher Form auch in der Programmiersprache *Java* gibt.

Eine **interface**-Definition ist eine abstrakte Definition der Methoden, die alle Klassen besitzen müssen, die diese Schnittstelle implementieren. Eine Klasse im ODBG-Modell kann mehrere Schnittstellen implementieren, darf aber nur höchstens von einer Klasse mit **extends** abgeleitet werden

Also würde man für die Angestellten lediglich die Schnittstelle *AngestellteIF* festlegen. Die Klasse *HiWis* implementiert diese Schnittstelle und erbt den Zustand und die Methoden der Klasse *Studenten*. Die Liste der Schnittstellen, die eine Klasse implementiert, wird in der Klassendefinition nach dem Klassennamen und der möglichen **extends**-Anweisung hinter einem Doppelpunkt angegeben. Zusätzlich muß der nicht mitgeerbte, aber benötigte Teil des Zustandes der ursprünglichen *Angestellten*-Klasse nachgereicht werden.

```

interface AngestellteIF {
    short Alter();
    long  Gehalt();
};

class Angestellte : AngestellteIF (extent AlleAngestellte) {
    attribute long  PersNr;
    attribute string Name;
    attribute date  GebDatum;
};

class Hiwis extends Studenten : AngestellteIF (extent AlleHiwis) {
    attribute long  PersNr;
    attribute date  GebDatum;
    attribute short Arbeitsstunden;
};

```

Man beachte, daß die *HiWis* nun nicht in der Extension *AlleAngestellten* enthalten sind. Dazu müßte man diese Extension der Schnittstelle *AngestellteIF* zuordnen, was aber nach ODMG-Standard nicht möglich ist. Konflikte bei gleichbenannten Methoden werden im ODBG-Modell dadurch vermieden, daß Ableitungen, bei denen solche Konflikte auftreten würden, verboten sind.

## 16.13 Die Anfragesprache OQL

*OQL (Object Query Language)* ist eine an SQL angelehnte Abfragesprache. Im Gegensatz zu SQL existiert kein Update-Befehl. Veränderungen an der Datenbank können nur über die Mutatoren der Objekte durchgeführt werden.

Liste alle C4-Professoren (als Objekte):

```

select p
from p in AlleProfessoren
where p.Rang = "C4";

```

Liste Name und Rang aller C4-Professoren (als Werte):

```

select p.Name, p.Rang
from p in AlleProfessoren
where p.Rang = "C4";

```

Liste Name und Rang aller C4-Professoren (als Tupel):

```

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

```

Liste alle Angestellte mit einem Gehalt über 100.000 Euro:

```
select a.Name
from a in AlleAngestellte
where a.Gehalt() > 100.000;
```

Liste Name und Lehrbelastung aller Professoren:

```
select struct (n: p.Name, a: sum(select v.SWS from v in p.liest))
from p in AlleProfessoren;
```

Gruppieren alle Vorlesungen nach der Semesterstundenzahl:

```
select *
from v in AlleVorlesungen
group by kurz: v.SWS <= 2, mittel: v.SWS = 3, lang: v.SWS >= 4;
```

Das Ergebnis sind drei Tupel vom Typ

```
struct (kurz: boolean, mittel: boolean, lang: boolean,
       partition : bag(struct(v: Vorlesungen)))
```

mit dem mengenwertigen Attribut *partition*, welche die in die jeweilige Partition fallenden Vorlesungen enthält. Die booleschen Werte markieren, um welche Partition es sich handelt.

Liste die Namen der Studenten, die bei Sokrates Vorlesungen hören:

```
select s.Name
from s in AlleStudenten, v in s.hoert
where v.gelesenVon.Name = "Sokrates"
```

Die im relationalen Modell erforderlichen Joins wurden hier mit Hilfe von Pfadausdrücken realisiert. Abbildung 16.14 zeigt die graphische Darstellung des Pfadausdrucks von *Studenten* über *Vorlesungen* zu *Professoren*.

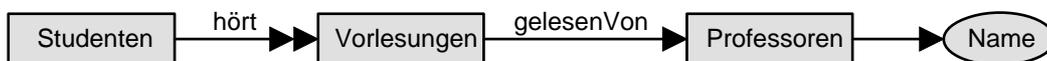


Abbildung 16.14: Graphische Darstellung eines Pfadausdrucks

Der Ausdruck *s.hoert* ergibt die Menge von Vorlesungen des Studenten *s*. Pfadausdrücke können beliebige Länge haben, dürfen aber keine mengenwertigen Eigenschaften verwenden. Verboten ist daher eine Formulierung der Form

```
s.hoert.gelesenVon.Name
```

da *hoert* mengenwertig ist. Stattdessen wurde in der *from*-Klausel die Variable *v* eingeführt, die jeweils an die Menge *s.hoert* gebunden wird.

Die Erzeugung von Objekten geschieht mit Hilfe des Objektkonstruktors:

```
Vorlesungen (vorlNr: 4711, Titel: "Selber Atmen", SWS: 4,
    gelesenVon : ( select p
                  from p in AlleProfessoren
                  where p.Name = "Sokrates"));
```

## 16.14 C++-Einbettung

Zur Einbindung von objektorientierten Datenbanksystemen in eine Programmiersprache gibt es drei Ansätze:

- Entwurf einer neuen Sprache
- Erweiterung einer bestehenden Sprache
- Datenbankfähigkeit durch Typ-Bibliothek

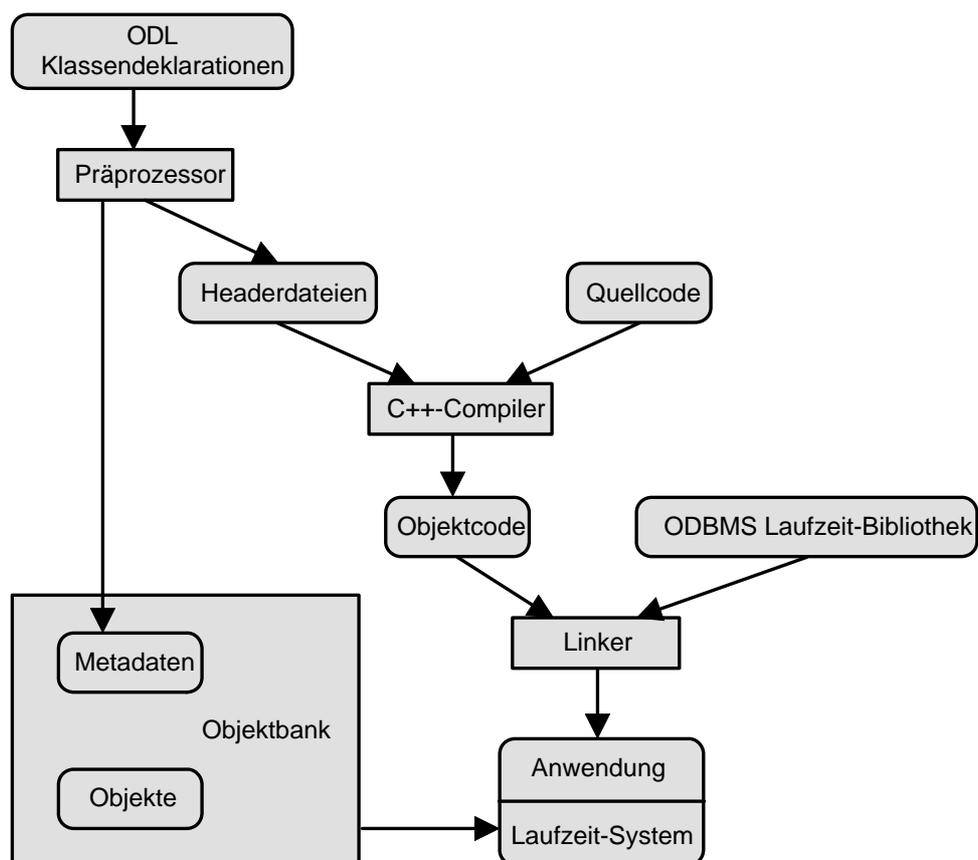


Abbildung 16.15: C++-Einbindung

Die von der ODMG gewählte Einbindung entspricht dem dritten Ansatz. Ihre Realisierung ist in Abbildung 16.15 dargestellt. Der Benutzer erstellt die Klassendeklarationen und den Quellcode der Anwendung. Die Klassendeklarationen werden mit Hilfe eines Präprozessors in die Datenbank eingetragen. Zusätzlich werden Header-Dateien in Standard-C++ erzeugt, die durch einen handelsüblichen C++-Compiler übersetzt werden können. Der Quellcode enthält die Realisierungen der den Objekttypen zugeordneten Operationen. Der übersetzte Quellcode wird mit dem Laufzeitsystem gebunden. Das Laufzeitsystem sorgt in der fertigen Anwendung für die Kommunikation mit der Datenbank.

Zur Formulierung von Beziehungen zwischen persistenten Objekten bietet die C++-Einbettung die Typen *d\_Rel\_Ref* und *d\_Rel\_Set* an:

```
const char _liest[]      = "liest";
const char _gelesenVon[] = "gelesenVon";

class Vorlesungen : public d_Object {
    d_String Titel;
    d_Short  SWS;
    ...
    d_Rel_ref <Professoren, _liest> gelesenVon;
}

class Professoren : public Angestellte {
    d_Long PersNr;
    ...
    d_Rel_Set <Vorlesungen, _gelesenVon> liest;
}
```

Es wurden hier zwei Klassen definiert. *Vorlesungen* ist direkt vom Typ *d\_Object* abgeleitet, *Professoren* ist über *d\_Object* und dann über *Angestellte* (nicht gezeigt) abgeleitet. Der Typ *d\_object* sorgt dafür, daß von *Vorlesungen* und *Professoren* nicht nur transiente, sondern auch persistente Instanzen gebildet werden können. Die Typen *d\_String*, *d\_Short* und *d\_Long* sind die C++-Versionen der ODL-Typen **string**, **short** und **long**.

In der Klasse *Vorlesungen* referenziert das Attribut *gelesenVon* durch *d\_Rel\_Ref* ein Objekt vom Typ *Professoren*. Als zweites Argument in der Winkelklammer wird die entsprechende inverse Abbildung *liest* angegeben. In der Klasse *Professoren* referenziert das Attribut *liest* durch *d\_Rel\_Set* eine Menge von Objekten vom Typ *Vorlesungen*. Als zweites Argument in der Winkelklammer wird die entsprechende inverse Abbildung *gelesenVon* angegeben.

Zum Erzeugen eines persistenten Objekts verwendet man den Operator **new**:

```
d_Ref <Professoren> Russel =
    new(UniDB, "Professoren") Professoren (2126, "Russel", "C4", ...);
```

Hierbei ist *Russel* eine Variable vom Typ *d\_Ref* bezogen auf *Professoren*, die auf das neue Objekt verweist (im Gegensatz zu *d\_Rel\_Ref* ohne inverse Referenz). Als zweites Argument wird der Name des erzeugten Objekttypen als Zeichenkette angegeben.

Als Beispiel einer Anfrage wollen wir alle Schüler eines bestimmten Professors ermitteln:

```
d_Bag <Studenten> Schüler;
char * profname = ...;
d_OQL_Query anfrage ("select s
                      from s in v.Hörer,
                      v in p.liest,
                      p in AlleProfessoren
                      where p.Name = $1");
anfrage << profname;
d_oql_execute(anfrage, Schüler);
```

Zunächst wird ein Objekt vom Typ *d\_OQL\_Query* erzeugt mit der Anfrage als Argument in Form eines Strings. Hierbei können Platzhalter für Anfrageparameter stehen; an Stelle von \$1 wird der erste übergebene Parameter eingesetzt. Dies geschieht mit dem <<-Operator der Klasse *d\_OQL\_Query*. Die Anfrage wird mit der Funktion *d\_oql\_execute* ausgeführt und das Ergebnis in der Kollektionsvariablen vom Typ *d\_Bag* (Multimenge mit Duplikaten) zurückgeliefert.