11. Objektorientierte Datenbanken

- relationale Datenmodellierung ist für viele Anwendungen unzureichend
  - ingenieurwissenschaftliche Anwendungen (Geo-Wissenschaften)
  - Büroanwendungen
  - CAD/CAM
  - Biologie
  - Medizin
- evolutionärer Lösungsansatz: Verbesserung der relationalen DBMS (z. B. Informix)
  - komplexe Datentypen
  - Integration benutzerdefinierter Datentypen und Operationen
- revolutionärer Ansatz: Entwicklung eines DBMS
  - basierend auf einer OO Sprache wie C++
  - erster Standard der ODMG:
    (i) einheitliches Objektmodell
    (ii) OQL als Anfragesprache,
    (iii) ODL als Datendefinitionssprache
    (iv) Anbindung an OO-Sprachen wie C++, Java und Smalltalk

Defizite relationaler Systeme

- Datenbank für die Verwaltung einer Katasterdatenbank:
  Point(pid, x, y)
  Strecke(sid, von, nach)
  Polygon(pid, strecke, ordnung)
- Nachteile bei der Datenmodellierung
  - Segmentierung: Verteilung der Objektinformation über mehrere Relationen
  - künstliche Schlüsselattribute, die keinerlei Bedeutung haben, aber vom Benutzer zu verwalten sind.
  - kein Verhalten in der Datenbank wie z. B.
    Berechnung des Flächeninhalts,
    Translation eines Objekts,
    Schnittoperationen.
  - Anbindung an Programmiersprachen:
    auf Grunde der unzureichenden Mächtigkeit der Anfragesprache
    mengenorientierte Anfragesprache <---> satzorientierte Programmiersprache

OO-Datenmodellierung

- vermeidet die Probleme relationaler Systeme

Konzepte der OO-Datenmodellierung

- Beschreibung komplexer Datenstrukturen und deren Verhalten
  - benutzerdefinierte Operationen werden Teil der Datenbank
- Bewahrung des Geheimnisprinzips (Abstrakte Datentypen):
  - Zugriff auf den Zustand eines Objekts über vordefinierte Funktionen
- System sorgt sich um die Identifizierung der Objekte
- komplexe Datenstrukturen und Operationen können in der DB-Programmiersprache realisiert werden:
  einfachere Anbindung der DB-Programmiersprache an Host-Programmiersprache

11.1 Objekte

1. Identität:
   - eindeutig im System
   - ändert sich nicht während der Lebenszeit eines Objekts
2. Objekttyp
   - definiert die Datenstruktur + Verhalten
3. Wert (Zustand)
   - Objekte besitzen zu jedem Zeitpunkt einen Zustand
   - Zustand ist gegeben durch die Belegung der Attribute und der mit anderen Objekten bestehenden Beziehungen

Objektidentität

in relationalen Systemen:

- Identifizierung eines Datensatzes durch die Werte des Schlüssels (identity through contents)
- z. T. sind keine Änderungen von Schlüssel möglich

Nachteil:
Objekte mit gleichen Werten sind stets auch identisch!

- Einführung von künstlichen Schlüsseln (siehe oben: Kataster-Datenbank)

in OODBMS:

- Identifizierung durch logische OIDs
- OIDs werden vom System verwaltet

Objekttyp

- gleichartige Objekte werden durch einen einheitlichen Objekttyp (Klasse) beschrieben.
- Objekte = Instanzen des Objekttyps
- Extension = Menge aller Instanzen

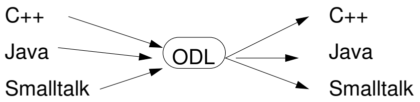
Wert eines Objekts

- Wertebereiche der Attribute sind nicht notwendigerweise atomar, sondern bestehen teilweise aus komplexen Strukturen wie z. B.
  - Mengen
  - Listen
  - Tupel

11.2 Definition von Objekttypen

- Dieser Abschnitt basiert auf der *Spezifikationssprache* ODL, die Teil des ODMG-Standards ist.

Anwendung ist implementiert in — Datenbank ist implemntiert in



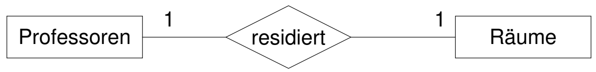
- Objekttyp
  - Strukturbeschreibung
  - Verhaltensbeschreibung
  - Typeigenschaften
- Beispiel:

```
interface Professoren {
        attribute long PersNr;
        attribute String Name;
        attribute String Rang;
};
```


Beziehungen

1:1-Beziehung zwischen Objekten:

- Modellierung in beiden Objekttypen



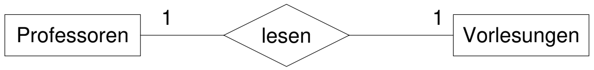

```
interface Professoren {
    attribute long PersNr;
    …
    relationship Räume residiert_in;
};

interface Räume {
    attribute long RaumNr;
    …
    relationship Professoren beherbergt;
};
```

- Defizite
  - Symmetrie ist nicht notwendigerweise gewährleistet
  - Konsistenz der 1:1-Beziehung ist nicht gewährleistet
- Überprüfung der Symmetrie und Konsistenz durch den inverse-Ausdruck:
  - relationship Räume residiert_in inverse Räume::beherbergt;

    Professoren p und Räume r : p.residiert_in = p $\Leftrightarrow$ r.beherbergt = p

1:N Beziehungen



interface Professoren {
    attribute long PersNr;
    …
    relationship Set⟨Räume ⟩liest inverse Vorlesungen::gelesen_von;
};
interface Vorlesungen {
    attribute long VNr;
    …
    relationship Professoren gelesen_von inverse Professoren::liest;
};

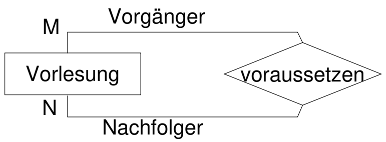
- Modellierung benutzt den Set-Konstruktor
- Angabe der Inverse-Operation stellt Konsistenz der 1:N-Beziehung sicher:
  Professoren p, Vorlesung v: v.gelesen_von = p ⇔ v ∈ p.liest

M:N Beziehungen (binär, nicht rekursiv, ohne Attribute)

interface Studenten {
    attribute long MatrNr;
    …
    relationship Set⟨Vorlesungen⟩ hört inverse Vorlesungen::Hörer;
};
interface Vorlesungen {
    attribute long VNr;
    …
    relationship Set⟨Studenten⟩ Hörer inverse Studenten::hört;
};

- Angabe der Inverse-Operation stellt Konsistenz der M:N-Beziehung sicher:
  Studenten s, Vorlesung v: s ∈ v.Hörer ⇔ v ∈ s.hört

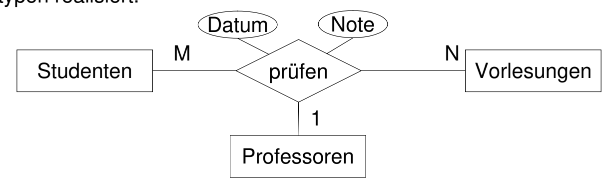
rekursive N:M-Beziehungen



interface Vorlesungen {
    attribute long VNr;
    …
    relationship Set⟨Vorlesungen⟩ Vorgänger inverse Vorlesungen::Nachfolger;
    relationship Set⟨Vorlesungen⟩ Nachfolger inverse Vorlesungen::Vorgänger;
};

drei- und mehrstellige Beziehungen, Beziehungen mit Attributen

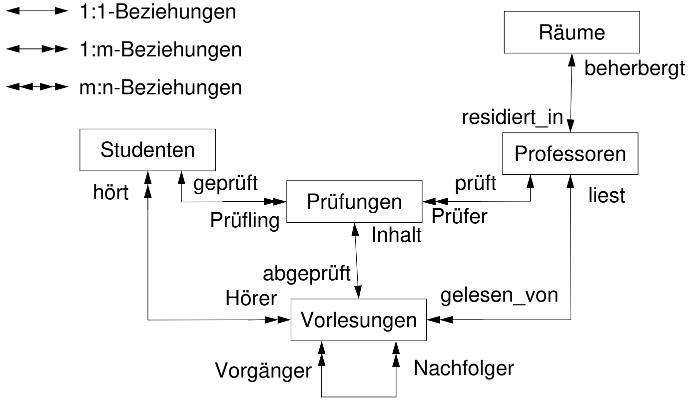
- analog zum Netzwerkmodell wird solch ein Beziehungstyp durch einen eignen Datentypen realisiert.



interface Prüfungen {
    attribute Struct⟨short Tag, short Monat, short Jahr⟩ Datum;
    attribute float Note;
    relationship Professoren Prüfer inverse Professoren::prüft;
    relationship Studenten Prüfling inverse Studenten::geprüft;
    relationship Vorlesungen Inhalt inverse Vorlesungen::abgeprüft;
};

- Struct ist die Konstruktor-Operation für ein Tupel
- durch Struct- und Set-Konstruktoren können bliebig komplexe Datentypen erzeugt werden

Graphische Rep. des Objektmodells



Datenbanksysteme

Typeigenschaften: Extension und Schlüssel

- Extension = Menge aller Instanzen eines Objekttyps.
- Schlüssel = Menge von Attributen, die einen Datensatz in einer Extension identifiziert.
  - im Gegensatz zum relationalen Modell: Schlüssel wird nicht zur Modellierung der Beziehungen benutzt!

```
interface Studenten {
    extent AlleStudenten;
    key MatrNr;
    attribute long MatrNr;
    …;
};
```


Datenbanksysteme

11.3 Verhalten

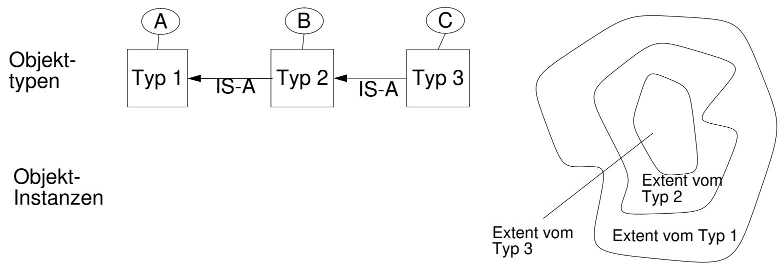
- Verhalten: beschrieben durch dem Objekttyp zugeordneten Operationen
- ausschließlich Operationen lesen und verändern den Objektzustand
  - Konsistenz soll bewahrt bleiben (Programmierer)
  - Konsistenz kann nicht durch den Benutzer gefährdet werden.
- Schnittstelle stellt folgende Operationen zur Verfügung:
  - Konstruktoren und Destruktoren
  - Beobachter-Operationen
  - Mutatoren: einzige (!) Möglichkeit zum Änderen des Objektzustands.
- Angabe der Signatur einer Operation bei der Definition eines Interfaces:
  - Name der Operation
  - Typen der Parameter
  - ggf. einen Rückgabewert
  - ggf. eine Ausnahmebehandlung

```
interface Professoren { …
    Prüferqualität(out float) raises (KeineErfahrung);
    Befördert() raises (HöchsteStufe); …}
```


Datenbanksysteme

Vererbung

- ähnliche Objekttypen können entsprechen wie bei einer IS-A-Beziehung zusammengefaßt werden:
  - Obertyp enthält alle gemeinsamen Eigenschaften.
  - Untertyp erben die Eigenschaften des Obertyps und besitzen zusätzlich noch weitere Eigenschaften.



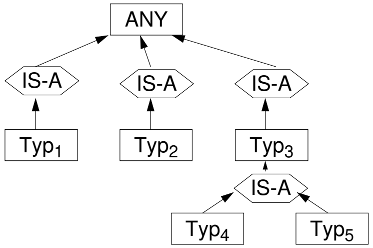
Inklusionspolymorphismus

Da Untertypen alle Eigenschaften eines Obertyps besitzen, kann folgende Substituierbarkeitsregel angewendet werden:

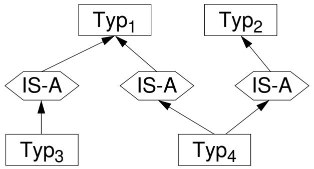
Eine Instanz eines Untertyps ist überall dort einsetzbar, wo eigentlich eine Instanz eines Obertyps gefordert wird.

Einfache und Mehrfache Vererbung

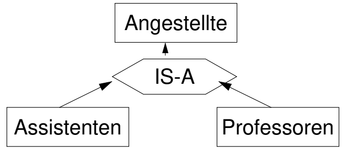
- einfache Vererbung:
  Jeder Objekttyp hat höchstens einen direkten Obertyp.



- mehrfache Vererbung:
  Ein Objekttyp kann mehrere oberobjekte besitzen



Beispiel einer Typhierarchie:




```
interface Angestellte {
    extent AlleAngestellten;
    attribute long PersNr;
    attribute String Name;
    attribute Struct⟨short Tag, short Monat, short Jahr⟩ GebDatum;
    Alter(out short);
    Gehalt(out long);
}

interface Assistenten: Angestellte {
    extent AlleAssistenten;
    attribute String Fachgebiet;
}

interface Professoren: Angestellte {
    extent AlleProfessoren;
    attribute String Rang;
    relationship Räume residiert_in …
    relationship Set⟨Vorlesung⟩ liest …
    relationship Set⟨Prüfungen⟩ prüft …
}
```


Spätes Binden von Operationen

- Vererbung bezieht sich nicht nur auf Attribute, sondern insbesondere auf Operationen
  - Untertypen können die Implementierung einer Operation vom Obertyp direkt übernehmen
  - Untertyp kann aber auch eine Verfeinerung der Operaion vornehmen
- Beispiel: Operation Gehalt in der Objekthierarchie Angestellte
  - Angestellte: Gehalt1(out long x){ x = 3000 + (Alter() - 21)*100;}
  - Assistenten: Gehalt2(out long x){ x = 3500 + (Alter() - 21)*125;}
  - Professoren: Gehalt3(out long x){ x = 4000 + (Alter() - 21)*150;}
- Typüberprüfung zur Laufzeit
  - Objekt kennt seinen Objekttyp
  - in Abhängigkeit des Objekttyps wird die entsprechende Implementierung ausgewählt, z. B.

```
sum = 0;
for each a in AlleAngestellte do {
    a->Gehalt(x); sum = sum + x;
}
```


11.4 Die Anfragesprache OQL

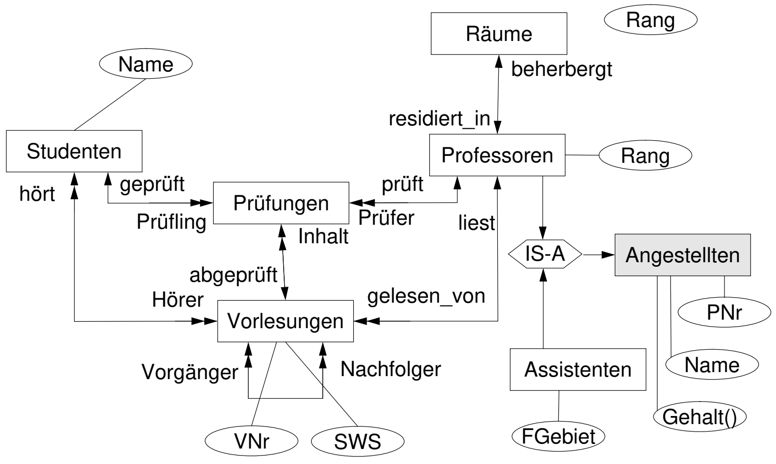
Ziele beim Entwurf

- Erweiterung von SQL
- basiert auf dem ODMG-Standard
- mächtiges Typkonzept
- “funktionale” Sprache

Vorteile gegenüber SQL

- orthogonal
- Verarbeitung komplex strukturierter Objekte
- Objekte besitzen Methoden, die in OQL bei der Anfrageformulierung genutzt werden können.

Die Beispiele dieses Abschnitts basieren folgendem Modell.



Datentyp einer Anfrage

- Einfache Anfragen unterscheiden sich nicht wesentlich von SQL-Anfragen.
  - In der from-Klausel werden Tupelvariablen über das Schlüsselwort in an die Relation gebunden.
- Ein Ergebnis eine Anfrage besitzt einen Typ, der durch die in ODMG verfügbaren Möglichkeiten zur Typkonstruktion beschrieben wird.

Beispiele

- Gib alle Namen der C4-Professoren:

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

  Das Ergebnis besitz den Typ **Bag<String>**
- Gib alle Namen und Personalnummern der C4-Professoren:

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

  Das Ergebnis besitzt den Typ **Bag<Struct<String,Intgerer>>**

- Eine Anfrage kann auch als Ergebnis eine Menge von Objekten oder ein einzelnes Objekt liefern.
- Komponenten eines strukturierten Anfragetyps können auch wieder mengenwertig sein.

Beispiel

- Gib alle C4-Professoren.

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

  Das Ergebnis besitzt den Datentyp **Set<ANY>**
- Gib den Professor mit der Personalnummer "7".

  ```
  element(select p
          from p in AlleProfessoren
          where p.PNr = 7);
  ```

  Das Ergebnis besitzt den Datentyp **ANY**.
- Gib alle Professorennamen und deren Vorlesungen.

  ```
  select struct (n:p.Name, vset: p.liest)
  from p in AlleProfessoren;
  ```

  Das Resultat dieser Anfrage ist vom Typ **Bag<Struct<String,Set<ANY>>>.**

Geschachtelte Anfragen

- OQL-Anweisungen können in der select-, from- und where-Klausel beliebig geschachtelt werden.

Beispiele

- Gib alle Namen von C4-Professoren und deren vierstündige Vorlesungen.

  ```
  select struct (n:p.Name, vset: ( select distinct v
                                   from v in p.liest
                                   where v.SWS = 4))
  from p in AlleProfessoren
  where p.Rang = "C4";
  ```
- Als Alternative könnte man die where-Klausel auch in die from-Klausel hochziehen.

  ```
  select …
  from p in ( select q
              from q in AlleProfessoren
              where q.Rang = "C4")
  ```

Pfadausdrücke

- Statt explizit einen Join zu berechnen wird in OQL mittels des Punktoperators eine Objektreferenz “verfolgt”. Man spricht dann auch von **impliziten** bzw. **funktionalen** Joins.
- Ein **Pfadausdruck** besteht aus einer Tupelvariablen und einem Attribut (Beziehung) bzw. einer Kette von Attributen (Beziehungen), die mit dem Punktoperator an die Tupelvariable gebunden sind. Der Wertbereich eines Attributs (Beziehung) darf nicht mengenwertige Elemente besitzen.

Beispiele

- Gib die Namen aller Studierenden, die eine Vorlesung bei Prof. Kemper besucht haben.

  select s.Name
  from s in AlleStudenten, v in s.hört
  where v.gelesen_von.Name = “Kemper”;

  Bemerkungen:
  - In der from-Klausel wird ein Join zwischen den zu den Tupelvariablen s und v zugeordneten Mengen berechnet.
  - In der where-Klausel wird ein impliziter Join formuliert.

Problem

- Eine Anfrage bezieht sich auf einen Pfad $A_1.A_2.\dots.A_{i-1}.A_i.A_{i+1}.\dots.A_n$ bezieht, wobei A_i mengenwertig ist. Wie kann diese Anfrage formuliert werden?

  select …
  from u in A_1, v in v.$A_2.A_3\dots A_{i-1}$, w in v.$A_i.A_{i+1}\dots A_n$
  where …

Objekterzeugung

- Wie üblich werden Objekte mittels Konstruktoren erzeugt.

Beispiele

- Erzeuge ein neues Objekt der Klasse Professoren mit PNr = 8, Rang = “C4” und Name = “Fritz”.

  Professoren (PNr: 8, Rang: “C4”, Name: “Fritz”);
  Alle nicht initialisierten Felder bekommen einen voreingestellten Wert zugewiesen.
- Erzeuge ein neues Objekt der Klasse Vorlesung mit VNr = 9999, Titel = “Informatik I”, SWS = 4 und gelesenVon sei der Professor mit PNr = 8.

```
Vorlesungen(    VNr: 9999, Titel: “Informatik I”, SWS: 4,
                gelesenVon: element( select p
                                     from p in Professoren
                                     where p.PNr = 8));
```


Methodenaufruf

- Methoden, die als Funktionsprozeduren implementiert sind, werden wie gewöhnliche Attribute angesprochen. Zusätzlich müssen noch entsprechende Parameter mitgegeben werden.
- Zu beachten ist dabei aber, daß (aufgrund von Klassenerweiterungen) Methoden dynamisch gebunden werden können.

Beispiele

- Gib alle C4-Professoren mit einem Gehalt > 120000.

  select a.Name
  from a in AlleProfessoren
  where a.Gehalt() > 120000;
- Gib alle Angestellten mit einem Gehalt > 120000.

  select a.Name
  from a in AlleAngestellten
  where a.Gehalt() > 120000;

  Da die Klasse Angestellten eine abstrakte Oberklasse von Professoren und Assistenten ist, wird die Methode Gehalt (je nach Objekttyp) dynamisch gebunden.

Quantoren

- In OQL steht sowohl der Existenzquantor als auch der Allquantor zur Verfügung.

Allquantor

for all x in <Collection>: <Predicate>

- Hierbei ist <Collection> eine Menge oder eine Multimenge und <Predicate> ein Boolscher Ausdruck.
- Das Ergebnis ist genau dann *true*, falls alle x aus <Collection> <Predicate> erfüllen.

Existenzquantor

exists x in <Collection>: <Predicate>

- Das Ergebnis ist genau dann true, wenn es mindestens ein x aus <Collection> gibt, welches <Predicate> erfüllt.

Berechnung von Aggregatsfunktionen

- Eine Aggregatsfunktion ist definiert auf einer Menge bzw. eine Multimenge von numerischen Werten und liefert einen Zahl zurück.
- Gib die Summe aller Gehälter von C4-Professoren

```
sum(select a.Gehalt()
    from a in AlleProfessoren
    where a.Rang = "C4");
```

Die Gruppenbildung und die Anwendung von Aggregatsoperationen auf die einzelnen Gruppen kann in OQL ohne die "group by"-Klausel formuliert werden.

- Gib die Summe der Gehälter der Professoren gruppiert bzgl. des Rangs.

```
select struct(r: a.Rang, s:sum(  select b.Gehalt()
                                 from b in AlleProfessoren
                                 where b.Rang = a.Rang))
from a in AlleProfessoren;
```


Partitionieren durch group-by

- Trotz der in OQL vorhandenen Möglichkeit Mengen (bzw. Multimengen) zu gruppieren, bietet OQL die aus SQL bekannte "group by"-Klausel an.
- Diese besitzt aber zusätzlich noch die Möglichkeit, Mengen (bzw. Multimengen) in verschiedene Bereiche zu partitionieren.

Beispiel

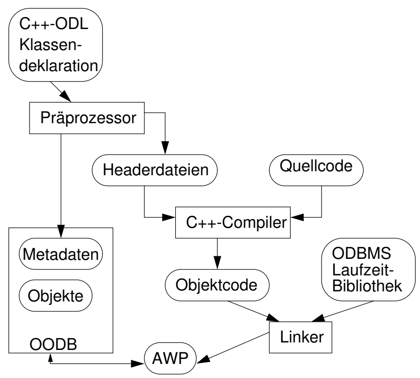
- Partitioniere alle Vorlesungen in kurze (SWS <3), mittlere (SWS = 3) und lange (SWS > 3) Vorlesungen.

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

Der Ergebnistyp hierfür ist
Struct<Boolean, Boolean, Boolean, Set<Vorlesungen>>.
Das Tupel (true, false, false, {VId_1, …, VId_n}) enthält alle kurzen Vorlesungen.

11.5 Anbindung an C++

- erfolgt über eine Bibliothek
  - am einfachsten zu realisieren
  - schwierig zu benutzen



C++-ODL Klassendeklaration:

```
class Vorlesung {
    String Titel;
    short SWS;
    Ref⟨Professoren⟩ gelesen_von
    inverse Professoren::liest;
};
```

Quellcode:
Set⟨Studenten⟩ Schüler;

oql(Schüler, "select s from s in v.Hörer, v in p.liest, p in AlleProfessoren where p.Name = "Kemper"");

Zusammenfassung

- OODBMS besitzen viele Vorteile gegenüber relationalen Systemen
  - erweiterbares Typsystem
  - komplexe Objekte
  - Verhalten
- Zukunft von OODBMS ist aber nicht unbedingt rosig
  - derzeitige Systeme laufen noch nicht stabil genug
  - ODMG-Standard zu spät gekommen
  - Anfrageoptimierung
  - Anbieter relationaler Systeme entwickeln objektrelationale Systeme mit einem neuen Standard für SQL (SQL3)
  - Anwender bevorzugen evolutionäre Entwicklungen
- eine kleine Auswahl von kommerziellen OODBMS:

  GemStone, Illustra (Informix), O_2, Objectivity/DB, ObjectStore, Versant, …