

Zusammenfassung Objekt- und dokumentenzentrierte Informationssysteme

Philipp Jäcks

21. Januar 2016

Inhaltsverzeichnis

1	Einführung	3
1.1	Entwicklung der Datenbankmodelle	3
1.2	Nachteile relationaler DB	7
2	Objektorientierte Modelle und Operationen	10
2.1	Konzepte objektorientierter Programmiersprachen	10
2.1.1	Prinzipien des OO Entwurfs	10
2.2	Einschub	14
2.2.1	”Reines” OO programmieren mit Smalltalk	14
2.2.2	Ist C++ streng typisiert?	14
2.3	Ein objektorientiertes Datenbankmodell: Überblick	14
2.4	Strukturteil eines OODM	15
2.4.1	Typkonstruktoren	16
2.4.2	Objektidentität	17
2.4.3	Klassen und Typen	19
2.4.4	Beziehungen zwischen Klassen	21
2.4.5	Strukturvererbung	22
2.4.6	Integritätsbedinungen	24
2.5	Operationenteil eines objektorientierten Datenbankmodells	25
2.6	Höhere Konzepte eines objektorientierten Datenbankmodells	27
2.7	Klassifikation objektorientierter Datenbanksysteme	32
3	Der ODMG-Standard	36
3.1	Modelle für OODBPL	36
3.2	der ODMG-Standard	36
3.3	Der Strukturteil und höhere Konzepte des ODMG-Standards	37
3.4	Die ODL des ODMG-Standards	40
3.5	Der Operationenteil und die OQL des ODMG-Standards	41
3.6	Umsetzung in OODBPL-Systemen	43

4	Das objektrelationale Modell	45
4.1	Einführung in objektrelationale Konzepte	45
4.2	Der SQL:1999/SQL:2003 Standard	45
4.3	Der Strukturteil des ORDB-Modells	46
4.4	Der Operationenteil des ORDB-Modells	50
4.5	Höhere Konzepte des ORDB-Modells	51
4.6	Umsetzung in ORDBMS	52
4.7	Fazit und Vergleich	52

1 Einführung

- Einsatz hierarchischer DB ab 1970
- Relationale DB Einsatz ab 1980; hierarchische DB bleiben bestehen
- OO DB Einsatz ab Ende 80er; relationale DB bleiben bestehen
- XML DB Einsatz ab 2000; alle alten DB bleiben
- heute: hierarchische DB speichern meisten unternehmenskritischen Daten; gefolgt von Relationalen
- OO und XML DB aber in jedem guten relationalen System enthalten
- OO DBMS: ab Ende 80er; Startup Unternehmen; komplexes DB-Modell; später unvollkommener, inkonsistenter Standard
- Trend OO DBMS: instabil, bei read-write-Transaktionen wenig performant; unvollständig (Zugriffsrechte, Recovery, Sichten, Integritätsbedingungen); verschwinden wieder vom Markt
- Objektrelationale DB: Relationale DB integrieren OO Konzepte -> überholen OO DBMS

1.1 Entwicklung der Datenbankmodelle

Relationenmodell, RDBSs: Objekte dargestellt durch Zeilen in Tabellen

SET OF RECORD

A_1 : Standard-Datentyp₁

...

A_n : Standard-Datentyp_n

END;

Objekttypen	Eigenschaft
<i>Personen</i>	<i>Name</i> (bestehend aus <i>Vor-</i> und <i>Nachname</i>) <i>Adresse</i> (bestehend aus <i>PLZ</i> , <i>Ort</i> , <i>Straße</i> und <i>Hausnummer</i>) <i>Hobbies</i> (bestehend aus einer Menge von <i>Hobbies</i>) <i>Geburtsdatum</i>

Tabelle 1: Beispiel: Modellierung von Personen

In RDBS:

- Eigenschaften *Name*, *Adresse* in Komponenten zerlegen
- Eigenschaft *Hobbies* auslagern (1NF)

Vorname	Nachname	PLZ	Ort	Straße	Hausnr	Gebdat

Vorname	Nachname	PLZ	Gebdat	Hobby

- Eigenschaften *Vorname*, *Nachname*, *PLZ* und *Geburtsdatum* sollen Schlüssel sein

Eigenschaften relationaler DB

- Starre Strukturen (Relationenschemata)
- Einfache Strukturen (nur Tabelle, 1NF)
- Für einfache Attributwerte (Zahlen, Zeichenketten, also Standard-Datentypen)
- Mit fester Semantik (Built-in Funktionen für Std-Typen)
- Austausch von Daten (etwa im Web) erschwert: Trennung Schema (Relationenschema) und Instanz (Relation), zum Versand muss beides zusammengefasst werden

Entwicklung: OO DB

- Forschung Ende 80er, Hype 90er => Nischenprodukt für neue Anwendungen; Ende 90er in RDBS
- Konzepte: komplexe Strukturen (statt nur Tabellen nun auch beliebig strukturierte Objekte)
Komplexe Attributwerte (Typen können konstruiert werden)
mit variabler Semantik (Methoden können def. werden)

Entwicklung: XML DB

- Forschung 90er, Hype Anfang 00er => Nischenprodukt für bestimmte Anwendungen; XML-Konzepte im objekrelationalen SQL-Standard
- Konzepte: Ausgangspunkt Dokumentbeschreibungssprache (Markup-Sprache) statt starrer Datenstruktur
Variable Strukturen zur Beschreibung von Daten und Dokumente
Komplexe Strukturen mit variabler Semantik mgl
Austauschformat im Web zum Dokument- und Datenaustausch

Entwicklung: Digitale Bibliotheken (ab 2000)

- XML-Dokumente (oder andere Dokumentformate, pdf, doc... oft textlastig) langfristig speichern

- Auffindbar machen über strukturierte Metadaten (etwa in XML oder im relationalen DB-Modell)
- Spezielle Anforderungen: Dokumente haben Wert, können gekauft werden (E-Commerce)
Dokumenten weltweit eindeutig identifiziert
Identifizierer sind persistent, nicht flüchtig, obwohl Dokument vergriffen oder gesperrt sein kann
Versionierung der Dokumente
- Suche nach Features in Texten (Stichworte,...) oder nach strukturierten Metadaten

Entwicklung: Multimedia DB (ab 95)

- Dokumente nicht nur textlastig, sondern Bild, Audio, Video, 2D/3D-Geoobjekt,...
- Problem: Features sind nicht nur Stichworte, sondern
Bei Bildern: Farben und Farbverteilungen, erkannte Objekte wie Gesichter, Muster, Strukturen,...
Bei Videos: Schnitte, Szenen, Bewegungen,...
Bei Audio: speziell Musik, Melodien, Dynamik, Rhythmus,...
Bei Geoobjekten: Enthaltensein, Schnittflächen,...

Darstellung komplexer Objekte: OODBS

In OO DBS sind nicht nur Std-Typen für Eigenschaften von Objekten erlaubt, sondern auch wieder Anwendung von Typkonstruktoren.

```

CLASS Personen
    TYPE TUPLE
        (Name: TUPLE
            (Vorname: STRING,
             Nachname: STRING) ,
        Adresse: TUPLE
            (PLZ: INTEGER,
             Ort: STRING,
             Strasse: STRING,
             Hausnummer: INTEGER) ,
        Hobbies: SET (Hobby: STRING) ,
        Geburtsdatum: DATE)

```

Objektidentität

- RDBS: Schlüsselwerte können sich ändern, Identität eines Objektes geht evtl verloren
- OODBS:
 - Objekte existenzunabhängig von Werten ihrer Eigenschaften, d.h. Identität bleibt gleich, während sich Eigenschaften ändern

- in technischen Anwendungen: teilweise Objekte nicht durch äußere Eigenschaften unterscheidbar (Bsp: Menge von Schrauben gleicher Art)
- Entscheidung evtl durch Position, aber nicht durch Namen

Ist-Hierarchie (Vererbung)

- Fehlt in RDBS; quasi nur über viele Fremdschlüssel simulierbar
- OODBS:
 - Objekttypen in Vererbungshierarchie mgl

```

CLASS Studenten INHERITS Personen
    TYPE TUPLE
        (Matrikelnummer: INTEGER,
         Studienfach: STRING,
         Vater: Personen,
         Mutter: Personen,
         ...)

```

- Vererbung (Studenten sind spezielle Personen) und Komponentenobjekte (Vater und Mutter sind Personen)

Methoden statt Host-Prozeduren

- RDBS: spezielle Prozeduren und Funktionen von *außen* aufgesetzt
SQL Ausdruck oder sogar Programm in höheren Programmiersprache
Bsp: Alter einer Person aus Geburtsdatum: Sicht in SQL auf Basistabelle oder C-Programm mit eingebetteter SQL-Anfrage
- OODBS: neben Eigenschaften auch die mit ihnen durchführbaren Methoden in die Objekttyp-Definition einkapseln und vererben
Bsp: Alter ist in Definition erklärt (Interface, getrennt davon Impl)

Dokumente

- Text- oder große Multimediadokumente: groß, unstrukturiert/maximal semistrukturiert, variabel strukturiert (Text nicht immer starr in Kapitel/Abschnitt/...)
- in RDBS:
 - als CLOB oder BLOB (völlig unstrukturiert), Metadaten extrahieren in relationale Tabelle (Autor, Titel, Format, Länge...)
 - oder *schreddern*: Dokument in kleinste Anteile zerlegen und in relationaler Tabelle speichern mit folgenden Problemen
Relation sieht starre Struktur vor
Relation muss Ordnung der kleinste Anteile bewahren bei der Rekonstruktion des gesamten Dokuments

- in XML DB:
 - von unstrukturiert bis voll strukturiert (auch variabel)
 - Markup-Sprache für Dokumente geeignet
 - evtl stark strukturierte Anteile in Relationen gespeichert -> Side Tables

1.2 Nachteile relationaler DB

Komponenten DB-Modell

- **Strukturell:** Datenstrukturen für Anwendungsobjekte, Konzepte, Modellierung von Beziehungen zw Anwendungsobjekten, Integritätsbedingungen
im Relationenmodell: Relationen (Tabellen) für alles, Schlüssel, Fremdschlüssel
- **Operationenteil:** Generische Operationen auf Datenstrukturen und Beziehungen
im Relationenmodell: Relationenalgebra, SQL-Anfragen und SQL-Updates
- **Höhere Konzepte:** Metainformationen und objektspezifische Operationen,..
im Relationenmodell: höchstens Data Dictionary, sonst nichts

Vorteile im Relationenmodell

- **Strukturteil:** einfache, einheitliche Beschreibung der Anwendungsdaten
Exaktes, mathematisches Fundament
- **Operationenteil:**
 - Deskriptivität: was, nicht wie; mengenorientiert
 - Abgeschlossenheit: Ergebnis ist wieder eine Relation
 - Adäquatheit: alle Konzepte des Strukturteils unterstützt
 - Optimierbarkeit: System kann selbst schnellere Auswertungsreihenfolge finden
 - effiziente Impl: jede Operation der Relationenalgebra effizient implementierbar
 - Sicherheit: jede syntaktisch richtige Anfrage liefert Ergebnis
 - Orthogonalität: Alle Operationen beliebig miteinander kombinierbar

Nachteile Datenmodellierung

- Komplexe Attribute: Wertemengen oder mehrere Komponenten nur über Fremdschlüssel simulierbar
- Beziehungen: immer über Fremdschlüssel dargestellt

Nachteile Datenbankentwurf

- Methoden:

- Informale Methoden: Entity-Relationship Modell abbilden in Relationenschemata
Schwach bei komplexen Attributen (simuliert über n:m-Beziehungen)
- Formale Algorithmen: Attribute und Abhängigkeiten bestimmen Relationenschemata
Normalformen, Abhängigkeitstreue, ...
Entwurf ohne Semantik, Abhängigkeiten reichen nicht zur Anwendungsbeschreibung
- Allgemeine Schwächen: Ergebnis verliert mühsam erfasste Semantik
- mangelnde Semantik: Beschreibung der Semantik mit Abhängigkeiten zw Attributen (funktional¹, mehrwertig)
Reale Bsp komplizierter: CIM-Datenbank, Ventulfeder, Anlasserzahnkranz,.. (was bestimmt sich funktional oder mehrwertig)
vernachlässigt: Verbund- und Inklusionsabhängigkeiten

Nachteile Anfragesprache

Strukturmangel im Ergebnis

Beispiel: Dreifacher Verbund zur Rekonstruktion EINES Buches

Nachteile Anfrageoperationen

Anfragen an komplexe Attribute

- keine Unterstützung komplexer Strukturen in Anfrageformulierung
- Notwendigkeit expliziter Verbundoperationen

Nachteile Update-Operationen

Identifikation der Objekte über sichtbare Schlüssel -> Kein Unterschied zw Umzug, Kauf neues Auto

OODBS: Objekte eindeutig identifizierbar -> Unterschied zw Umzug und Autokauf

¹Attribute bestimmen eindeutig den Wert anderer Attribute, dann spricht man von funktionaler Abhängigkeit

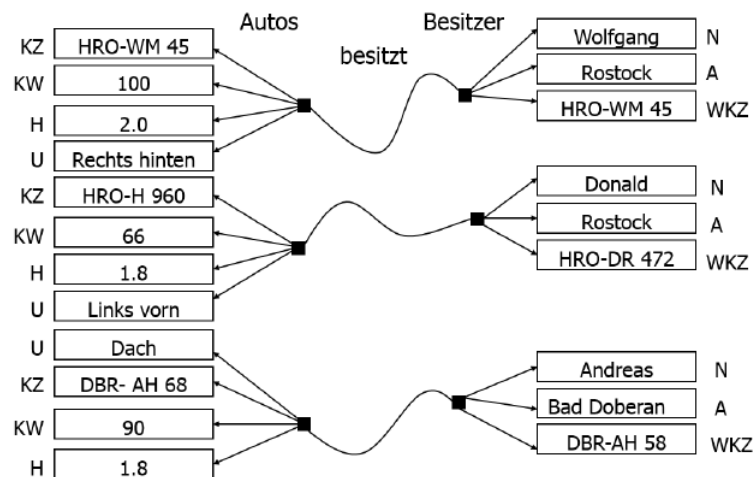


Abbildung 1: Update Operation in OODBS

Klassifikation der Probleme

- Art des Problems:
 - Systemspezifisch (konkret am System, bspw MySQL)
 - Sprachspezifisch (SQL, jeweiliger Sprachstandard)
 - Modellspezifisch (liegt am Relationenmodell)
- Schwere des Problems
 - Umständlich oder ineffizient (Formulierung von Anfragen)
 - Zusätzliche Tricks notwendig (Darstellung komplexer Objekttypen)
 - nicht machbar (bestimmte Arten von Anfragen)

2 Objektorientierte Modelle und Operationen

2.1 Konzepte objektorientierter Programmiersprachen

- starker Einfluss der OO DB-Modelle durch OO Systeme
- **Entwurfsphasen**
 1. Identifiziere Objekte der Anwendung
 2. Beschreibe Objekte der Anwendung
 3. Identifiziere Beziehungen und Gemeinsamkeiten zw Objekten
 4. Fasse Objekte mit gemeinsamen Eigenschaften zu Klassen zsm
 5. Identifiziere Beziehungen zw Klassen
 6. Bilde Klassenhierarchien
 7. Implementiere die Funktionen einzelner Klassen
 8. Entwickle Programme aus Objektbeschreibungen

Phase 2 bis 4 möglichst mit *abstrakten Datentypen*

Typen: generischer Typ T, beschreibt Menge von Objekten, ein Objekt heißt Instanz

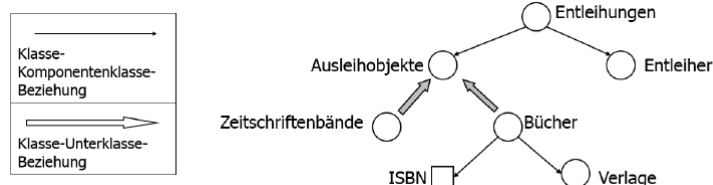
Funktionen: Operationen durch Signatur beschrieben (Typen des Def.- und Bildbereichs); formal seiteneffekt-frei (verändern nie ein Argument)

Vorbedingungen: insbesondere für partielle Fkt (bsp: Kelleroperation top(), nur anwendbar, wenn Keller nicht leer)

Axiome: Semantikbeschreibung von Fkt, Festlegung des Verhaltens

2.1.1 Prinzipien des OO Entwurfs

- Klassen sind ADT-Implementierungen (meist ohne Axiome, Vorbedingungen)
- Beziehungen zw Klassen:
 - Klasse - Komponentenkategorie: wird zur Implementierung einer anderen Klasse benutzt
 - Komponentenobjekt: im Zustand eines anderen Objekts
 - Klasse- Unterklasse: steuert die Vererbung von Attributen und Methoden



Attribute und Methoden

Komponenten eines Objekttyps

- Attribute: Eigenschaften von Objekten
- Methoden: auf Objekten durchführbare Funktionen

Einkapselung

- Schnittstelle (public): Methodensignatur (Eingabe, Ausgabe) -> Protokoll einer Botschaft
- Implementierung (private): Attribute und Methodenimplementierung
- Methodenaufruf: Senden einer Botschaft mit *Objekt.Methode* -> Objekt ist Empfänger der Botschaft
- Klasse: Menge von Objekten mit gleichen Attributen und Methoden
 Programmiersprachen: Klasse ist Implementierung eines ADT
 DB: Klasse ist Objektfabrik (und -lager)

Konstruktor und Destruktor

Zum Erzeugen (Konstruktor; zusätzlich Initialisierung des Objekts) und Löschen eines Objekts (Destruktor)

Zuweisung:

Unterscheidung zw Wert und Referenzsemantik

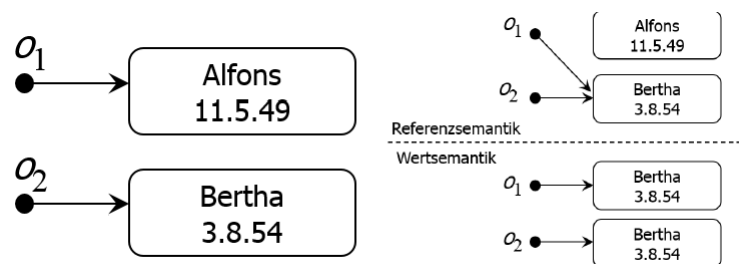


Abbildung 2: Unterscheidung Zuweisung Wert- und Referenzsemantik

Kopieren

- Flaches kopieren: lediglich Verweis auf die Referenz des zu kopierenden Objektes; Originalobjekt und Kopie teilen sich Attribute
- Deep Copy: zusätzliche Kopie der Attribute -> Original und Kopie teilen sie nicht

Identität

- Objekte identisch: gleiche Referenz
- Objekte oberflächlich gleich: gleichen Zustände

- Objekte in der Tiefe gleich: rekursiv gleiche Zustände

Typisierung

- statisch: Typ jedes Ausdrucks zur Übersetzungszeit bekannt
- streng: keine Typfehler zur Laufzeit (Bsp: C++, Eiffel)

Vererbung

- Weitergabe von Attributen und Methoden von Ober- zu Unterklasse

Begriff	Haupteigenschaft
Spezialisierung	Integritätsbedingungen: Objektmenge der Unterklasse ist Teilmenge der Objektmenge in der Oberklasse
IST-Hierarchie	wie Spezialisierung
Typhierarchie	Gleiches Verhalten: jedes Objekt des Untertyps verhält sich wie eines des Obertyps Alle Attribute und Methoden des Obertyps sind auf Objekte des Untertyps anwendbar Substitution: jedes Objekt des Untertyps kann für bel. Objekt des Obertyps eingesetzt werden
Klassenhierarchie	Vererbung der Implementierung: Unterklasse wird mit Hilfe der Datenstrukturen für Attribute und Implementierung von Methoden aus der Oberklasse impl

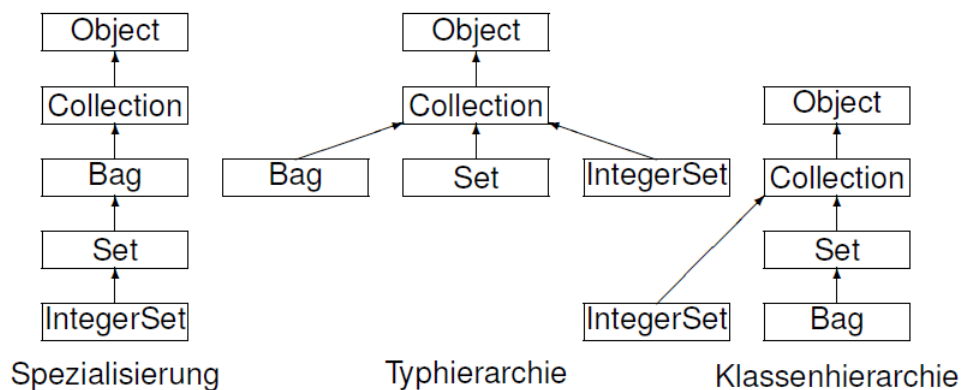


Abbildung 3: Vererbung Unterschiede am Bsp

- Mehrfachvererbung: Klasse darf mehrere Oberklassen haben
Probleme: Konflikte bei gleicher Methodensignatur -> Vermeidung oder Auflösung bspw. durch REDEFINE einer Methode

Overriding

- Oberklasse: Methode M, Implementierung MI1
- Normalfall der Vererbung: Unterklasse erbt Methode M, Implementierung MI1
- **Overriding:** Unterklasse erbt Methode M, ersetzt Implementierung durch MI2 => erfordert dynamisches Binden
- Varianten des Overriding:
 - *Ersetzung:* völlige Ersetzung von MI1 durch MI2
Bsp: Eiffel REDEFINE
 - *Verfeinerung:* MI1 wird von MI2 aufgerufen

Polymorphismus und dynamisches Binden

- Methode polymorph: kann auf Objekte unterschiedlicher Klasse angewandt werden
=> Wiederverwendbarkeit
Bsp: Addition -> unterschiedliche Impl je nach Datentyp
- dazu notwendig: dynamisches Binden:
Auflösung eines Methodenaufrufs (dessen Impl) zur Laufzeit anhand des Objekt-typs

Vergleich OOP und OO DB-Modelle

Zusätzlich in OODMS notwendig:

Eigenschaft	OOP	OODM
Attribute, Methoden, Typisierung	untypisiert oder wenig orthogonales Typkonzept; mengen oft durch generische Klassen simuliert	orthogonales Typkonzept zur Darstellung komplexer Werte
Einkapselung	Attribute sollen privat sein	Attribute und Datentypen wahlweise bekannt -> Unterstützung von Anfragen und Zugriffspfaden
Klassen	Implementierung eines ADT	Objektfabrik und -lager, das automatisch verwaltet wird
Konstruktoren & Destruktoren	Objekt wird erzeugt und "klebt" an seiner Klasse	Objekt wird erzeugt, kann aber zu mehreren Klassen gehören und diese auch wechseln
Vererbung	Klassenhierarchie = Vererbung der Impl	Klassenhierarchie = Spezialisierung Typhierarchie = Erweiterung der anwendbaren Attribute, Methoden

generische Operationen: sicher, optimierbar, deskriptiv; Definieren Relationen, dynamische Klassen; in OOPs simuliert: Methoden auf Mengen(=generische Klassen)
Transaktionskonzept, Flexible Speicherungsstruktur

2.2 Einschub

2.2.1 "Reines" OO programmieren mit Smalltalk

Eigl nur konkrete Umsetzung des vorherigem anhand von Smalltalk. Denke nicht, das dies prüfungsrelevant ist.

- Alle Elemente der Sprache sind Objekte => Kommunikation dazwischen nur Botschaften
- ausschließlich dynamisches Binden
- **Klassen**
 - Klasse ist Instanz ihrer Metaklasse
 - besteht aus: Instanzvariablen (Zustand jedes Objekts)
Klassenvariablen (Zustand der Klasse als Objekt)
Instanzmethoden (kann jedes Objekt der Klasse ausführen)
Klassenmethoden (kann die Klasse ausführen)
 - Klassenhierarchie: Baum mit Wurzel *Object*
- Smalltalk kennt nicht:
Typisierung
Mehrfachvererbung
Gesteuerte Vererbung
Öffentliche Attribute
Statisches Binden

2.2.2 Ist C++ streng typisiert?

streng typisiert = keine Typfehler zur Laufzeit

Problem (?) bei

- statisch erzeugten Objekten
- Zuweisung mit Wertsemantik
- Vererbung und Overriding

Fazit: Fehler nicht nachweisbar in neuen Versionen

2.3 Ein objektorientiertes Datenbankmodell: Überblick

Definiton 1

Ein OODBS ist ein System, das

- auf einem OODM basiert

- erweiterbar ist (zumin. konzeptuell)
- weitere DB-Eigenschaften besitzt,
 - Persistenz
 - Speicherungsstrukturen, Zugriffspfade
 - Transaktionen, Concurrency Control
 - Recovery
- neben generischen Operationen (etwa Anfragesprache) auch eine komplette Programmier-Umgebung beinhaltet.

Beispiel in O₂ - Methoden, Vererbung

```

CLASS Studenten INHERITS Personen
  TYPE TUPLE
    (Matrikelnummer: INTEGER,
      . . . . .)
  METHOD Zur_Verfuegung: REAL END

METHOD BODY Zur_Verfuegung: REAL IN CLASS Studenten
  { RETURN (SELF.Vater.Zur_Verfuegung +
    SELF.Mutter.Zur_Verfuegung) * 0.1;\}

```

2.4 Strukturteil eines OODM

Definition 2 - Strukturteil OODM

Der Strukturteil eines objektorientierten Datenbankmodells besteht aus

- Typen und Typkonstruktoren
- Objektidentität
- Klassen und Typen
- Beziehungen zwischen Klassen
- Klassen- und Typhierarchie
 - Strukturvererbung
 - Mehrfachvererbung
 - Konfliktauflösung bei Mehrfachvererbung
- Integritätsbedingungen

2.4.1 Typkonstruktoren

Definition 6 - Typen und Typkonstruktoren

An Typen stehen im OODM zur Verfügung:

- Standard-Datentypen *integer*, *string*, ..., die für das OODM elementar und vordefiniert sind
- ADT *DATE*, *TIME*, ..., die mit Hilfe von Typen und Typkonstruktoren gebildet wurden entweder vordefiniert oder benutzerdefiniert sind

sowie alle Typen, die mit den obigen Typen und

- Typkonstruktoren *TUPLE OF*, *SET OF*, *LIST OF*, *BAG OF*, *ARRAY OF*

definiert wurden. Die Typkonstruktoren sind dabei orthogonal anwendbar.

- Nach Beeri: SET OF, TUPLE OF orthogonal anwendbar (komplexe Werte)
- Geschachtelte Relationen (NF2-Relationen): SET OF TUPLE OF (Relationenkonstruktor)
- Komplexe Werte und geschachtelte Relationen äquivalent vom Informationsgehalt her

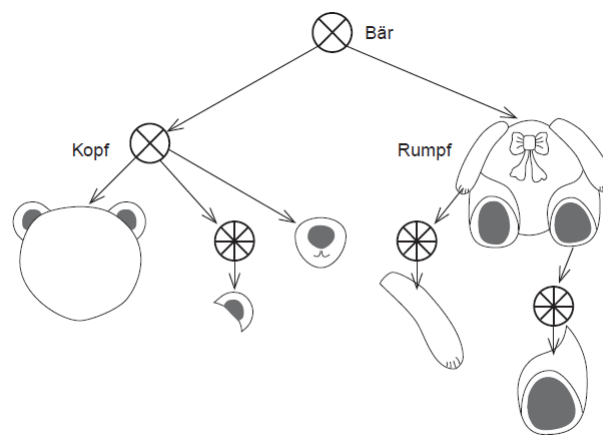
SET OF (TUPLE OF (Name: TUPLE OF (...)))

- Relationenmodell: nur SET OF TUPLE OF <Standard-Datentyp>
- OOP: nicht typisiert; nicht orthogonal; einige Konstruktoren nur simulierbar
- rekursive Typdefinitionen (Personen: SET OF TUPLE OF (...Freunde: Personen)) nicht erlaubt, besser durch Objektidentität auflösen (sonst endlos)
- Simulation der Typkonstruktoren in C++
 - statt Typkonstruktor: generische Klasse
 - Nachteile: nicht fest verdrahtet im System; kann redefiniert werden
Duplikateliminierung nicht automatisch mgl; Semantik einer Menge nicht bekannt
- Operationen
 - Tupelkonstruktor: Komponentenzugriff; Test auf (Un-)Gleichheit
 - Mengenkonstruktor: Zugriff auf ein Element: Iteratoren
Test auf ein Element
Vergleich von Mengen mit $=$, \neq , \subset , \subseteq , und deren Negationen
Mengenoperationen Vereinigung, Durchschnitt und Differenz

- Listenkonstruktor: Zugriff auf erstes (first), nächstes (next), letztes (last) Element
Teilliste erstellen ohne erstes Element (tail)
Iterator zum Durchlaufen der Liste in vorgegebener Reihenfolge
Konkatenation von Listen

- **Grenzen**

- Redundanzen bei nichthierarchischen Strukturen
- geschachtelte Relationen können redundanzfrei nur rein hierarchische Objektmengen darstellen
- darum Objektidentität notwendig



2.4.2 Objektidentität

Definition 3 - Objektidentität

Eine Objektidentität ist ein *abstrakter* Wert, der für jedes Objekt der Datenbank

- bei Erzeugen dieses Objektes vom System vergeben wird,
- systemweit eindeutig ist,
- unveränderbar ist,
- von außen nicht sichtbar ist.

- daher Beziehungen zw Objekten darstellbar, etwa eine gemeinsame Komponentenobjekte

- bei einigen System von außen sichtbar: dann gelöschte Objektidentitäten nicht wiederverwendbar

- **Unterscheidung Werte - Objekte**

Objekt	Wert
nicht druckbar	druckbar
anwendungsabhängige Abstraktion	anwendungsunabhängige Abstraktion
müssen erzeugt und definiert werden	müssen <i>nicht</i> erzeugt und definiert werden
trägt selbst keine Information	trägt Information
werden beschrieben	beschreiben etwas

Danach sind Werte Element von unstrukturierten (atomaren), konkreten Menge, den Domänen von Std-Typen etwa oder strukturierten Mengen, die mittels TUPLE OF, SET OF, LIST OF oder anderen Typkonstruktoren erzeugt werden (in denen dann neben Werten auch wieder Objekte vorkommen können)

- **Zuordnung Objekte - Zustände**

- Zustand eines Objekte o
 - komplexe Werte
 - andere (Komponenten-)Objekte (o heißt dann zusammengesetztes Objekt)
- Darstellung von Objekten mit Zustand
 - Objekt (kleiner Kreis), Zuordnung des Zustands (Pfeil), Zustand (Oval)
 - geschachtelte Relationen mit spezieller Spalte für Objektidentitäten (Objektrelationen)

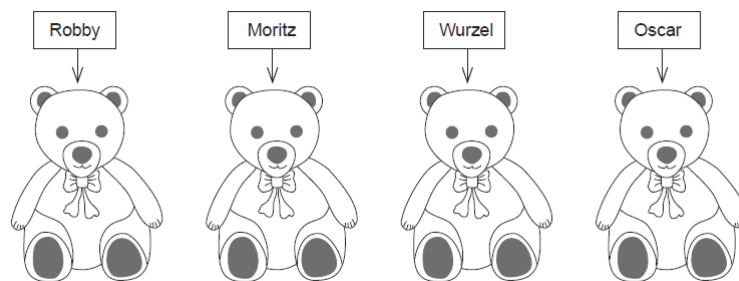
- **Unterschiede, Einordnungen**

- Relationenmodell
 - * Objekte nur über sichtbare Schlüssel und Fremdschlüssel zu identifizieren
 - * veränderbar
 - * nur relationenweit eindeutig
 - * vom Nutzer vergeben
- OOPLs
 - * Objektidentität meist physischer Zeiger, der aber veränderbar ist
 - * ein Objekt kann nicht in mehreren Klassen mit der gleichen Identität auftauchen

- **Realisierungen**

- Abstrakte Objekte
 - * Elemente einer globalen Menge abstrakter Objekte

- * Elemente verschiedener, disjunkter, abstrakter Domänen
- Surrogat-Attribute
 - * beste Implementierung von abstrakten Objekten
 - * als konzeptuelle Objektidentität mit Vorsicht zu behandeln (Sichtbarkeit, Änderbarkeit, funktionale Beziehungen zu Zuständen)
- Namen
 - * zur Zusatz-Identifikation einiger Objekte geeignet
 - * problematischer Test auf Identität (mehrere Namen für ein Objekt)
- direkte oder indirekte Referenzen
 - * nur als Implementierungshilfsmittel geeignet
 - * indirekte Referenzen sind flexibler (Verschiebbarkeit von Objekten)



2.4.3 Klassen und Typen

Definition 4 - Klassen, Zustände, Zustandstypen

eine Klasse besteht aus

- einer abstrakten Domäne (der Wertevorrat der Objektidentitäten)
- einer Extension (auch: Instanz), also der aktuellen Objektmenge = Menge bislang erzeugter und noch nicht gelöschter Objekte (= Persistenz)
- einem zugeordneten Zustandstyp (mit Typen und Typkonstruktoren)
- einer Zuordnung von Zuständen zu Objekten

- ein Objekt kann in mehreren Klassen vorkommen (mehrere Rollen spielen: Person, Student, Angestellter,...)
- spielt dann mehrere Rollen durch Ober- und Unterklassen

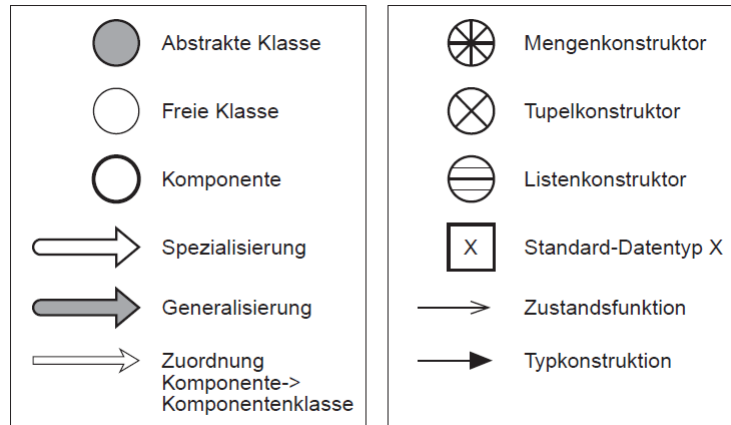


Abbildung 4: Grafische Symbole für OODM

- **Beispiel**

Der Klasse *Bücher* ordnen wir die folgenden Informationen zu:

- die abstrakte Domäne $\{\alpha_1, \alpha_2, \dots\}$, *Bücher* ist abstrakte Klasse (s.u.)
- die Extension (aktuelle Objektmenge), zunächst leer, nach zehnmaliger Anwendung der Erzeugungsfunktion CREATE etwa $\alpha_1, \alpha_2, \dots, \alpha_{10}$
- den Zustandstyp

```

TUPLE OF (
    ISBN: STRING,
    Titel: STRING,
    Verlag: Verlage,
    Autoren: LIST OF (Autor: STRING),
    Stichworte: SET OF (Stichwort: STRING),
    Versionen: SET OF (TUPLE OF (Auflage: integer,
)

```

der jedem Buch α ein Tupel zuordnet, das unter anderem wieder ein Objekt der Klasse *Verlag* beinhaltet.

- **Unterschiede und Einordnungen**

- Relationenmodell: Relation sammelt Werte, keine Objekte
- OOPs: keine Instanz, wird meist explizit in Variable vom SET OF- Typ gesammelt
Objekte nur in einer Klasse
kann keine unterschiedlichen Rollen spielen

- zwei Arten von Klassen

Abstrakte Klasse: wird eine abstrakte Domäne zugeordnet; hier gibt es Kon-

strukturen, hier werden Objekte in der DB erzeugt

Freie Klasse: wird *keine* abstrakte Domäne zugeordnet, erhält Domäne durch Vererbung, hier gibt es keine Konstruktoren, hier werden schon in der DB bestehende Objekte neu aufgenommen

2.4.4 Beziehungen zwischen Klassen

Definition 5 - Beziehungen

Eine Klasse kann in Beziehung zu anderen Klassen, ein Objekt in Beziehung zu anderen Objekten stehen. Hat eine Klasse K_1 eine Komponentenklasse K_2 , so nennt man die Objekte in K_1 zusammengesetzte Objekte mit den zugehörigen Komponentenobjekten aus K_2 . Komponentenklassen können folgende Eigenschaften haben:

- *gemeinsam (shared)* oder *privat*
 - gemeinsam: ein Komponentenobjekt in vielen zusammengesetzten Objekten
 - privat: ein Komponentenobjekt in maximal einem zusammengesetzten Objekt (ACHTUNG: nicht OOPL Einkapselung privat)
 - Bsp: Verlage gemeinsam in Bücher, Motor privat in Autos
- *abhängig* oder *unabhängig*
 - abhängig: Komponentenobjekt wird gelöscht, wenn (letztes) zugehöriges zusammengesetztes Objekt gelöscht wird
 - unabhängig: Komponentenobjekt bleibt auch in diesem Fall bestehen
 - Bsp: Entleiher unabhängig von Entleihungen; Eltern abhängig in Studenten
- *eingekapselt* oder *nicht eingekapselt*
 - eingekapselt: Zugriff auf Komponentenobjekt nur über zusammengesetztes Objekt
 - nicht eingekapselt: Zugriff auf Komponentenobjekt auch direkt mgl
 - Bsp: Kleinteile eingekapselt in Fahrzeuge

- **Operationen**

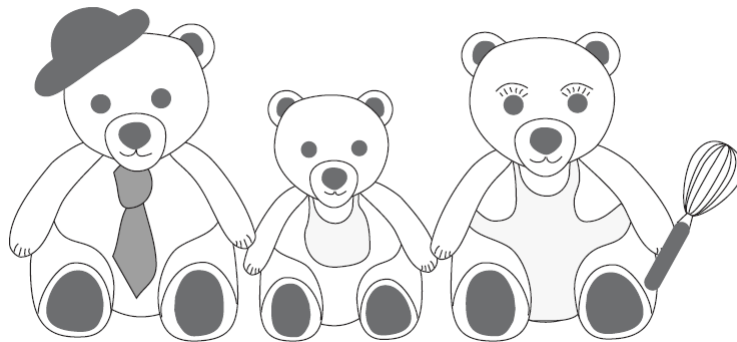
- Zugriff auf Komponentenklassen/Komponentenobjekte mit dot-Operator in Pfadausdrücken
- für diese kann man evtl. auch Invertierung anwenden

- **Unterschiede**

- statt asymmetrischer Beziehung von zusammengesetztem Objekt zu Komponentenobjekt auch symmetrisch mgl:
- Relationships wie im ER-Modell: 1:1, 1:n, n:m

- **Einordnung**

- Relationenmodell: alle Relationships mgl; Komponentenklassen nur simuliert (privat, unabhängig)
- OOPs: 1:n-Relationships durch Komponentenklassen; Komponentenklassen meist mit fixierter Semantik



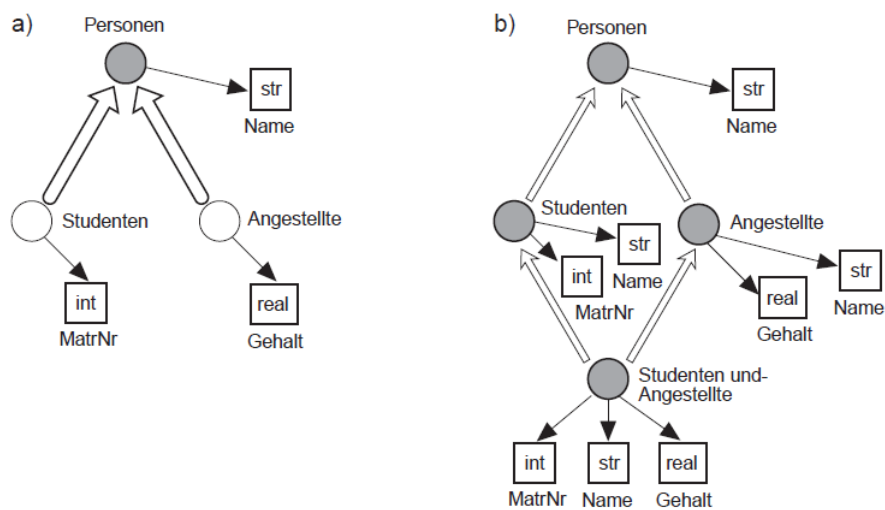
2.4.5 Strukturvererbung

Definition 6 - Strukturvererbung: Klassen- und Typhierarchie

K_1 Unterklasse von K_2 , wenn Extension zu K_1 Teilmenge der Extension zu K_2

T_1 Untertyp von T_2 , wenn T_1 mehr Komponenten hat als T_2 (Definition vereinfacht für T Tupeltyp)

- nach DB-Entwurf: beide Hierarchien parallel
- nach Anfragen: Hierarchien müssen nicht mehr übereinstimmen
- **Einordnung:**
 - OOPs: K_1 Unterklasse von K_2 wenn K_1 die Methoden von K_2 erbt (nutzt die Impl. von K_2)
 - etwa mgl: BAG Unterklasse von SET, da BAG Impl von SET nutzt, konzeptuell ist SET Unterklasse von BAG



a): Sicht Klassenhierarchie; b): Sicht Typhierarchie

Begriffe	Bedeutung in OOPs	Bedeutung in OODMs
Klassenhierarchie	Vererbung der Implementierung	Integritätsbedingungen
Typhierarchie	Gleiches Verhalten, mehr anwendbare Attribute	Gleiches Verhalten, mehr anwendbare Attribute
IST-Hierarchie	Integritätsbedingung	Integritätsbedingung und gleiches Verhalten
Spezialisierung	wie IST-Hierarchie	Festlegung der Domäne von Oberklassen
Generalisierung	invers zu Spezialisierung	Festlegung der Domäne von Oberklassen
allgemein	ohne Wertvererbung	mit Wertvererbung

- **OODB-Begriffe**

- **Spezialisierung und Generalisierung**

in OODB: beides spezieller Arten der Klassenhierarchie

- Spezialisierung
 - * Oberklasse (abstrakt oder frei) gegeben
 - * Unterklassen sind Teilmengen (frei)
- Generalisierung
 - * Unterklassen (abstrakt oder frei) gegeben
 - * Oberklasse ist Vereinigung (frei)
- Beispiele

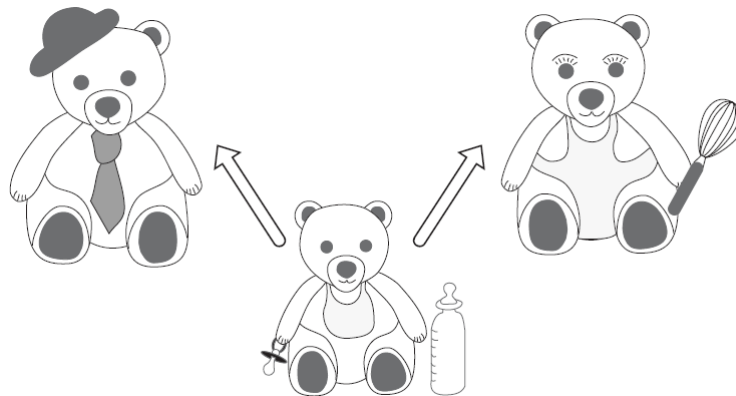
- * Spezialisierung von Personen (abstrakt) zu Studenten (frei)
- * Spezialisierung von Studenten (frei) zu Hilfsassistenten (frei)
- * Generalisierung von Studenten und Angestellte (beide frei) zu Entleiher (frei)
- * Generalisierung von Angestellte (frei) und Geräte (abstrakt) zu Haus-haltspositionen (frei)

- **Problem: Mehrfachvererbung**

Lösung wie in OOPLs

- **Flache und tiefe Extension**

- flach: alle Extensionen sind disjunkt (in OOPLs üblich)
- tief: simuliert Mehrfachzugehörigkeit von Objekten zu Klassen
- aber kein allgemeines Inklusionsprinzip (da disjunkter Durchschnitt)
- Objekte werden immer in speziellster Klasse erzeugt



2.4.6 Integritätsbedingungen

Definition 7 - Integritätsbedingungen

Schlüssel

- vererbte Schlüssel (bei Personen definieren, bei Unterklassen nicht nötig)
- komplexe Schlüssel (Titel und Menge von Autoren bei Büchern)
- Schlüssel von Komponenten (Titel von Buch und Name von Verlag)
- andere Identifizierungsmechanismen (Klassenzugehörigkeit)

Kardinalitäten

- Nullwerte oder nicht
- Beziehungen 1:1, 1:n, m:n, meist asymmetrisch simuliert: Klassen und Komponentenklassen
- Mengenkardinalitäten (student darf max 10 Bücher ausleihen, muss min drei VL hören); Vorsicht: unentscheidbar

Integritätsbedingungen an die Strukturvererbung

- Überdeckungsbedingung: außer Angestellten, Studenten gibt es keine weiteren Personen
- Disjunktheitsbedingung: Angestellte, Studenten sind disjunkt

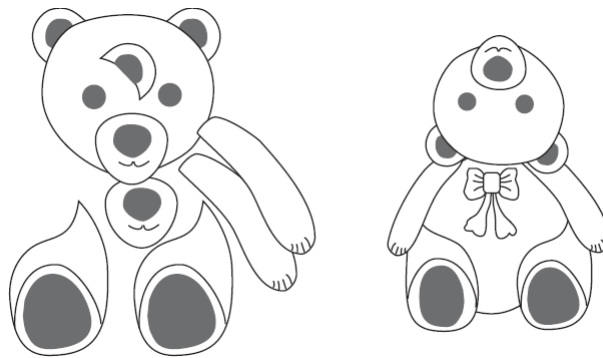


Abbildung 5: Auswirkung fehlender Integritätsbedingungen

2.5 Operationenteil eines objektorientierten Datenbankmodells

- min die Möglichkeiten wie in SQL
- relationale Semantik: man extrahiert Werte aus Zuständen von Objekten
Ergebnis ist geschachtelte Relation
- objekterzeugende Semantik: man erzeugt neue Objekte als Anfrageergebnis mit Zuständen, die von vorhandenen Objekten extrahiert wurden
Ergebnis ist eine dynamisch erzeugte Klasse
- objekterhaltende Semantik: man erhält eine Auswahl der in der DB vorkommenden Objekte mit neuen Zuständen
Ergebnis ist dynamisch erzeugte Ober-/Unterklasse

- **Einordnung, Unterschiede**

- Relationenmodell: generische Anfragen und Updates auf flachen Relationen
- OODBs: Standard-Methoden auf COLLECTION-Klassen (Selektionen mit sehr einfachen Selektionsprädikaten)
OSQL mit relationaler Semantik (nicht so mächtig wie Std-SQL)

- Taxonomie generischer Operationen

Operationen in rot: angelehnt an Relationenalgebra

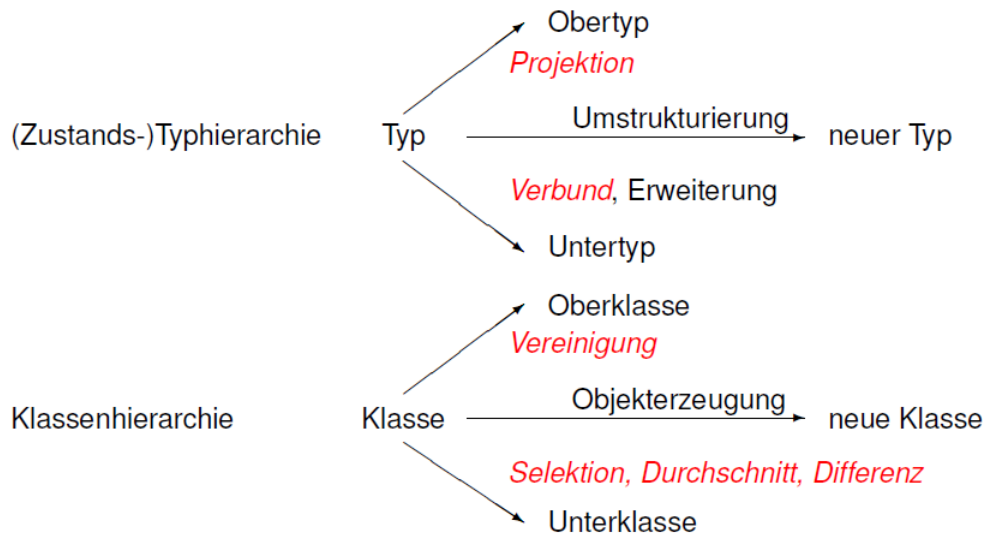


Abbildung 6: Taxonomie generischer Operationen

- **Relationale Operationen**

- Relationenalgebra
- Minimale geschachtelte Algebra (auf geschachtelten Relationen)
- orthogonal geschachtelte Algebra
- PNF-Algebra (auf geschachtelten Relationen in PNF = Partitioned Normal Form): bewahren Schlüssel, können also auch Objektidentitäten (und damit Objektrelationen) bewahren

- **Anfragen:** klassenbasiert oder extensionsbasiert

- **Klassenbasierte Anfragen**

- * bei objekterhaltender Semantik: man erhält eine Auswahl der in der DB vorkommenden Objekte mit neuen Zuständen
- * Ergebnis ist dynamisch erzeugte Ober-/Unterklasse

- **Extensionsbasierte Anfragen**
 - * bei objekterhaltender Semantik: man erhält eine Auswahl der in der DB vorkommenden Objekte mit neuen Zuständen
 - * Ergebnis ist eine neue Extension einer bereits bestehenden Klassen
- Beispiel:
 - * Selektion auf Klasse Studenten nach Studiengang 'Informatik'
 - * Klassenbasiert: Unterklasse Informatiker von Klasse Studenten
 - * Extensionsbasiert: Neue Extension Informatiker zur existierenden Klasse Studenten

2.6 Höhere Konzepte eines objektorientierten Datenbankmodells

- höhere Konzepte: formal nur in Prädikatenlogik höherer Ordnung beschreibbar (Struktur- und Operationenteil in 1. Ordnung)
- Methoden:
 - Schnittstellen, Impl, Einkapselung, Vererbung, Overriding, Mehrfachvererbung, Konfliktauflösung
- Metaklassen
- **Methoden**
 - Anfrage- und Update-Methoden
 - Anfragen liefern neues (abgeleitetes Attribut) bel. Typs
 - Updates liefern Fehlercode, Seiteneffekt: Änderung des Zustands des aktuellen Objekts
 - Schnittstelle: Ein- und Ausgabeparameter, ihre Typen
 - Impl: meist in OOPL (eingekapselt)
 - Folgende Konzepte wie in OOPLs
 - Vererbung, Overriding, Einkapselung, Mehrfachvererbung, Konfliktauflösung

- Varianten der Einkapselung

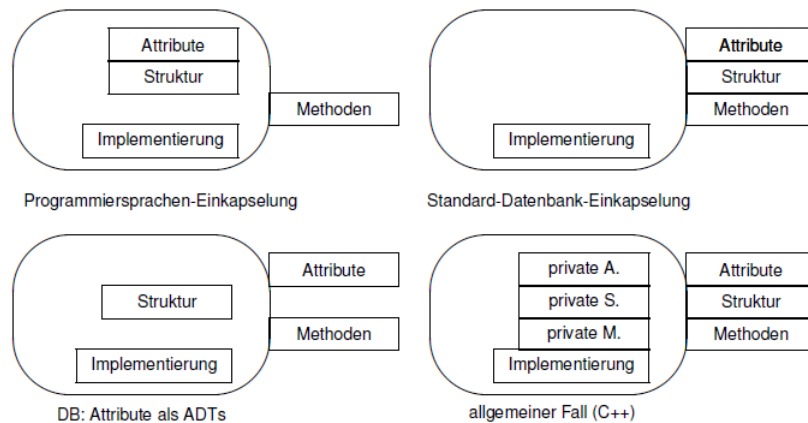


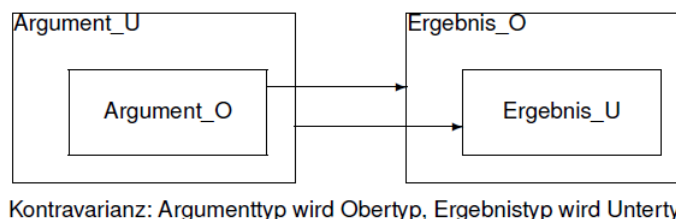
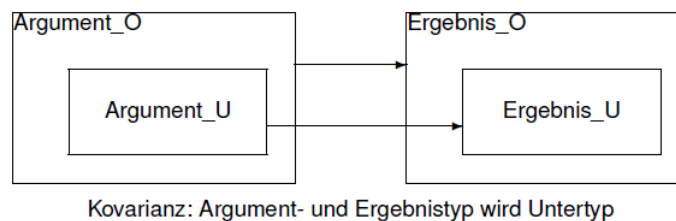
Abbildung 7: Varianten der Einkapselung

- **Overriding von Schnittstellen**

bisher: Ersetzen von Impl.

jetzt auch: kontrolliertes Ersetzen von Schnittstellen

Notation: O - Methode der Oberklasse; U - Methode der Unterklasse



- **Ko- und Kontravarianz**

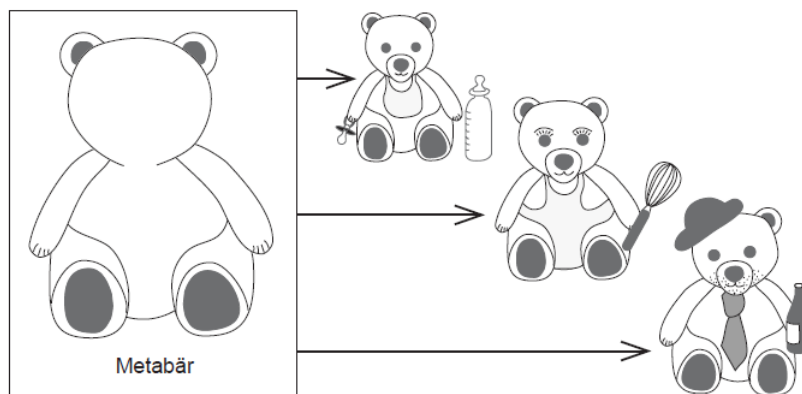
- Kovarianz (Eiffel)

Argument(typ) + Ergebnis(typ) wird jeweils Untertyp
sinnvoll, aber nicht typsicher

- Kontravarianz
Argument(typ) wird Obertyp, Ergebnis(typ) wird Untertyp
nicht sinnvoll, aber typsicher
- No-Varianz (C++)
Argument(typ) und Ergebnis(typ) bleiben unverändert

- **Metaklassen**

- Klassen werde als Objekte (Instanzen) einer höheren Klasse (Metaklasse) aufgefasst
- dem Objekt (der Klasse) können dann Zustände zugewiesen werden, auf dem Objekt (auf der Klasse) können Methoden ausgeführt werden
- in OOPs:
Klassenattribute statt Instanzattribute
zb C++ statische Var mit static
- Anwendung: setzen Defaultwerte
- Anwendung: Methodendefinition
höhere Konzepte = Beschreibbar in Prädikatenlogik > 1. Ordnung



- **Instanzbeziehungen**

- neben Klasse-Unterklasse Beziehung (IST, INHERIT, Strukturvererbung, Klassenhierarchie) und
- Klasse-Komponentenklasse-Beziehung (IS_PART_OF) auch
- IS_INSTANCE_OF: Klasse-Instanz-Beziehung oder kurz Instanzbeziehung

- **Einige formale Definitionen**

Definition 8 - Typen, Typenkonstruktoren

Ein Typ ist ein Standard-Datentyp T , dem eine unstrukturierte Menge $dom(T)$ zugeordnet wird, oder ein konstruierter Typ:

- $T = \text{tuple of}(A_1 : T_1, \dots, A_n : T_n)$, wobei T_1, \dots, T_n wiederum Typen sind und $dom(T) = dom(T_1) \times \dots \times dom(T_n)$ gilt,
- $T = \text{set of}(T_1)$ (oder auch $T = \text{set of}(A_1 : T_1)$), wobei T_1 wiederum ein Typ ist und $dom(T) = \rho(dom(T_1))$ gilt,
- $T = \text{list of}(T_1)$ (oder auch $T = \text{list of}(A_1 : T_1)$), wobei T_1 wiederum ein Typ ist und $dom(T) = T_1^*$ gilt.

Definition 9 - abstrakte Domäne und Objektidentitäten

Sei \mathbb{D}_A eine Menge disjunkter, unendlicher Mengen D_A . Dann nennen wir jedes D_A eine abstrakte Domäne. Jedes Element von D_A ist eine Objektidentität (oder ein (abstraktes) Objekt).

Definition 10 - Zustandstyp und Zustand

Jedem (abstrakten) Objekt (oder jeder Objektidentität) wird ein komplexer Typ T als Zustandstyp funktional zugeordnet. Der Zustand eines Objektes ist dann eine Instanz seines Zustandstyps.

Definition 11 - Klasse, Abstrakte Klasse, Extension

Ein gegebener Anwendungs-Objekttyp wird durch eine Klasse K aus der Menge aller Klassen \mathbb{K} repräsentiert. Jeder Klasse wird eine Domäne, eine Extension, ein Zustandstyp und eine Zustandsfunktion zugeordnet. Wird K eine abstrakte Domäne als Domäne zugeordnet, so bezeichnen wir K als abstrakte Klasse. Die Zuordnung geschieht über die Funktion dom mittels $dom : \mathbb{K} \rightarrow \mathbb{D}_A$. Die aktuelle Extension einer Klasse $o(K)$ ist eine Teilmenge von $dom(K)$. Wird K keine abstrakte Domäne zugeordnet, so heißt K freie Klasse.

Definition 12 - Zustandstyp einer Klasse, Zustandsfunktion

Jeder Klasse K wird ein Zustandstyp T_K funktional zugeordnet, der ein Standard-Datentyp, wiederum eine Klasse oder ein komplexer Typ ist. Ist im Zustandstyp eine Klasse enthalten, so wird diese Klasse Komponentenklasse von K genannt. Jedem Objekt α aus der Extension $o(K)$

wird mittels der Zustandsfunktion ZUSTAND ein Element w von T_K zugeordnet. Dabei ist das Element von T_K aus der Domäne des Typs bei Standard- und komplexen Typen und aus der Extension der Klasse bei Komponentenklassen. w wird Zustand von α genannt.

Definition 13

Die Menge aller Klassen \mathbb{K} sei partitioniert in die Menge aller abstrakten Klassen \mathbb{K}_A und die Menge aller freien Klassen \mathbb{K}_F . Die Menge aller Spezialisierungen ist eine binäre Relation *spec* über Klassen. Für jedes Element $K_1 spec K_2$ gilt, dass K_1 eine freie Klasse sein muss.

Die Menge aller Generalisierungen ist eine binäre Relation *gen* über Klassen. Für jedes Element $K_1 gen K_2$ gilt, dass K_2 eine freie Klasse sein muss. Die reflexive und transitive Hülle von $spec \cup gen$ wird mit \leq bezeichnet und Klassenhierarchie genannt. Es wird zusätzlich gefordert, dass \leq eine partielle Ordnung auf Klassen ist. Für jede freie Klasse K definieren wir die Domäne durch

$$dom(K) := \cap_{(K, K_i) \in spec} o(K_i) \text{ oder } dom(K) := \cup_{(K_i, K) \in gen} o(K_i)$$

wobei $o(K_i)$ die Extension der Klasse K_i ist. Im zweiten Fall wird oft die Überdeckungsbedingung, also $o(K) = \cup_{(K_i, K) \in gen} o(K_i)$ gefordert.

Man beachte, dass in der letzten Definition jeder Klasse genau eine wohldefinierte Domäne zugeordnet wird, falls folgende Zusatzeinschränkungen getroffen werden (die in den Modellen IFO und EXTREM vorhanden sind):

- Jede freie Klasse taucht wenigstens einmal entweder auf der linken Seite eines *spec*-Tupels oder auf der rechten Seite eines *gen*-Tupels auf.
- Jeder Pfad aus *spec*-Tupeln, der in einer bestimmten Klasse K startet, endet in derselben Klasse K' .
- Die binäre Relation $spec \cup gen^{-1}$ ergibt einen gerichteten, azyklischen Graphen. Man beachte, dass zur Kontrolle der Azyklizität die Richtung der *gen*-Tupel umgedreht werden muss. Andreas

Die entsprechende Typhierarchie kann nun aus der Klassenhierarchie abgeleitet werden: alle in Oberklassen spezifizierten Attribute sind implizit auch für die Unterklassen definiert. Zunächst gehen wir davon aus, dass die jeweiligen Attributmengen disjunkt sind.

Definition 14 - Vervollständigter Zustandstyp, Typhierarchie

Sei $K \leq K_1, K \leq K_2, \dots, K \leq K_p$ gegeben, dabei seien K_1, K_2, \dots, K_p alle direkten Oberklassen von K . Dann ist der vervollständigte Zustandstyp von K der Tupeltyp, der durch Konkatenation der vervollständigten Zustandstypen von K_1, K_2, \dots, K_p entsteht. Dann gilt: Ist $K \leq K'$, dann ist der vervollständigte Zustandstyp von K Untertyp vom vervollständigten Zustandstyp von K' . Die Zustandstypen stehen dann bezüglich \leq in einer Typhierarchie zueinander.

2.7 Klassifikation objektorientierter Datenbanksysteme

- Entwicklungsrichtungen
 - OO Datenbankprogrammiersprachen (OODBPLs)
 - Objektrelationale Datenbanksysteme (ORDBMSs)
 - Neuentwicklungen
- Andere Einteilung
 - OO (ODGMSs): OODBPLs; Neuentwicklungen (native OODBMSs)
 - Objektrelational (ORDBMSs): erweitert relational (nur Typkonstruktoren und Objektidentität); voll objektrelational; offen, Wrapper oder Mapper
- Entwicklungen und Herausforderungen
 - OOPs \rightarrow OODBs: OOP erweitern um
 - Strukturteil: Extension, Persistenz, Typen,...
 - Operationenteil
 - Speicherungsstrukturen, Zugriffspfade, Transaktionen, Concurrency Control
 - RDBs \rightarrow OODBs: relationales DBS erweitern um
 - Strukturteil (Typkonstruktoren, Objektidentität, Klassen, Klassen-/Typhierarchie)
 - Methoden, Vererbung, Overriding
 - Völlige Neuentwicklungen
 - nicht OOP oder relationales Datenmodell als Basis; eigenes OODM
- OODBPLs
 - Basis: C++, Smalltalk, CLOS, Java
 - Standard: ODMG-Bindings
 - Bsp: ObjectStore
 - setzt Persistenzkonzept von Atkinson um (siehe Kap 5)
 - Effizienz bei Zugriffen auf komplexe Objekte
 - Nachteile: kein Operationenteil, kein Sichtkonzept
 - nur Strukturteil (eingeschränkte Implementierungshierarchie) und Verhaltensteil verwirklicht

dadurch starke Einschränkungen in Anwendungsmodellierung
keinen Ebenentrennung: Speicherstrukturen, Sperren sichtbar

- **ORDBMS**

- Basis: Relationenmodell
- Standard: SQL:1999,..., SQL:2011
- Objektrelationale DBS
 - Relationen mit Klassen, Methoden, Klassen- und Typhierarchie
 - Bsp: POSTGRES, alle großen RDBMS wie Oracle, DB2

- Wann ist ein System objektrelational?

umstrittene Quadrantenqualifikation nach Stonebreaker besser: nach Standard SQL:1999

	einfache Daten	komplexe Daten
Anfragen	relationale DB	Objektrelationale DB
keine Anfragen	Dateisysteme	Objektorientierte DB

oder SQL:2003

Kriterien		objektrelationale DBS
Bestandteile	Konzepte	
Strukturteil	Typkonstruktoren	✓
	Objektidentität	✓
	Klassen	(✓)
	Beziehungen	(✓)
	Strukturvererbung	✓
	Integritätsbeding.	✓
Operationenteil	generische	✓
	extensionsbasiert	✓
	klassenbasiert	
Höhere Konzepte	Metaklassen	
	Methoden	✓
	Vererbung	✓
	Overriding	(✓)
	Einkapselung	(✓)

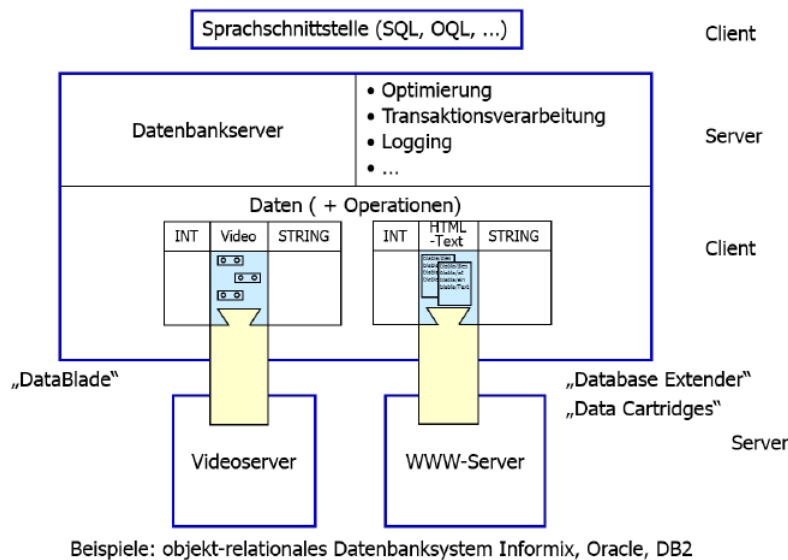
- **Objektrelationale Strukturen**

- relationales Datenmodell
- Drei-Ebenen-Architektur: volle Datenunabhängigkeit und Flexibilität mgl
- als Typen jedoch ADTs mit Typhierarchie, Methoden, evtl Overriding
- Objektidentitäten, Klassen oder Relationen (Tabellen)
- Klassen- oder Relationenhierarchien (Tabellenhierarchien)
- jedoch grundlegender Datentyp (auch für Anfragen): Relationen

drei Architekturen objektrelational

- OO-Schnittstelle auf RDBS: Wrapper (zb hibernate)

- RDBS mit internen ADT-Erweiterungen: halboffen (DB2, Oracle)
- RDBS mit externen ADT-Erweiterungen: offenes ORDBMS (Informix)



• Nachteile objektrelationaler Systeme

- weiterhin Impedance Mismatch zu OOPL-Umgebungen
- oft Etikettenschwindel (nur DBS mit ADTs): noch große Unterschiede zw geplantem Standard SQL4, verabschiedetem Standard SQL:2003 und realen ORDBMS
- Anfragen/Sichtkonzept unterstützen relationale, aber nicht die objektorientierten Anteile im Modell
- Persistenzprinzip eingeschränkt: nur Tupel in Relationen persistent

• Neuentwicklungen

- Basis: eigenes OODM
nicht: Relationen mit Objekttypen und Objektidentität
nicht OOPL-Klassen wie C++ und Smalltalk
- O_2
zunächst OODM wie in VL
danach Klassenhierarchie durch Typhierarchie ersetzt
danach Verzicht auf Extensionen
aber: Persistenz durch Namenskonzept, Erreichbarkeit
- **ITASCA**
- **OSCAR**
- keine dieser Entwicklung hat überlebt (Stand 2015)

- Wie sieht es aktuell aus?
Relationale DBSs → OODBs? PostgreSQL, IBM DB2 ab V2, Oracle ab V8
aktuelle Richtungen: NoSQL, Postrelationale DBS, Dokumentorientierte DBS

3 Der ODMG-Standard

3.1 Modelle für OODBPL

- Attribute und Methoden, Typisierung
OOPLs: untypisiert oder wenig orthogonales Typkonzept, Mengen oft durch generische Klassen simuliert
- Einkapselung
OOPLs: Attribute sollen privat sein
- Klassen
OOPLs Klasse ist Implementierung eines ADT
- Konstruktoren und Destruktoren
OOPLs: Ein Objekt wird erzeugt und *lebt* an seiner Klasse
- Vererbung
OOPLs: Klassenhierarchie = Vererbung der Implementierung

3.2 der ODMG-Standard

- ODMG = Object Database Management Group
- Struktur: Objektmodell, ODL und OQL, Sprachanbindung (C++, Java)
- seit 2006 ist V4 geplant, 2014 Arbeit eingestellt

Produkt	ODL	OQL	C++	Smalltalk
GemStone	(< √ >)			√
MICRAM	√		(√)	
O ₂		√	√	√
Objectivity	(√)	(√)	(√)	(√)
ObjectStore			√	√
ODBMS				< √ >
Omniscience	< √ >	(√)	< √ >	
POET		(< √ >)	√	
UniSQL		√	< √ >	< √ >
Versant			√	

- **Grundkonzepte**
 - Objekte: Zustand direkt Bestandteil des Objektes (atomare und strukturierte Objekte); früher mutable objects
 - Literale: Werte (atomar oder strukturiert), früher immutable objects
 - Eigenschaften von Objekten: Attribute und Beziehungen
Attribute: Werte oder Objekte
Beziehungen: Objekte

Beziehungen vs. objektwertige Attribute: Beziehungen immer mit inversen Referenzen, bei objektwertigen Attributen nicht

- Verhalten von Objekten: wird mit Operationen beschrieben (Methoden(-Schnittstellen))
- Typen: sammeln Objekte mit gemeinsamen Eigenschaften und gemeinsamen Verhalten
- Unterscheidung Schnittstelle und Implementierung
pro Schnittstelle diverse Impl.; nicht nur bei Methoden, sondern auch im Strukturteil
- Typ durch Schnittstelle und mehrere Impl beschrieben
- Klasse ist eine konkrete Impl. dieses Typs
Impl. des Strukturteils (Repräsentation der Attribute durch Datenstrukturen oder Methoden)
Impl. des Verhaltensteils durch eine Menge von Methoden
- Def. von Schnittstellen und Impl
Für Schnittstellendef. ODL oder PL-ODL
Def. der Impl. sprachabhängig (C++, Java,...)
- Typhierarchie: Strukturvererbung und Vererbung von Methoden (in ODMG: Operationen; Substituierbarkeitsprinzip), Overriding
- Implementierungshierarchie: auf Klassen auch Impl.hierarchie (Wortsymbol extends); keine Mehrfachvererbung
- Instanzen: Instanz eines Typs ist (erzeugtes) Objekt von diesem Typ; virtuelle Typen (abstrakte Typen, abstract types in ODMG) haben keine Instanzen
- Extension: aktuelle Objektmenge einer Klasse; definierbar für jede Klasse; alle Instanzen des Typs werden Elemente der Extension
- tiefe Extension: Extension einer Unterklasse Teilmenge der Extension der Oberklasse (aber kein Rollenkonzept)
- Schlüssel: gilt für Extension der Klasse

3.3 Der Strukturteil und höhere Konzepte des ODMG-Standards

- Objekte haben Objektidentität und evtl Namen
- Lebensdauer eines Objektes zur Erzeugungszeit festgelegt
- **Objektidentität**
Test auf Identität: Operation *same_as*

Erzeugung von Objekten durch Konstruktor-Operation (vordef. Schnittstelle: factory interfaces)

persistente und transiente Objekte können zu einem Typ gehören und können auch mit den gleichen Operationen manipuliert werden

- **Namen**

Objektidentität nach außen nicht sichtbar

falls keine Extension: benannte Objekte sind DB-Einstiegspunkte

- **Collection-wertige Objekte**

- OODM: Objekte immer unstrukturiert (atomar)
- ODMG: die beiden Bestandteile zusammengezogen (nicht bei tupelwertigen Zuständen: atomare Objekte mit einem tupelwertigen Zustandstyp)
- ODMG-Objektmodel nähert sich den OOPL-Modellen wie C++, Java an
Set<T>, Bag<T>, ... T bestimmt Elementtyp der Collection
- vordef. Operationen
Erzeugung: new_of_size (in long size)
über die collection factory interfaces, initiale Größe durch size
Tests: cardinality, is_empty, contains_element
Update-Operationen: insert_element, remove_element
Iteratoren: create_iterator,..
....

- **Werte oder Literale**

- atomare Typen
(unsigned) long/short, float, double, boolean, char, string, enum
- komplexe Typen (collection-wertig)
set<t>, bag<t>, list<t>, array<t>, dictionary<t,w>
- komplexe Typen (tupelwertig)
vordef. Tupeltypen date, time, interval, timestamp
Typ struct mit Operationen set_element,..

- **Schnittstellen**

def. mit interface Klausel der ODL

Attribute

- können mit Werten oder Methoden realisiert (gute Einkapselung)
- Bsp:

```
interface PERSON_S
{
    attribute long PANr;
    ...
    attribute date Geburtsdatum;
```

```

        attribute unsigned short Alter;
    }

```

- Impl. von Alter ist noch völlig frei
- Attribut: unidirektionale Referenz zu Komponententyp

Beziehung

- bidirektionale Referenz zu Komponententyp (Konsistenzchecks)
- Beispiele

```

interface Student_S : Person_S {
    attribute long Matrnr;
    ...
    attribute Person_S Mutter;
    relationship Angestellter_S Betreuer
        inverse Angestellter_S::Betreut;}

```

korrespondierende Klausel des Betreuers in Typ Angestellter_S

```

interface Angestellter_S : Person_S {
    attribute long Angnr;
    ...
    relationship Set<Student_S> Betreut
        inverse Student_S::Betreuer}

```

- ist 1:n Beziehung; 1:1 und n:m über Typen steuern

Operationen

durch Signatur (Namen der Operation, name und Typ der Argumente und Ergebnis, Namen von Ausnahmen)

Vererbung, Overriding (Overloading genannt), dynamisches Binden

Konzept in OODM	Konzept in ODMG
Wert Objekt Objekt mit <code>Collection</code> als Zustandstyp Tupelkonstruktor Mengenkonstruktor Listenkonstruktor ...	Literal atomares Objekt <code>Collection</code> -wertiges Objekt Wertetyp <code>struct</code> Wertetyp <code>set</code> Objekttyp <code>Set</code> Wertetyp <code>list</code> Objekttyp <code>List</code> ...
Klasse Implementierung eines Typs Instanz Schlüssel virtuelle Klasse vordefinierter ADT	Objekttyp und Extension Klasse Extension Schlüssel abstrakter Typ vordefinierter <code>struct</code> -Typ
Attribut Anfrage-Methode Klasse—Komponentenklasse-Beziehung diese auch mit inverser Beziehung	Attribut Abgeleitetes Attribut oder Operation Objektwertiges Attribut Beziehung
Klassenhierarchie Typhierarchie Implementierungshierarchie	nicht verwirklicht Typhierarchie <code>extends</code> -Hierarchie auf Klassen
Methode Methodenimplementierung Overriding	Operation Methode Overloading

Abbildung 8: Zusammenfassung Vergleich

3.4 Die ODL des ODMG-Standards

- definiert Schnittstellen, Klassen

```
interface <Name> : <Obertypen>
    {<Schnittstellenspezifikation>}
class <Name> extends <Oberklassen> : <Obertypen>
    (extent <Extension>
     keys <Schluesselmenge>)
    {<Schnittstellenspezifikation>}
```

- Schnittstellenspezifikation

```
[readonly] attribute
    <Typ>
    <Attributname>

relationship
    <Typ>|<Collection-Typ>< <Typ_1> >
    <Attribut_1>
    inverse <Typ_2>::<Attribut_2>

/* fuer Operationen:
<Typ>| void
    <Operationenname>
    (<Param_1>, ... , <Param_n>)
    raises (<Ausnahmen>)
```


- Definition der Parameter

```
on | out | inout
<Typ><Parametername>
```

- Typ- und Impl.hierarchie

	Typhierarchie	Implementierungshierarchie
Obertyp	Schnittstelle	Klasse
Untertyp	Schnittstelle oder Klasse	Klasse
Art der Vererbung	Mehrfachvererbung	Einfachvererbung

- Klassenhierarchie ermöglicht keine Mehrfachvererbung

3.5 Der Operationenteil und die OQL des ODMG-Standards

- Object-Query-Language ist Anfragesprache basierend auf SFW-Block von SQL-92
- zusätzlich: komplexe Werte, Objektidentitäten, Pfadausdrücke über Komponenteobjekte hinweg, Methoden, Overriding von Methoden
nicht nur Mengen, sondern allgemeinen Collection
neben dem SFW-Block auch bel. andere Anfrageblöcke
- funktionale, orthogonale Sprache
- Grundprinzip einer Anfrage: Ausgangspunkt -> Name eines atomaren, strukturierten oder Collection-wertigen Objektes oder Wertes
Name der Extension des Typs Person

Personen

ist gültige Anfrage (Collection-wertiges Objekt)

- **SFW-Block** wird zum Filtern von Mengen eingesetzt wie bei

```
select distinct struct (f: s.Studienfach, b: s.Betreuer)
from Studenten s
where s.Adresse.Ort = 'Rostock'
```

In dieser Anfrage wird die Extension Studenten nach dem Wohnort 'Rostock' gefiltert.

Adresse kein Attribut des Typs Student, aber vom Obertyp Person vererbt
Ort Komponenten des Strukturierten Attributs Adresse, mit Pfadausdruck erreichbar

- Relationale und objekterzeugende Anfragen
 - relationale Anfrage: letzte Anfrage nach Rostocker Studenten

- objekterzeugende Anfrage: statt des Typkonstruktors struct Objektkonstruktor (In ODL def. Typen)

```
Person (PANr: 8883494,
       Name: struct (Vorname: 'Otto', Nachname: 'Ohnmacht'))
```

erzeugt neues Objekt vom Typ Person
 Objekte hier jedoch nur für bestehende Typen
 keine dynamische Typisierung und Klassifizierung

- **Objekterhaltende Anfragen**

- Objektidentitäten der Rostocker Studenten in einer Multimenge aufsammeln

```
select s
from Studenten s
where s.Adresse.Ort = 'Rostock'
```

- keine dynamische Klassifizierung oder Typisierung des Anfrageergebnisses
- extensionsbasierte, keine klassenbasierte Anfrage
- Rostocker Studenten bilden neue Extension vom Typ Student als Anfrageergebnis, jedoch keine dynamisch erzeugte Unterklasse von Student
- auch Typ der Objekte nicht veränderbar

- **Orthogonalität**

- auf jede Collection Anfrageoperationen anwendbar

```
select z.Fach
from Huho.Zeugnis z
```

liefert eine Multimenge von STRINGS, die Prüfungsfächer des Studenten Hugo

- Menge von Prüfungsfächern aller Studenten

```
select distinct z.fach
from Studenten s, s.Zeugnis z
```

- **Nullwerte**

- Gewöhnungsbedürftig: Behandlung von undef. Objekten
- Laufzeitfehler bei

```
select s.Betreuer
from Studenten s
```

falls Betreuer mindestens eines Studenten nicht def.

- korrekte oder 'sichere' Anfrage wäre gewesen

```
select s.Betreuer
from Studenten s
where is_defined (s.Betreuer)
```

- **Ausnutzung von Typkonstruktoren und Klassenhierarchie**

- OQL erlaube direkt Vergleich von Mengen

```
select b
from b in Buecher
where set(RDBS, lehrbuch) >= b.Stichworte
```

OQL bietet Mengenoperationen an, aber folgende nicht erlaubt

```
Ausleihobjekte union Geraete
```

nur Mengen mit gleichen oder vergleichbaren Elementtypen können vereinigt werden (alle in Anfragen auftretenden Klassen müssen def. sein, Typ des Ergebnisses muss eindeutig sein)

- Unvergleichbare Klassen können mehr als eine kleinste gemeinsame Oberklasse haben: Eindeutigkeit nicht gegeben
- haben sie überhaupt keine gemeinsame Oberklasse, ist das Ergebnis nicht einmal darstellbar

- Pfadausdrücke einmal anders: geschachtelte Anfrage; relationale Anfrage

- weitere Klauseln

- Quantoren for all und exists
- Sortierung sort und Gruppierung group by mit having
- Aggregatfunktionen im Umfang von SQL
- Def. temporärer Relationen mit defined
- Operationen zur Typkonvertierung wie listtiset und flatten
- Aufruf bel. Methoden in jeder Klausel
- völlige Orthogonalität

```
max(select Gehalt from Angestellte)
```

im Gegensatz zu Standard-SQL erlaubt

3.6 Umsetzung in OODBPL-Systemen

- OODBPL: üblicherweise auf ODMG basierend
- einige Umsetzungen:
 - O₂: ODMG-OQL
 - Ontos: eigenes Object SQL...

- ODMG Sprachanbindung
 - einheitliches Typsystem zw Programmiersprache und DB:
Die für die ODL-Konzepte erzeugten PL-Klassen mit ihren Methoden können persistente oder transiente Objekte aufnehmen; ODL-Klassen bilden Teil der PL-Klassenbibliothek.
 - Einbettung erfolgt in Syntax der Programmiersprache; soll den "impedance mismatch" relationaler Embedded-SQL-Versionen vermeiden
 - Ergänzungen der Klassenbibliotheken so klein wie mgl halten; Methoden für deskriptive Anfragen und Transaktionsverwaltung hinzufügen
 - BD- und OOPL-Teile frei kombinierbar
 - persistente und transiente Objekte können zu einer Klasse gehören
- Fazit: ODMG Einschränkungen
 - Objekt fixiert in einer einzigen Klasse (keine Mehrfachzugehörigkeit, kein Klassenwechsel)
 - kein orthogonales Typkonzept
 - Programmiersprachensemantik statt Datenbanksemantik
 - nur extensionsbasierte Anfragen, keine klassenbasierten Anfragen
 - Mehrfachvererbung: nur Interfaces, keine Implementierung
 - Persistenzprinzip: jedes Binding reagiert anders
- Planung in ODMG nicht behandelt -> siehe VL Folien (halte ich nicht für prüfungsrelevant)
- ODMG Stand 2015
 - seit 2006 geplant, 2014 eingestellt
 - JDO als einzige Sprachanbindung von Java zu OODBMS
 - seit 2006 JPA als Sprachanbindung von Java zu RDBMS

4 Das objektrelationale Modell

4.1 Einführung in objektrelationale Konzepte

- Vergleich ORDM zu RDM:
- Typen, Typkonstruktoren und ADT (statt nur Standard-Datentypen im RDM)
- Objektidentitäten (statt nur sichtbare, änderbare, lokale Schlüssel im RDM)
- Tabellen (geschachtelt und mit Objekten statt nur flach und mit Werten im RDM; statt Klassen im OODM-Sinne)
- Untertypen (Typhierarchie)
- Untertabellen (statt nur Fremdschlüssel im RDM, statt Klassenhierarchie im OODM-Sinne)
- Komponentenobjekte (statt nur Fremdschlüssel im RDM); Pfadausdrücke (statt vieler Verbunde im RDM)
- Anfragen und Sichten (analog zum RDM)
- Methoden (In SQL: UDMs) (statt nur Anwendungsprogramme auf Sichten im RDM)

4.2 Der SQL:1999/SQL:2003 Standard

- **Objektrelationale Konzepte:**
- user defined types (UDTs)
- user defined functions (UDFs), User defined Methods (UDMs)
- Prozeduren, Funktionen, Methoden: Unterschiede bei Überladen und Overriding
- LOBs (Large Objects) und Locators für Laden von großen Daten bei effizienter Pufferausnutzung
- Typkonstruktoren (row, reference, array) (andere Collection-Konstruktoren: SQL:2003)
 - row: Tupelkonstruktor
 - array: einziger Collection Konstruktor
 - reference: Komponentenobjekt
- Typ-, Tabellenhierarchien, Sichthierarchien (Object Views)
- **Conformance Level**
- SQL:1999 Kern ist
SQL-92-Entry + einige Konzepte aus Transitional, Intermediate, Full Level + Kernkonzepte von SQL:1999-Foundation

- SQL:2003-Packages
Enhanced Datetime Facilities
Enhanced Integrity Management...
Basic Object Support (eingeschränkte strukturierte und Referenz-Typen, Typkonzept, Typ-Test-Prädikate), LOB-Unterstützung mit Locators)
Enhanced Object Support (alle Typkonstruktoren, Methoden, Tabellenhierarchie, Cast-Operatoren, Locators für komplexe Attributwerte)

4.3 Der Strukturteil des ORDB-Modells

- **Typen**
 - Standard-Datentypen
 - Typkonstruktoren (unbenannte Typen)
 - UDT: Datentyp
Name, Repräsentation, Beziehung zu anderen Typen
distinct types
strukturierte (benannte) Tupeltypen: create type
strukturierte (benannte) Objekttypen: create type
- **Prozeduren, Funktionen, Methoden**
 - UDF: Funktion (Methode, Prozedur)
Name, Signatur, Resultat, Impl
Prozedur: kein Überladen, statisches Binden
Funktion: Überladen, statisches Binden
Methoden: Überladen und Overriding, dynamisches Binden
- **unbenannter Typkonstruktor row type**
 - Tupeltyp hat keinen Namen, enthält Tupelwerte
 - Tupeltyp als Wertebereich eines Attributs in einer Tabelle
 - wird nur an dieser Stelle verwendet
 - keine speziellen Funktionen, Methoden definierbar

```

create table Personen    (PANr integer ,
Partner ref(Person) ,
Wohnung row
    (PLZ integer ,
    Ort varchar(30) ,
    ....
    )
)

```

- **Unbenannter Typkonstruktor array type**

- einziger Collection Typkonstruktor in SQL:1999
- maximale Länge wie bei varchar mgl
- Operationen: Zugriff über Positionsnummer
Kardinalität, Vergleich, Konstruktoren,...

```
create table Personen (PANr integer ,
                      Partner ref(Person) ,
                      Wohnung row
                        (PLZ integer ,
                        Ort varchar(30) ,
                        ....
                      ) [4]
)
```

eine Person kann bis zu 4 Personen haben

- **Unbenannter Typkonstruktor multiset type**

- ab SQL:2003 Multimengen-(Bag)-Konstruktor
- Test, ob Multimenge eine Menge ist: is a set
- Multimenge M_1 in eine Menge umwandeln: set(M_1)
- Element, Kardinalität, Teilmultimenge, Vereinigung, Durchschnitt, Differenz

```
create table Personen (PANr integer ,
                      Partner ref(Person) ,
                      Wohnung row
                        (PLZ integer ,
                        Ort varchar(30) ,
                        ....
                      ) multiset
)
```

- **UDT: Distinct Types**

- final: keine Untertypen definierbar
- definierbar: Vergleichsoperatoren, Casts, Methoden und Funktionen

```
create type T1 as integer final
create type T2 as integer final
```

erstellt zwei unvergleichbare Typen (zb Alter und Gewicht)

- **UDT: strukturierter (benannter) Typ**

- können überall verwendet werden, auch als Parametertypen
- Persistenz: Konstruktorfunktion Adresse() liefert Instanz des Typs (Default-Werte) oder insert

```
create type Adresse as (PLZ, integer, ...) not final
create type Person as (PANr integer, Partner ref(Person))
```

- **UDTs: nicht-instantiierbar**

- virtuelle Typen: in OOPs: abstrakte Klassen
- keine Instanzen definierbar
- nicht-instantiierbare Typen in SQL:2003

```
create type T1 as (...)
not instantiable not final
```

- **Typen als Attributtypen: UDTs und Typkonstruktoren**

- mit create type erzeugte Typen:
- können mit Methoden angereichert werden
- können mit Untertypen verfeinert werden (falls not final)
- können als Attributtypen verwendet werden: Methoden sind dann auf Attribute anzuwenden
- können als Tabellentypen (Relationenschema) verwendet werden: Methoden sind dann auf Objekte (oder Tupel) anzuwenden
- mit Typkonstruktoren row, array, multiset erzeugte Typen werden nur für Attribute eingesetzt
- keine Methoden und Untertypen definierbar

- **UDTs: Tupeltypen und Objekttypen**

- mit create type erzeugte Typen:
- können Tupeltypen für Tabellen sein (relationale, evtl geschachtelte Tabelle ohne Objektidentitäten): Tabelle besteht aus (Tupel-)Werten
- können Objekttypen für Tabellen sein (Objekttabelle oder Objektrelation mit Objektidentitäten): Tabelle besteht aus (Tupel-)Objekten
- Beispiel: Tupeltyp

```
create type Adresse as (PLZ integer, ...)
not final;
```

- Beispiel Objekttyp (durch Referenztypspezifikation)

```
create type person as (PANr integer, Partner ref(Person))
not final
ref is system generated;
```


Referenztypspezifikation von Objekttypen

- * ref is system generated
Echte Objektidentität nach OODM, aber Eindeutigkeit nur in Tabelle
- * ref from (Attributliste)
etwa PANr, dann Objektidentität funktional bestimmt
- * ref using Typ
etwa integer, dann durch Benutzer zugewiesen

• Untertypen

- Einfach- oder Mehrfachvererbung(letzteres erst ab SQL4)
- Instantiierbarkeit und Abschluss
- nur Methoden dynamisch gebunden, nicht Funktionen

```
create type Hiwi under Angestellter as (...)  
instantiable not final ...
```

• Tabellen: Objektidentitäten

- Typen ergänzen um OID => objektrelationale Tabellen
- objektrelationale Tabelle (auch Objekttable) ersetzt Begriff der Klasse, sie haben
Objektidentitäten durch OIDs
eine Extension (die Relation)
einen Zustandstyp (mit of eingeleitet)

```
create table Angestellte of Angestellter  
(ref is oid system generated)
```

• Objektidentitäten und ihre Referenzgenerierungsart

- Referenzgenerierungsart der Tabelle muss zu Referenztypspezifikation des Typs passen
ref is name_der_oid-spalte system generated muss zu ref is system generated in Typ passen
ref is name_der_oid-spalte derived muss zu ref from (Attributliste) in Typ passen
ref is name_der_oid-spalte user generated muss zu ref using Typ in Typ passen
- bei derived muss Attributliste primary key oder (unique und not null) sein
- Die oid-Spalte hat dann folgenden Typ:
ref(Typname)scope(Tabellenname)
- abstrakte Domäne der Objektidentität ist für den Typ spezifiziert

- die aktuelle Objektmenge (Extension) wird durch scope auf die aktuell in der Tabelle vorhandenen OIDs eingeschränkt
- parallel zur Typhierarchie mit unter auch **Tabellenhierarchien**
 - aufgrund der Isomorphie von Typ- und Tabellenhierarchie auch hier nur Einfachvererbung erlaubt
 - Untertabelle ist Teilmenge der Obertabelle (projiziert auf die Spalten der Obertabelle)
 - Untertabelle kann nichtredundant /redundant definiert bzw gespeichert werden:
nichtredundant, split instance
redundant, repeat class

4.4 Der Operationenteil des ORDB-Modells

- **Die SFW-Klausel im objektrelationalen SQL**
 - strukturidentisch mit SQL: SELECT FROM WHERE GROUP BY HAVING
 - und Aggregatfunktionen, Sortierung, Umbenennung, Mengenoperationen, Verbundoperationen,... auch vom relationalen SQL
 - zusätzlich: tiefe vs. flache Extension
navigierende Anfragen (Pfadausdrücke)
Anfragen mit Methodenaufrufen
typspezifische Anfragen (etwa mit Collection-Attributen)
 - Anfragen an Tabelle auch an Untertabelle
select ... from Angestellte...
liefert auch Hiwis
 - Einschränkungen der Substituierbarkeit mgl
 - Pfadausdrücke für Komponentenobjekte:
Referenzen mit scope verwendbar; Methoden; Dereferenzieren liefert Objekt mit Zustand
- **Object Views: Sichthierarchien**
 - objekterhaltende Sichten

```
create view Keine_Hiwis of Angestellter
as select ... from only (Angestellte)
```
 - abgeleitete Sichten auf referenzierten (Komponenten-)Tabellen:
 - Anfrageergebnisse können dynamisch in neuen Sichthierarchien eingeordnet werden
 - letzteres verwirklicht dynamische Typisierung und Klassifizierung *per Hand*

4.5 Höhere Konzepte des ORDB-Modells

- **Methoden und Konstruktoren**

```
create type Angestellter as
    (Basisgehalt decimal(9,2),
    ...)
    instantiable not final
    method Endgehalt()
        returns decimal (9,2);

create method Endgehalt() for Angestellter
    <Methodenrumpf>;

create type Hiwi under Angestellter as
    (...)
    instantiable not final
    overriding method Endgehalt()
        returns decimal(9,2);
```

Konstruktorfunktion *Adresse()* liefert Instanz des Typs (Default-Werte) oder insert

- **UDF: Methodendefinition**

- Methoden sind Funktionen, die an einen UDT gebunden sind
- impliziter self-Parameter, Signatur und Impl getrennt
- Signaturen müssen identisch sein
 - No-Varianz (statt Ko- und Kontravarianz)
 - kein multiple dynamic dispatch
- Methodenaufruf mit Dot-Notation

Konzept in OODM	Konzept in SQL:1999/SQL:2003
Wert Objekt Tupelkonstruktor Mengenkonstruktor Arraykonstruktor Multimengenkonstruktor Listenkonstruktor ...	Tupel ohne OID Tupel mit OID Row-Type, strukturierter Typ (UDT) nicht vorhanden Array-Type Multiset-Type nicht vorhanden ...
Klasse Implementierung eines Typs Instanz Schlüssel virtuelle Klasse vordefinierter ADT	Objekttabelle (Schema; OID) — Relation (Menge von Tupeln mit OID) Schlüssel not instantiable Distinct Type (UDT)
Attribut Anfrage-Methode Klasse—Komponentenklasse-Beziehung mit inverser Beziehung	Attribut vom Typ ODT Prozedur, Funktion, Methode (UDF) Reference-Type —
Klassenhierarchie Typhierarchie Implementierungshierarchie	Tabellenhierarchie under table Typhierarchie under type —
Methode Overriding	(Prozedur, Funktion,) Methode (UDF) Overriding

Abbildung 9: Zusammenfassender Vergleich OODM und Standard Konzepte

4.6 Umsetzung in ORDBMS

- fehlende Umsetzung in ORDBMS: viele Teile des Standards fehlen bzw. sind abweichend davon umgesetzt
- Typkonstruktoren kaum vorhanden und nicht orthogonal umgesetzt
- Klassen- und Typhierarchie teilweise nur eine von beiden umgesetzt
- Objektidentitäten: nicht alle drei Optionen der Referenzgenerierung umgesetzt
....

4.7 Fazit und Vergleich

- **Modellierung - das Manta-Problem**
 - Rollen, Rollenwechsel, Mehrfachzugehörigkeit von Objekten zu Klassen
 - Typ- und Klassenhierarchie (wie OODM, ORDM)
- **Sichten - das Manta-Problem**
 - Klassen durch Anfragen und zusätzliche Strukturdefinitionen dynamisch ableiten
 - diese Sichtklassen wie Basisklassen nutzbar machen
- **Datenunabhängigkeit - das Fahrrad-Problem**
 - Drei Ebenen Konzept gilt für DBMS, nicht für RDBMS
 - je nach Anwendungssituation, flexible Speicherstrukturen intern ermöglichen

- impedance mismatch mindern
- Persistenz: Extension und Fortpflanzung
- **Mehrfachzugehörigkeit von Klassen**
 - keine Lösung (in OODBPL): Rollenobjekte als Komponente jedem Objekt mitgeben
Nachteile: Objekte und ihre Rollen bilden dann keine Klassen- und Typhierarchie
Rollen erweitern dann nicht den Typ
Vererbung und Overriding dann nicht nutzbar
so simulieren kann man es auch relational
 - keine Lösung (in OODBPL und manchen ORDBMS)
tiefe Extension simuliert Mehrfachzugehörigkeit
aber kein allgemeines Inklusionsprinzip (disjunkter Durchschnitt)
Objekte werden immer in speziellster Klasse erzeugt
- **Manta-Problem**

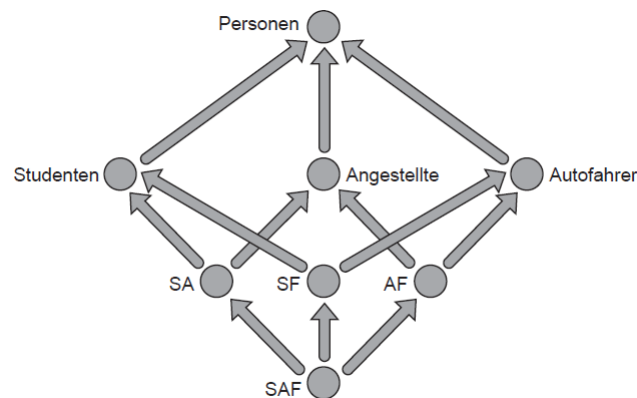


Abbildung 10: Mantra Problem Beispiel

Lösung:

- im Schema Klasse K definieren
- mit klassenbasierten Anfragen definiert man dynamisch die Unterklassen
- in diesen Sichtklassen wird dann die von Klasse K vererbten Methoden jeweils geeignet redefiniert
- Falls Objekt in mehreren Klassen, dann Konfliktauflösemethode spezifizieren
- **Einschränkungen ODMG** siehe 3.6
- **Einschränkungen SQL:2003**

- Standard nicht vollständig
Bsp: Typkonstruktoren schrittweise eingeführt und nicht orthogonal
- Klassen und Typhierarchie müssen isomorph (bijektiv - eineindeutig) sein
- impedance mismatch: Objekttabellen und Java-Klassen
- keine persistente Programmierungsumgebung nach Atkinson (siehe Kap 5)
- Mehrfachvererbung nicht vorhanden
- Anfragen nur extensionsbasiert