3. Objekt-orientierte DBS

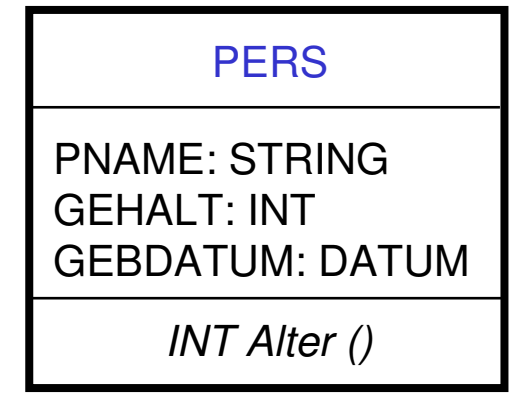
- OODBS-Konzepte
  - Kapselung/Abstrakte Datentypen, Erweiterbarkeit
  - Typhierarchie/Vererbung, Überladen /spätes Binden
  - Komplexe Objekte: Objektidentität
  - Komplexe Objekte: Typkonstruktoren
- ODMG-Objektmodell / ODL
  - Literale vs. Objekte
  - Typen: Interfaces vs. Klassen
  - Attribute und Beziehungen (relationships)
  - Objektdefinition mit ODL
- O/R-Mapping Frameworks
  - Java Data Objects (JDO)
  - Hibernate

Definition eines objekt-orientierten DBS

- OODBS-Manifesto von 1990:
  - OODBS muss ein DBS sein
  - OODBS muss ein objekt-orientiertes System sein
- DBS-Aspekte
  - Persistenz, Externspeicherverwaltung, Datenunabhängigkeit
  - Transaktionsverwaltung, Ad-Hoc-Anfragesprache
- essentielle OOS-Aspekte:
  - Objektidentität, komplexe Objekte, Kapselung
  - Typ-/Klassenhierarchie, Vererbung, Überladen und spätes Binden
  - operationale Vollständigkeit, Erweiterbarkeit
- optionale OOS-Aspekte:
  - Mehrfachvererbung, Versionen, lang-lebige Transaktionen •••

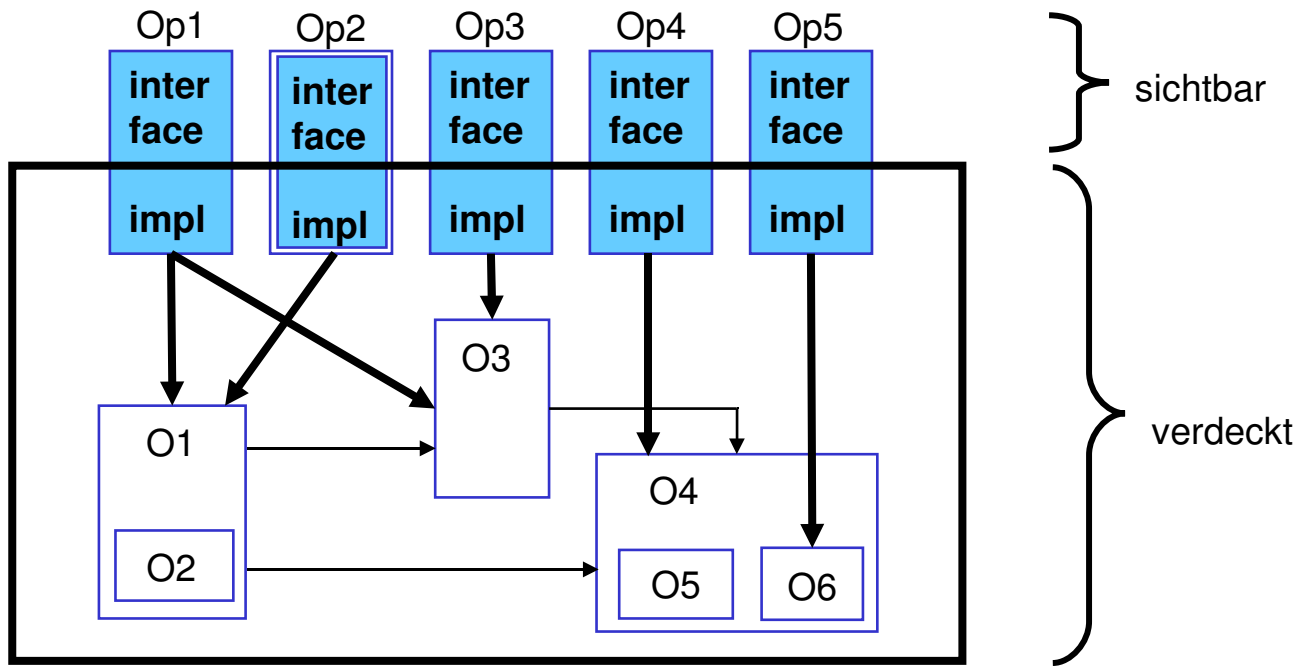
Objekttypen / Klassen

- Objekt = Struktur + Verhalten
- Spezifikation durch Objekttyp / Klasse
  - Struktur: Attribute und ihre Wertebereiche
  - Verhalten: zulässige Operationen / Methoden
- Objekt = Instanziierung eines Typs mit konkreten Wertebelegungen der Attribute
- Arten von Typen
  - Vordefinierte vs. benutzerdefinierte Typen
  - einfache vs. komplexe Typen
- Arten von Operationen
  - Vordefinierte vs. benutzerdefinierte Operationen
  - Konstruktoren (Erzeugung/Initialisierung neuer Instanzen eines Objekttyps)
  - Destruktoren
  - lesende Operationen (Beobachter)
  - Mutatoren

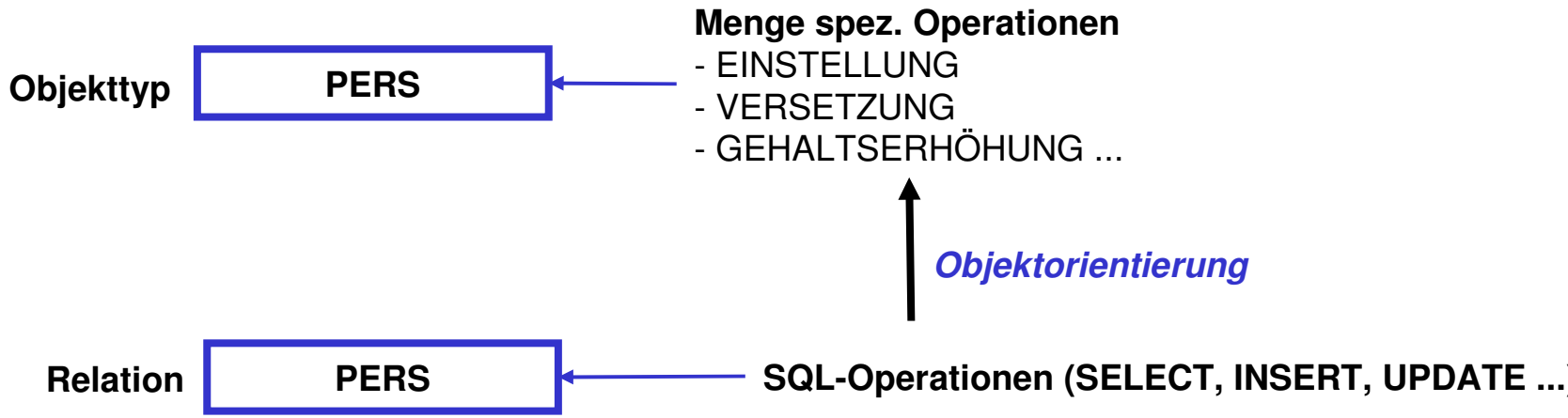


Kapselung

- Strenge Objekt-Orientierung verlangt Kapselung (Information Hiding)
  - Struktur von Objekten wird verborgen
  - Verhalten des Objektes ist ausschließlich durch seine Operationen (Methoden) bestimmt
  - nur Namen und Signatur (Argumenttypen, Ergebnistyp) von Operationen werden bekannt gemacht
  - Implementierung der Operationen bleibt verborgen



Kapselung (2)



- Verwaltung von Objekttypen und Operationen im DBS
  - zusätzliche Anwendungsorientierung im DBS gegenüber Stored Procedures
  - verringerte Kommunikationshäufigkeit zwischen Anwendung und DBS
  - Reduzierung des "impedance mismatch"
- Vorteile der Kapselung
  - hoher Abstraktionsgrad
  - logische Datenunabhängigkeit
  - Datenschutz

Kapselung (3)

- Aber: strikte Kapselung oft zu restriktiv
  - eingeschränkte Flexibilität
  - mangelnde Eignung für Ad-Hoc-Anfragen
- Koexistenz von attributbezogenen Anfragen und objekttypspezifischen Operationen
  - ADTs (Kapselung) vs. Benutzerdefinierte Datentypen (BDT)
  - rekursive Anwendbarkeit des BDT-Konzeptes auf Klassen und Attribute

BDT-Anwendungsbeispiel

- Verwaltung räumlicher Objekte
  - Relationenmodell bietet keine Unterstützung
  - hohe Komplexität bereits in einfachen Fällen
  - Beispiel: Darstellung von Rechtecken in der Ebene

a) Relationenmodell

R-ECK (RNR, X1, Y1, X2, Y2: INTEGER)

Finde alle Rechtecke, die das Rechteck ((0,0) (1,1)) schneiden!

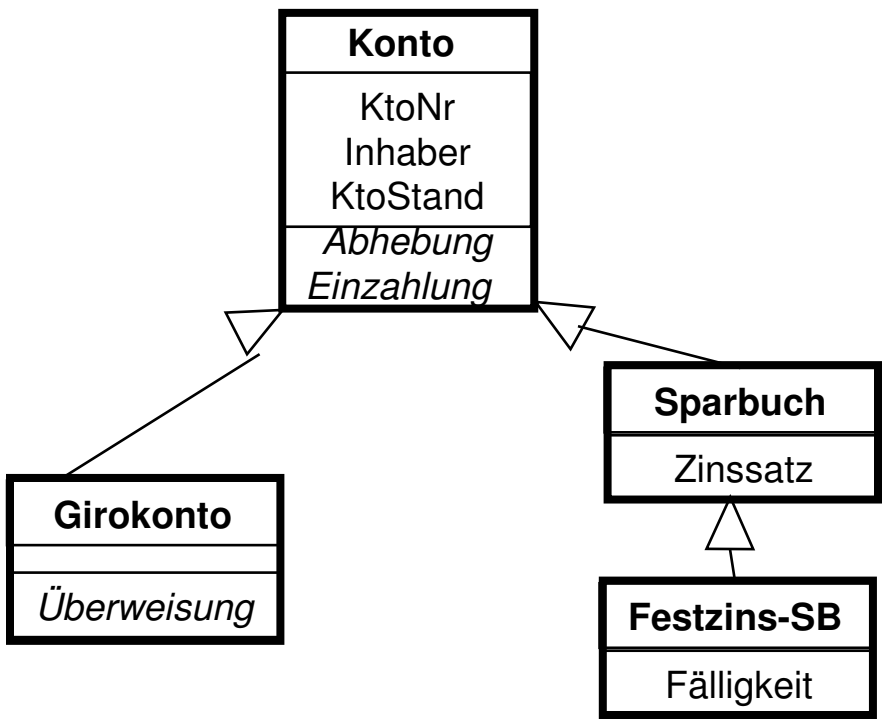
```
SELECT  RNR  FROM R-ECK
WHERE  (X1 > 0  AND  X1  < 1 AND Y1 > 0 AND Y1 < 1) OR  (X1 > 0  AND  X1  < 1 AND Y2 > 0 AND Y2 < 1)
   OR  (X2 > 0  AND  X2  < 1 AND Y1 > 0 AND Y1 < 1) OR  (X2 > 0  AND  X2  < 1 AND Y2 > 0 AND Y2 < 1) OR . . .
```

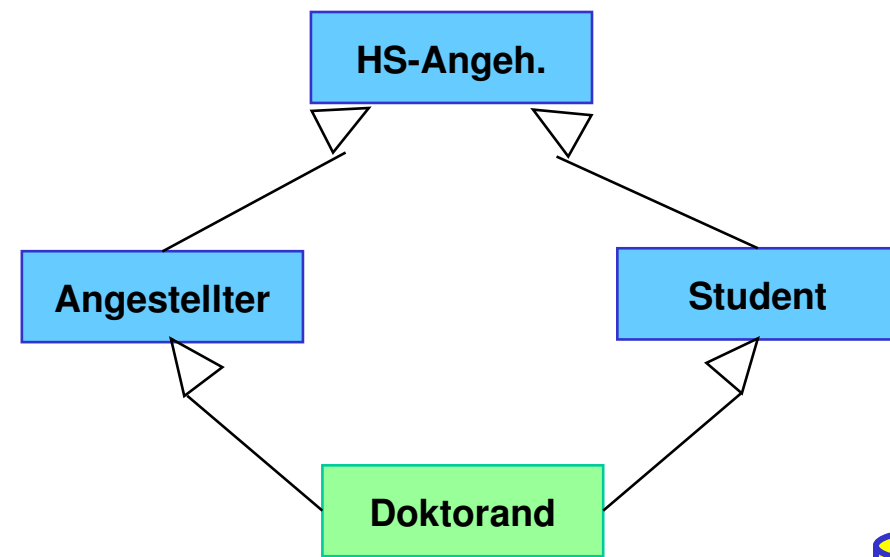
b) Lösung mit benutzerdefiniertem Datentyp BOX

BDT **BOX** mit Funktionen INTERSECT, CONTAINS, AREA, usw.

Generalisierung

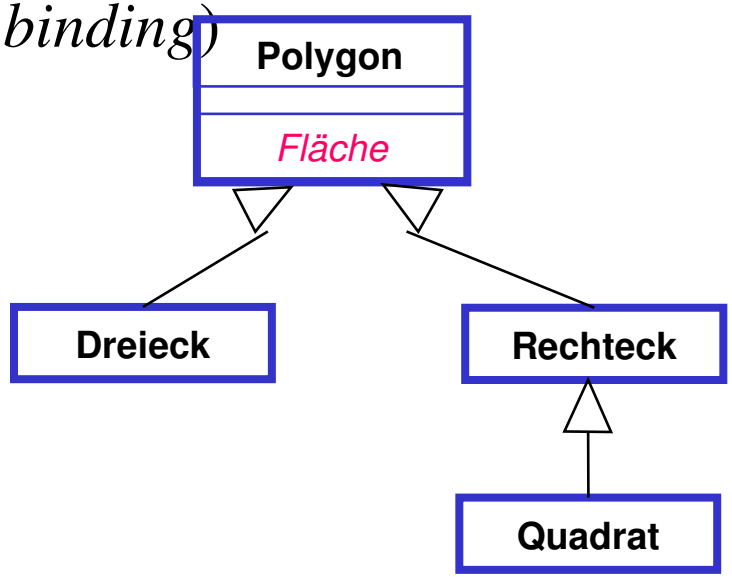
- Generalisierungs-/Spezialisierungshierarchie (IS-A-Beziehung)
  - Vererbung von Attributen, Methoden, Integritätsbedingungen ...
  - Arten der Vererbung: einfach (Hierarchie) vs. mehrfach (Typverband)
  - Prinzip der *Substituierbarkeit*: Instanz einer Subklasse B kann in jedem Kontext verwendet werden, in dem Instanzen der Superklasse A möglich sind (jedoch nicht umgekehrt)
    - impliziert, dass Klasse heterogene Objekte enthalten kann





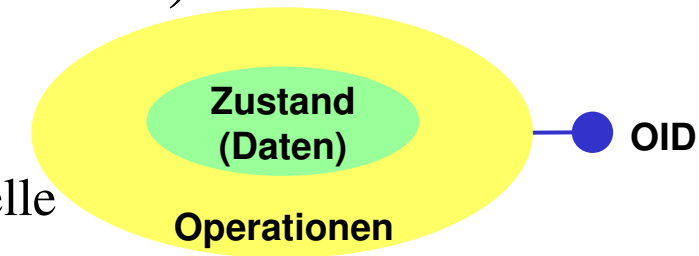
Überladen (Overloading)

- derselbe Methodenname wird für unterschiedliche Prozeduren verwendet (polymorphe Methoden)
  - erleichtert Realisierung nutzender Programme und verbessert Software-Wiederverwendbarkeit
- Overloading innerhalb von Typ-Hierarchien:
  - Redefinition von Methoden für Subtypen (Overriding)
  - spezialisierte Methode mit gleichem Namen
- Überladen impliziert dynamisches (spätes) Binden zur Laufzeit (*late binding*)



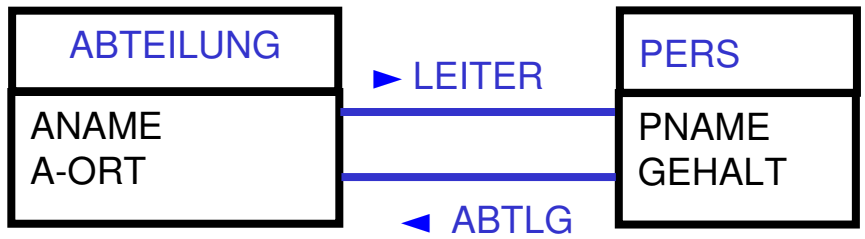
Objektidentität

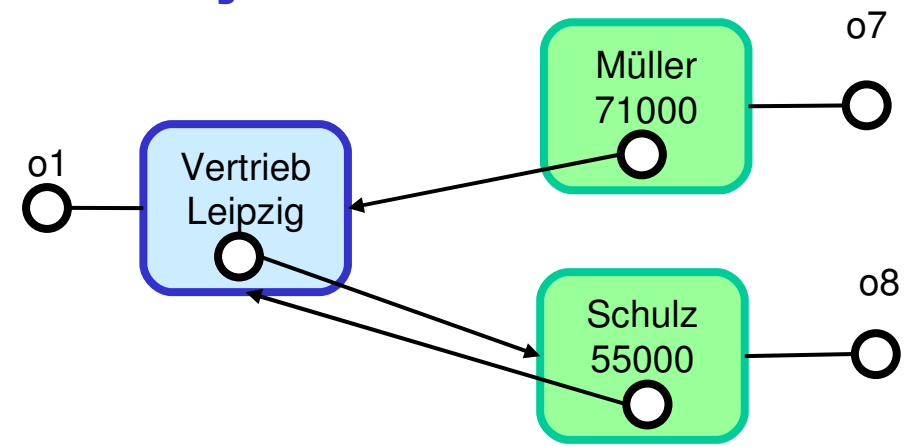
- OODBS: Objekt = (OID, Zustand, Operationen)
  - OID: Identifikator
  - Zustand: Beschreibung mit Attributen
  - Operationen (Methoden): definieren externe Schnittstelle



- Objektidentität
  - systemweit eindeutige Objekt-Identifikatoren
  - OID während Objektlebensdauer konstant, üblicherweise systemverwaltet
  - OID tragen keine Semantik (<-> Primärschlüssel im RM)
  - Änderungen beliebiger Art (auch des Primärschlüssels im RM) ergeben *dasselbe* Objekt
- Vorteile gegenüber „wertebasiertem“ Relationenmodell
  - Trennung von Identität und (Werte) Gleichheit
  - Notwendigkeit künstlicher Primärschlüssel wird vermieden
  - Beziehungen können über stabile OID-Referenzen anstelle von Fremdschlüsseln realisiert werden
  - einfachere Realisierung komplexer (aggregierter) Objekte
  - effizienterer Zugriff auf Teilkomponenten

OIDs: Komplexe Objekte






```
class ABT  ( ANAME:  STRING,
             A-ORT: STRING,
             LEITER:  REF (PERS) ...
```

```
class PERS   ( PNAME: STRING,
               GEHALT: INT,
               ABTLG: REF (ABT)  (* Referenz-Attribut *) ...
```

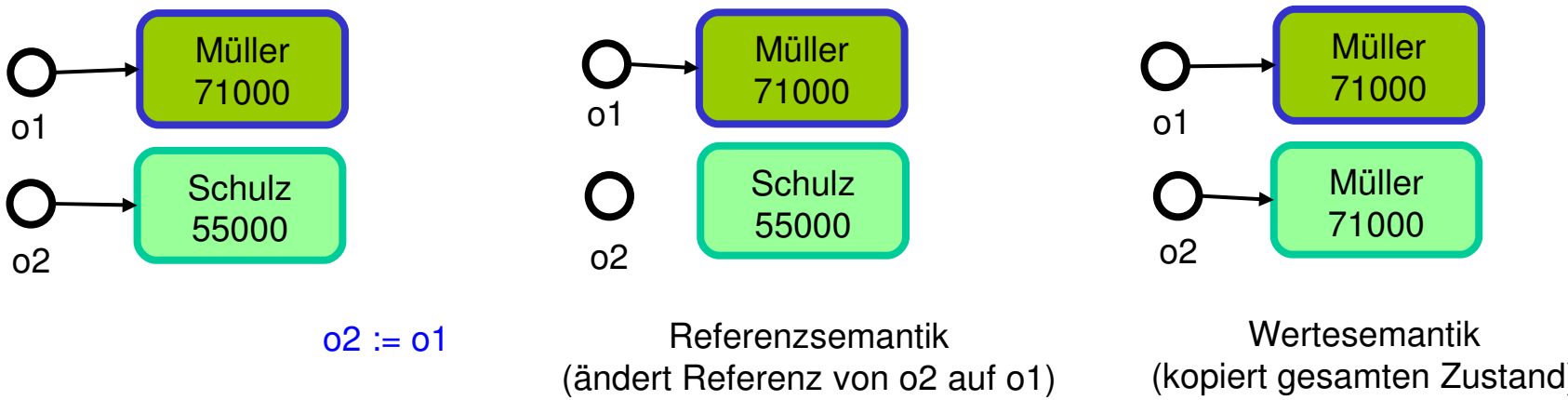
- Realisierung von Beziehungen über OIDs (Referenz-Attribute)
  - Objekt-Ids / Referenzen erlauben Bildung komplexer Objekte bestehend aus Teilobjekten
  - gemeinsame Teilobjekte ohne Redundanz möglich (referential sharing)

- implizite Dereferenzierung über *Pfadausdrücke* anstatt expliziter Verbundanweisungen

PERS->ABTLG->A-ORT

Objektidentität (3)

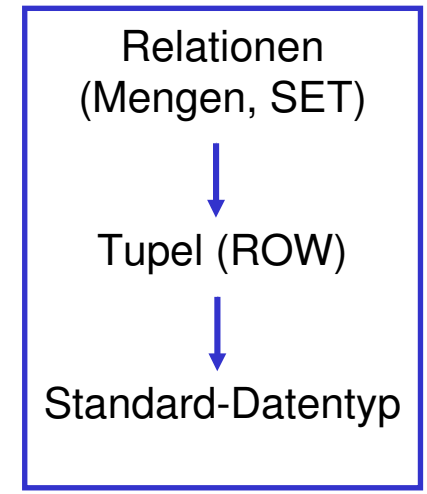
- Identität vs. Gleichheit
  - zwei Objekte sind identisch (O1 == O2), wenn sie dieselbe OID haben
  - zwei Objekte sind gleich (O1 = O2), wenn sie den gleichen Zustand besitzen
    - flache Gleichheit: Werteübereinstimmung aller Attribute inkl. Referenzattribute
    - tiefe Gleichheit: Werteübereinstimmung auch für alle referenzierten Objekte außer für Referenzattribute
- Zuweisungen: Referenzsemantik vs. Wertesemantik



- eindeutige Semantik von Änderungsoperationen im Gegensatz zu Werteänderungen im RM

Komplexe Objekte: Typkonstruktoren

- Relationenmodell
  - nur einfache Attribute, keine zusammengesetzte oder mengenwertige Attribute
  - nur zwei Typkonstruktoren: Bildung von Tupeln und Relationen (Mengen)
  - keine rekursive Anwendbarkeit von Tupel- und Mengenkonstruktoren



- OODBS
  - Objekte können Teilobjekte enthalten (Aggregation): eingebettet (Komposition, Wertesemantik) oder über OIDs referenziert
  - Objektattribute können sein:
    - einfach (Standardtypen: Integer, Char, ...)
    - über Typkonstruktoren strukturiert / zusammengesetzt
    - Instanzen benutzerdefinierter Typen
    - Referenzen

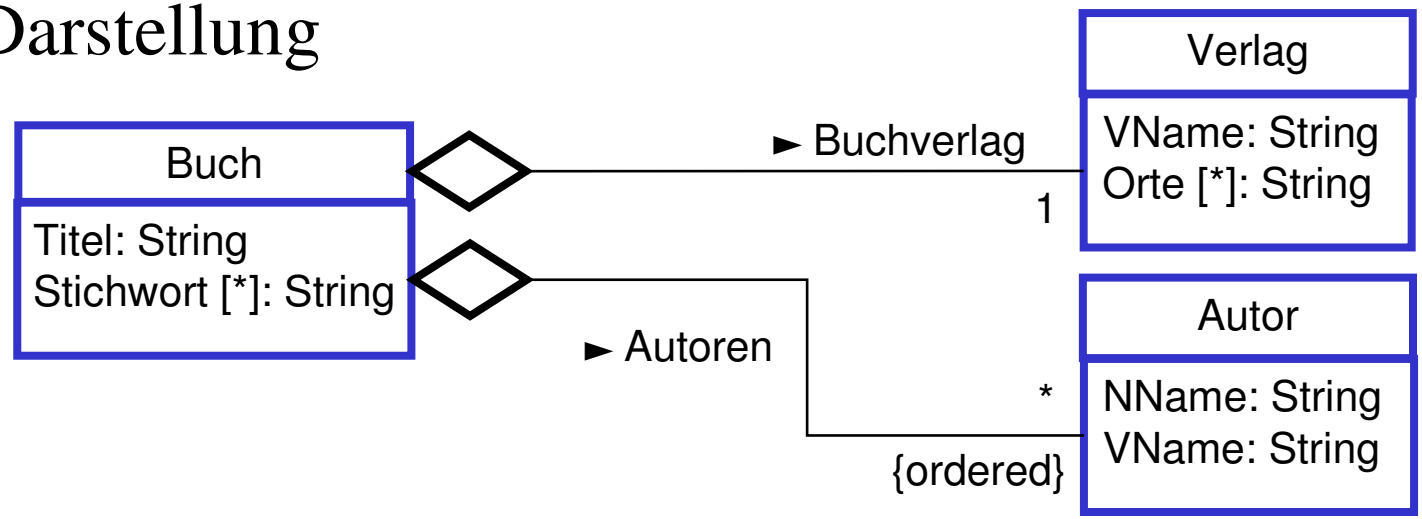
Komplexe Objekte: Typkonstruktoren

- Typkonstruktoren zum Erzeugen strukturierter (zusammengesetzter) Datentypen aus Basistypen
  - TUPLE (ROW, RECORD)
  - SET, BAG (MULTISET)
  - LIST (SEQUENCE), ARRAY (VECTOR)
- SET/BAG/LIST/ARRAY verwalten homogene Kollektionen: Kollektionstypen

Typ	Duplikate	Ordnung	Heterogenität	#Elemente	Elementzugriff über
TUPLE	JA	JA	JA	konstant	Namen
SET	NEIN	NEIN	NEIN	variabel	Iterator
BAG	JA	NEIN	NEIN	variabel	Iterator
LIST	JA	JA	NEIN	variabel	Iterator / Position
ARRAY	JA	JA	NEIN	konstant	Index

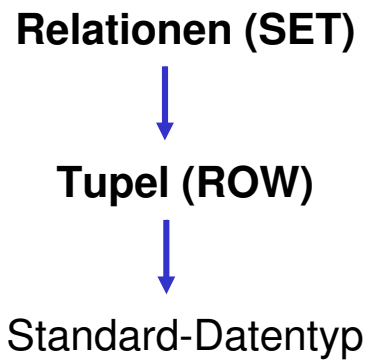
Komplexe Objekte: Beispiel

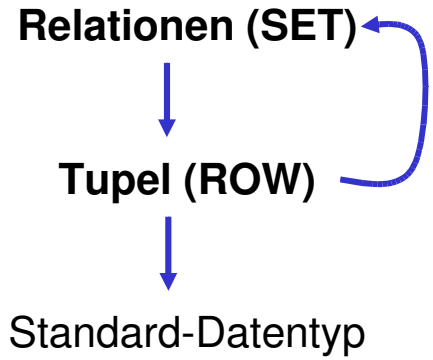
- UML-Darstellung

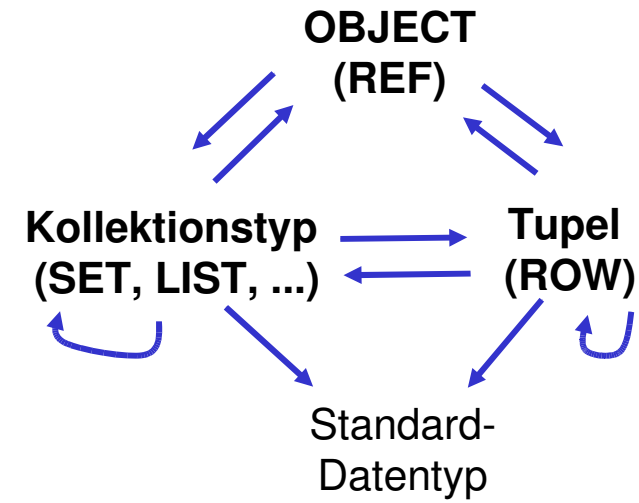


- Klasse "Buch“:

Vergleich Datenmodelle







Relationenmodell **NF2** **Objektmodell**

- Ziel: beliebige (rekursive) Kombinierbarkeit der Typkonstrukturen
- Generische Operatoren für aggregierte Objekte/Kollektionen
  - Komponentenzugriff
  - navigierender Zugriff über Iteratoren
  - Änderungen ...

ODMG Kollektionstypen

```
interface Collection: Object {
unsigned long     cardinality();
boolean           is_empty();
void              insert_element (in any element);
void              remove_element (in any element);
boolean           contains_element (in any element);
Iterator          create_iterator (in boolean stability);
}
```

```
interface Set: Collection {
Set               union_with (in Set other);
Set               intersection_with (in Set other);
Set               difference_with (in Set other);
boolean           is_subset_of (in Set other);
boolean           is_superset_of (in Set other);
}
```

```
interface Bag: Collection {
Bag               union_with (in Bag other);
Bag               intersection_with (in Bag other);
Bag               difference_with (in Bag other);
}
```

```
interface Dictionary: Collection {
exception         KeyNotFound {any key};
any               lookup (in any key) raises (KeyNotFound);
•••}
```

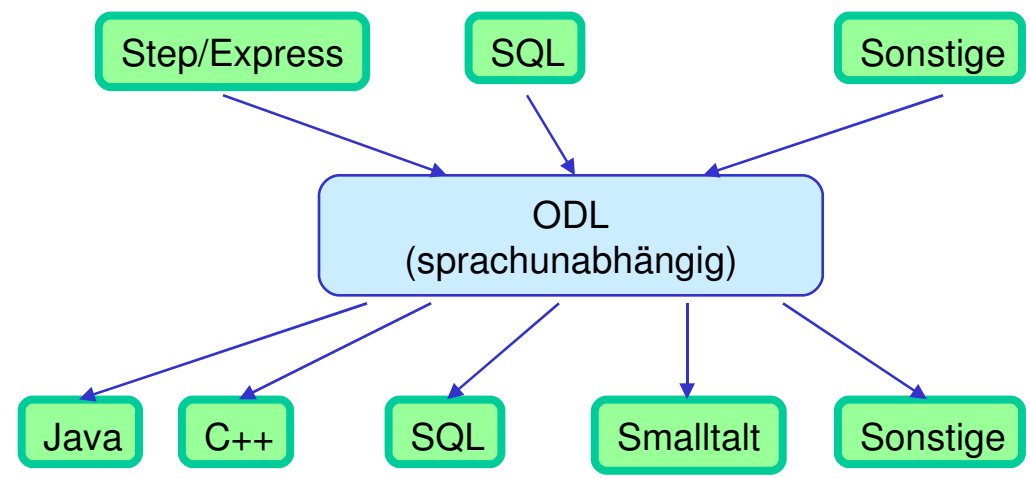
```
interface Iterator: Object {
exception         NoMoreElements{};
boolean           is_stable();
boolean           at_end();
void              reset();
any               get_element () raises (NoMoreElements);
any               next_position () raises (NoMoreElements);
}
```

```
interface List: Collection {
void              replace_element_at (in unsigned long index,
                  in any element);
void              remove_element_at (in unsigned long index);
void              retrieve_element_at (in unsigned long index);
void              insert_element_first (in any obj);
void              insert_element_last (in any obj); • • •
List              concat (in List other);
}
```

```
interface Array: Collection {
void              replace_element_at (in unsigned long index,
                  in any element);
void              remove_element_at (in unsigned long index);
void              retrieve_element_at (in unsigned long index);
void              resize (in unsigned long new_size);
}
```

ODMG (Object Data Management Group) *

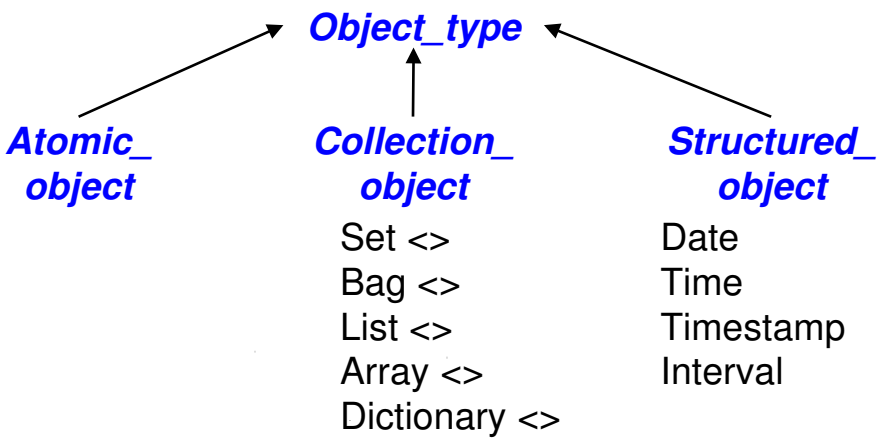
- Ziel: Entwicklung von Standards für objektorientierte DBS innerhalb OMG (Object Management Group, *www.omg.org*)
  - ODMG-Gründung 1991, Auflösung 2001
  - ODMG-Versionen: 1993 (V1), 1997 (V2), 2000 (V3)
  - partielle Produktunterstützung von ODL/OQL, z.B. FastObjects (Poet)
  - Java-Arbeiten gingen in JDO-Standard (Java Data Objects) ein
- ODMG-Standardisierung besteht aus
  - Objektmodell (Erweiterung des OMG Kern-Objektmodells)
  - Objekt-Definitionssprache (**ODL**)
  - Objekt-Anfragesprache (OQL)
  - Sprachbindungen (language bindings) für C++, Smalltalk, Java



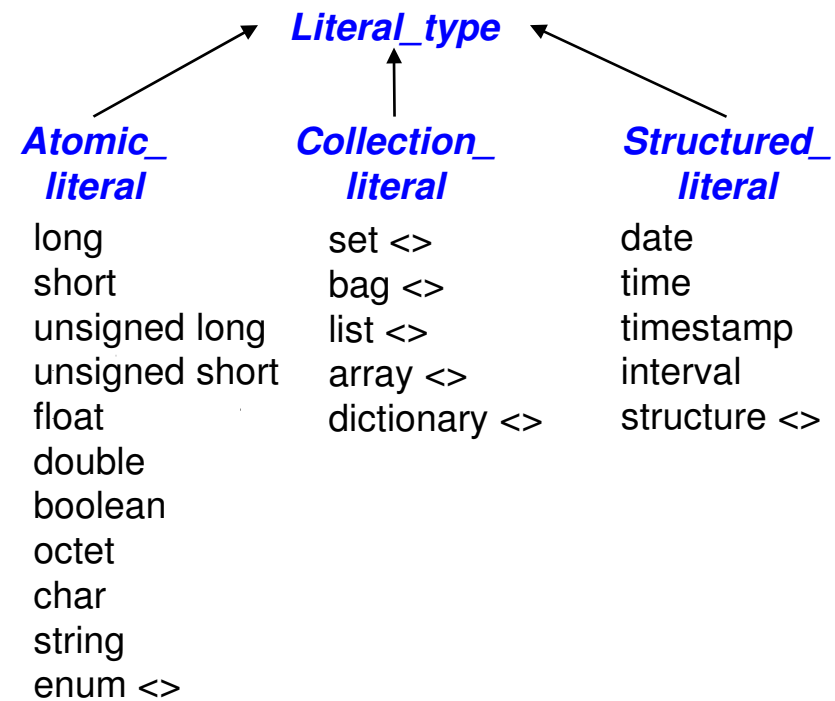
* http://www.odmg.org

ODMG-Objektmodell: Überblick

- Objekte (haben OID) vs. Literale/Werte (ohne Objekt-Identifier)
  - sowohl Literale als auch Objekte haben einen Typ
  - Objekte eines Typs weisen gemeinsame Charakteristika bezüglich Zustand (Attribute, Beziehungen) und Verhalten (Operationen) auf
- Benutzerdefinierte Typen erweitern vordefinierte Typen



kursive Typen: nicht direkt instantiierbar!



Typen

- Typen bestehen aus
  - externer Spezifikation: Festlegung der sichtbaren Eigenschaften, Operationen und Ausnahmen (exceptions)
  - einer (oder mehreren) Implementierung(en): sprachabhängige Festlegung der Datenstrukturen und Methoden zur Realisierung der Schnittstelle
- Typ-Spezifikation ist implementierungsunabhängig; Beschreibung durch ODL
  - Interface-Definitionen: nur Operationen, nicht instanziierbar
  - Klassen-Definitionen: Operationen + Zustand (Attribute, Beziehungen), instanziierbar
  - Literal-Definitionen: Zustandsbeschreibung eines Literal-Typs

```
interface Auto      { int Alter (); ... };
class Person        { attribute string Name; ... };
struct Complex      { float re; float im; };
```

- Typ-Implementierung ist sprachabhängig und im DB-Schema nicht sichtbar

Typen: Generalisierung

- Typen können durch Sub-Typen spezialisiert werden (IS-A-Beziehung, Vererbung von Operationen)
  - Definition zusätzlicher Operationen
  - Operationen können überladen werden (Overloading)
- Interfaces
  - Einfach- und Mehrfachvererbung
  - Interfaces können von anderen Interfaces erben/abgeleitet werden, jedoch nicht von Klassen **interface** PKW : Auto { ... }
- Klassen
  - können von Interfaces abgeleitet werden
  - zusätzliche Zustands-Vererbung (nur Einfachvererbung): EXTENDS-Beziehung

```
class Angestellter EXTENDS Person {
attribute float Gehalt;
attribute date Einstellungstermin; ...
}
```

ODL-Syntax

- Typspezifikation durch Interface- oder Class-Deklaration

<interface_dcl> ::= <interface_header> {[<interface_body>]}
<interface_header> ::= **interface** <identifier> [<inheritance_spec>]
<inheritance_spec> ::= : <identifier> [, <inheritance_spec>]

<class_dcl> ::= <class_header> { <interface_body> }
<class_header> ::= **class** <identifier> [**extends** <class-identifier>]
[<type_property_list>]

<type_property_list> ::= ([<extent_spec> | <key_spec>])
<extent_spec> ::= **extent** <string>
<key_spec> ::= key[s] <key_list>
<key_list> ::= <key> | <key>, <key_list>
<key> ::= <property_name> | (<property_name_list>)

- Beispiel

class Professor **extends** Person
(**extent** Profs **keys** PNR, (Name, Gebdat))
{
<interface_body> };

Objekte

- explizite Erzeugung neuer Objekte durch new-Konstruktor

```
interface ObjectFactory {
        Object new ();
}
```

```
interface Object {
        boolean same_as (in Object an Object); //Identitätstest
        Object copy ();
        void delete (); ...    }
```

  - datenbank-weit eindeutige Objekt-Identifier (→ Referenzierbarkeit)
  - Objekte können optional über Namen angesprochen werden ("Einstiegspunkte" in die Datenbank)
- Objektlebensdauer: transient oder persistent
  - einheitliche Verwaltung beider Objektarten (↔ relationale DBS)
- 2 Arten zur Verwaltung persistenter Objekte
  - Persistenz durch Erreichbarkeit: alle von einem persistenten Objekt (Einstiegs-punkt/Wurzel-Objekt) erreichbaren Objekte werden dauerhaft in der DB gespeichert
  - explizites Anlegen eines **Extent** zu einem Typ T:
    - umfasst Menge von (persistenten) Instanzen von T
    - für Subtyp T2 von T gilt, dass der Extent von T2 eine Teilmenge des Extents von T ist

Attribute

- Objekteigenschaften: Attribute und Relationships
- Attribute
  - sind jeweils genau 1 Objekttyp zugeordnet
  - Attributwerte sind Literale oder Objekt-Identifier (Nullwert: nil)
  - Attribute selbst sind keine Objekte (keine OID)

```
class Person {

        attribute   date          Geburtsdatum;
        attribute   enum          Geschlecht {m, w};
        attribute   Adresse       Heimanschrift;
        atrribute   set<string>   Hobbies;
        attribute   Abt           Abteilung;
}
```

Relationships

- Relationship: binäre Beziehung zwischen 2 Objekttypen
  - 3 Arten: 1:1, 1:n, n:m
  - symmetrische Definition in beiden beteiligten Objekttypen
  - jede Zugriffsrichtung (traversal path) bekommt eigenen Namen
- symmetrische Beziehungen erlauben automatische Wartung der referentiellen Integrität
- Repräsentation von (einfachen) n:m-Beziehungen ohne künstliche Objekttypen
- Pfade auf Objektmengen können geordnet sein (z. B. über List)

```
class Vorlesung { • • •
relationship set<Student> Hörer inverse Student::hört;
}
class Student {  • • •
relationship set<Vorlesung> hört inverse Vorlesung::Hörer;
}
```

Relationships (2)


```
class A {
        relationship                    inverse         }
class B {
        relationship                    inverse         }
```


```
class C {
        relationship                    inverse         }
class D {
        relationship                    inverse         }
```


```
class E {
        relationship                    inverse         }
class F {
        relationship                    inverse         }
```

ODL-Syntax (2)

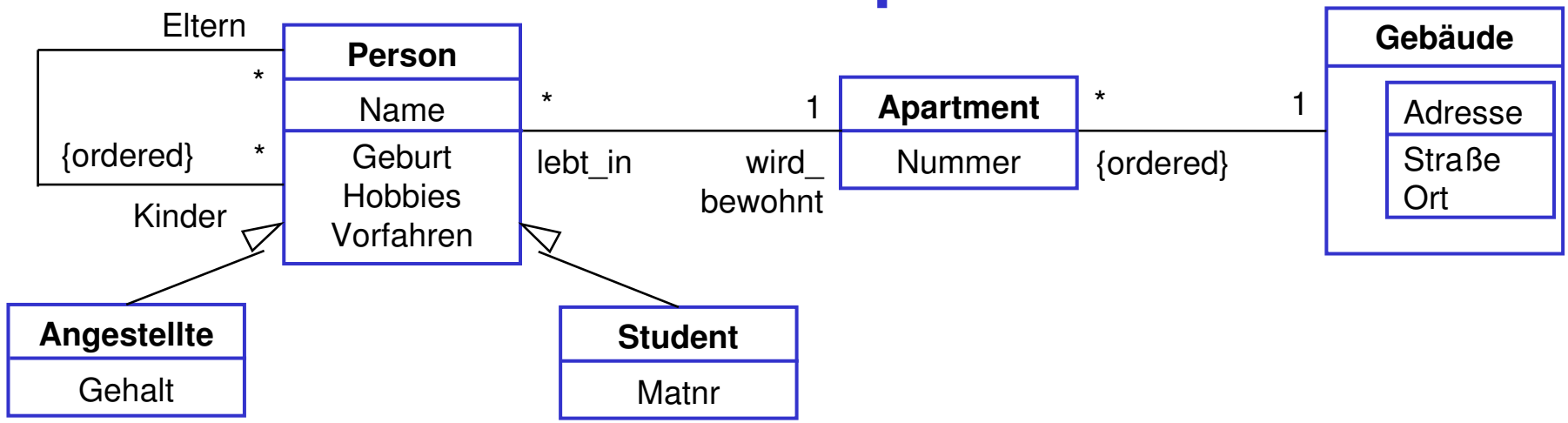
- Instanzen-Properties (Attribute und Relationships) und Operationen (analog zu IDL)

<interface_body>	::=	<export> | <export> interface_body>
<export	::=	<type_dcl> | <const_dcl> | <except_dcl> | <attr_dcl> | <rel_dcl> | <op_dcl>
<type_dcl>	::=	**typedef** <type_spec_declarators> | <struct_type> | <union_type> | <enum_type>
<struct_type>	::=	**struct** <identifier> { <member_list> }
<const_dcl>	::=	**const** <const_type> <identifier> = <const_exp>
<except_dcl>	::=	**exception** <identifier> { [<member_list>] }
<attr_dcl>	::=	**[readonly] attribute** <domain_type> <identifier> [<fixed_array_size>]
<rel_dcl>	::=	**relationship** <target_of_path> <identifier> **inverse** <inverse_path>
<target_of_path>	::=	<identifier> | <rel_collection_type> < <identifier> >
<inverse_path>	::=	<identifier> :: <identifier>
<op_dcl>	::=	[**oneway**] <op_type_spec> <identifier> ([<param_dcl_list]>) [**raises** (exception_list)]
<op_type_spec>	::=	<simple_type_spec> | **void**
<param_dcl_list>	::=	<param_dcl> | <param_dcl>, <param_dcl_list>
<param_dcl>	::=	<param_attribute> <simple_type_spec> <identifier>
<param_attribute>	::=	**in | out | inout**

- Beispiel

```
class Professor extends Person ( extent Profs;   keys PNR, (Name, Gebdat) )
{       struct Kind {string Kname, unsigned short Alter};
        attribute string Name;
        attribute unsigned long PNR;
        attribute date Gebdat;
        attribute set<Kind> Kinder;
        relationship ABT Abteilung inverse ABT::Abt_Angehörige
        relationship set<Prüfung> Prüfungen inverse Prüfung::Prüfer;
        void Neue_Pruefung (in Prüfung p);
        unsigned short Alter () raises (Geb_unbekannt);
        ... }
```

ODL-Beispiel




```
class Person (extent PERS) {
  attribute string Name
  relationship Apartment lebt_in inverse Apartment::wird_bewohnt_von;
  relationship set<Person> Eltern inverse Person::Kinder;
  relationship List<Person> Kinder inverse Person:: Eltern;
  void Geburt (in Person Kind);
  set<string> Hobbies();
  set<Person> Vorfahren ();};

class Gebäude{
  attribute struct Adresse {string Straße, string Ort};
  relationship List<Apartment> Apartments inverse Apartment::Haus};

class Apartment{
  attribute short Nummer;
  relationship Gebäude Haus inverse Gebäude::Apartment
  relationship set<Person> wird_bewohnt_von inverse Person lebt_in; };
```

```
class Angestellte extends Person{
  attribute float Gehalt; };

class Student extends Person{
  attribute string Matnr; };
```


DBS 2

Spracheneinbettung C++

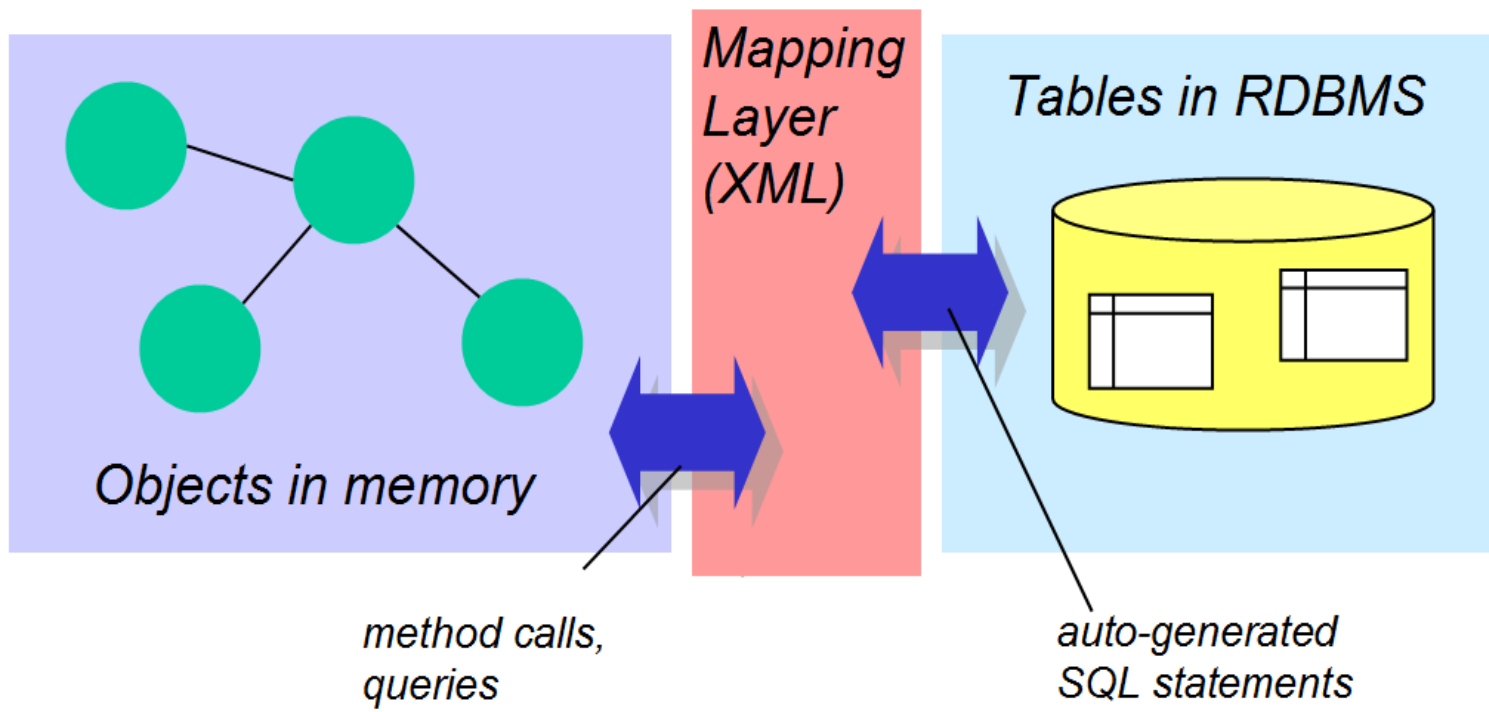
- Realisierung der ODL in C++
  - nur ein zusätzliches Sprachkonstrukt für Beziehungen (neues Schlüsselwort inverse)
  - Klassenbibliothek mit Templates für Kollektionen etc.
  - Ref-Template d_Ref<T> für alle Klassen T, die persistent sein können

```
class Person {
   public:   String Name;
   d_Ref<Apartment> lebt_in
                inverse wird_bewohnt_von;
   d_Set  < d_Ref<Person> > Eltern inverse Kinder;
   d_List < d_Ref<Person> > Kinder inverse Eltern;
   // Methods:
   Person(); // Konstruktor
   void Geburt ( d_Ref<Person> Kind);
   virtual d_Set<String> Hobbies();
   d_Set< d_Ref<Person> > Vorfahren();
   };
class Angestellte: Person {
   public: float Gehalt;
   };
class Student: Person{
   public: string Matnr; };
```

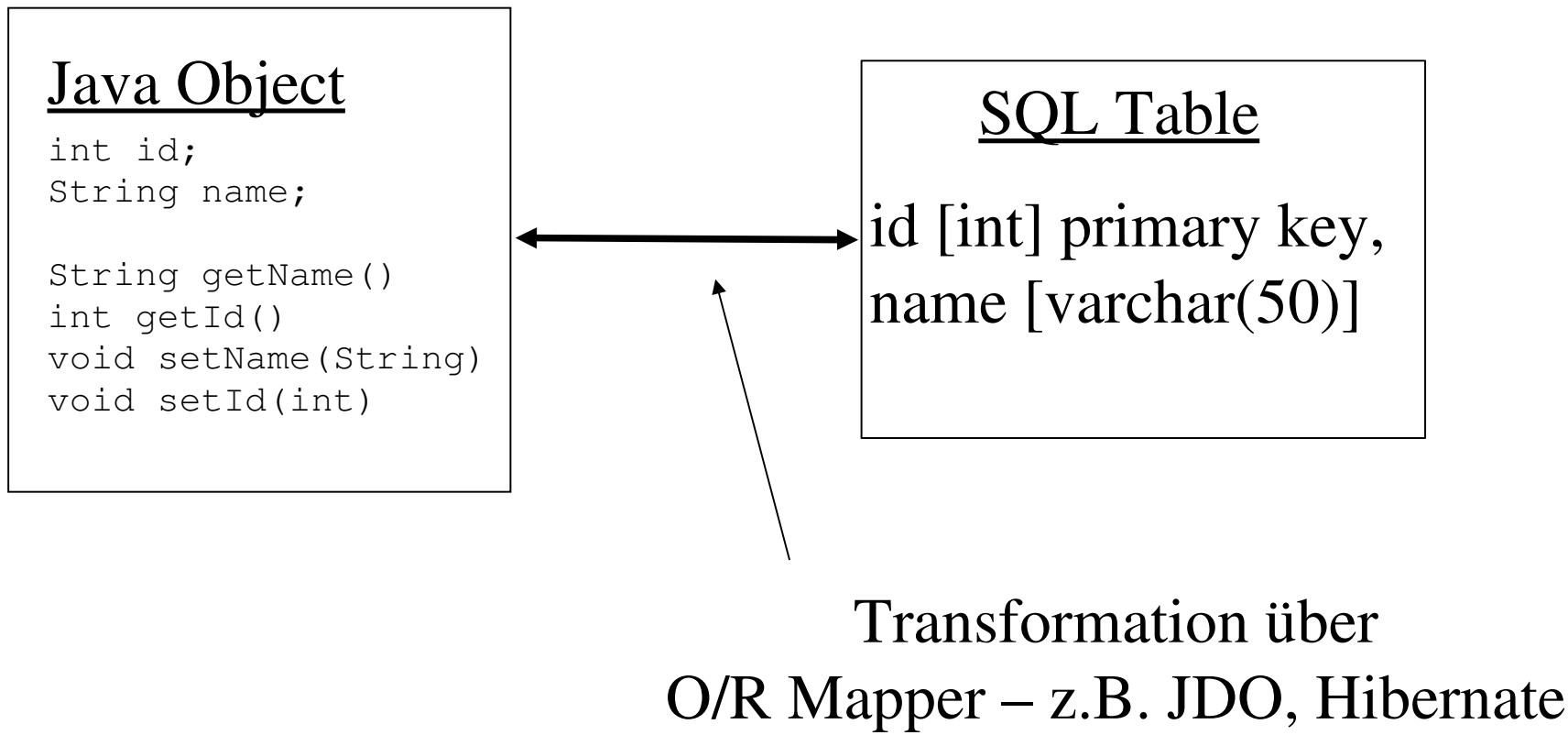
```
class Adresse {
      String Straße;
      String Ort;
   };
class Gebäude {
      Adresse adresse;
      d_List< < d_Ref<Apartment> > Apartments
            inverse Haus;
   };
class Apartment {
      int Nummer;
      d_Ref<Gebäude> Haus inverse Apartments;
      d_Set <d_Ref<Person>> wird_bewohnt_von
            inverse lebt_in;
      };

d_Set <d_Ref<Person>>  Pers;    // Personen-Extent
d_Set <d_Ref<Apartment>> Apartments;
```

O/R Mapping Frameworks

- Zwischenschicht zur Abbildung zwischen objektorientierten Anwendungen und relationalen Datenbanken (O/R-Mapping)
  - komfortablere Abbildung zwischen Anwendungsobjekten und Datenbank als mit SQL-Einbettung / CLI (z.B. JDBC), u.a. für komplexe Objekte
  - einheitliche Manipulation transienter und persistenter Objekte
  - größere Unabhängigkeit gegenüber DB-Aufbau



Object/Relational Mapping

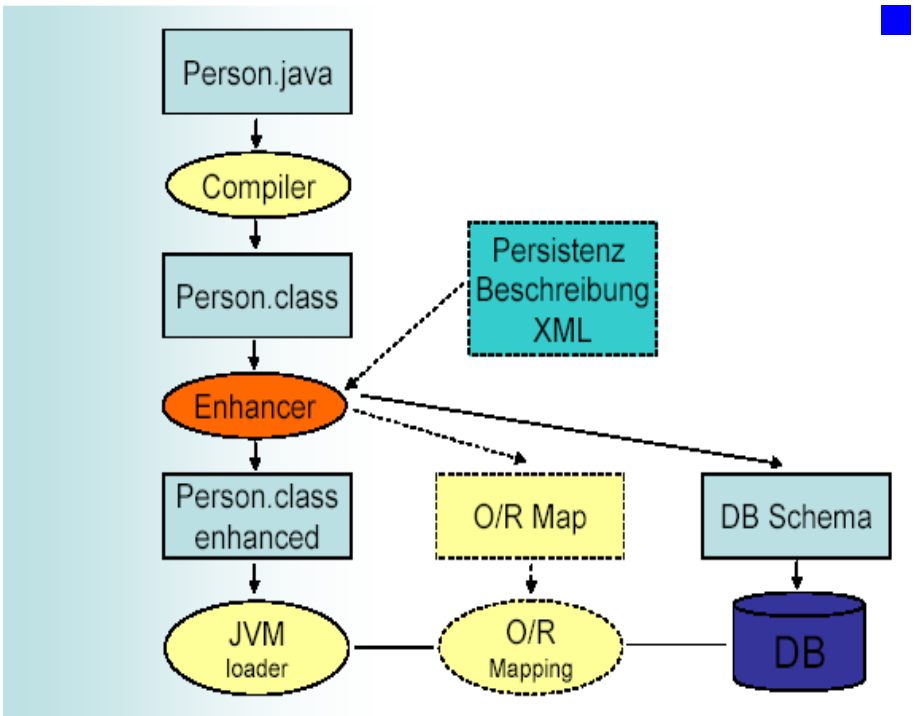


O/R Mapping Frameworks

- Anforderungen
  - Flexibles Mapping für Vererbungshierarchien, komplexen Objekten, …
  - Persistenzverwaltung
  - Transaktionsverwaltung
  - Query-Unterstützung
  - Caching
- Unterstützung innerhalb von Software-Entwicklungsarchitekturen
  - J2EE: Enterprise Java Beans (EJB)
  - .NET-Framework
- leichtgewichtigere Frameworks
  - Java Data Objects (JDO)
  - Hibernate

Java Data Objects (JDO)

- API zur persistenten Speicherung von Java-Objekten
- transparente Verwendung unterschiedlicher Speichertechniken
  - relationale DBS, objektorientierte DBS oder Dateisysteme
- Unterstützung von Queries (JDOQL) und Transaktionen
  - beschränkte Queries: Filterausdrücke auf 1 Klasse und Subklassen
  - Klassenübergreifende Auswertungen über Pfadausdrücke



- Mappings zu relationalen Datenbanken
  - Forward Engineering: Generierung von Tabellen aus Java-Klassen
  - Reverse Engineering: Generierung von Java-Klassen aus Tabellen
  - JDO-Enhancer führt erforderliche Mappings für persistent zu speichernde Java-Objekte durch (z.B. Java - relational)

www.jdocentral.com

Hibernate

HIBERNATE

- Open-Source-Framework zum O/R-Mapping für Java-Objekte
  - .NET-Unterstützung über NHibernate
- gleichartige Verarbeitung transienter und persistenter Objekte
- flexible Mapping-Optionen über XML-Konfigurationsdateien
  - Vererbung: table-per-class, table-per-class-hierarchy, table-per-subclass
  - explizite Unterstützung für 1:1, 1:n, n:m-Beziehungen
- Query-Sprache HQL (SQL-Anfragen weiterhin möglich)
- Caching-Optionen (session, shared, distributed)
- Lese/Ladestrategien („lazy loading“ von Objektmengen)
- Schreibstrategien (WriteOnCommit, BatchUpdate)
- Locking: optimistisch (timestamp) oder pessimistisch

www.hibernate.org

Hibernate Mapping: Beispiel

Person.java

```
public class Person {
    private java.util.Date birthday;
    private Name name;
    private String key;
    public String getKey() {
        return key;
    }
    private void setKey(String key) {
        this.key=key;
    }
    public java.util.Date getBirthday() {
        return birthday;
    }
    public void setBirthday(java.util.Date birthday) {
        this.birthday = birthday;
    }
    public Name getName() {
        return name;
    }
    public void setName(Name name) {
        this.name = name;
    }
    ......
    ......
}
```

Name.java

```
public class Name {
    char initial;
    String first;
    String last;
    public String getFirst() {
        return first;
    }
    void setFirst(String first) {
        this.first = first;
    }
    public String getLast() {
        return last;
    }
    void setLast(String last) {
        this.last = last;
    }
    public char getInitial() {
        return initial;
    }
    void setInitial(char initial) {
        this.initial = initial;
    }
}
```

person.hbm.xml

```
<hibernate-mapping>
  <class name="Person" table="person"
    <id name="key" column="pid" type="string"
    /> >
          <generator class="native"/>
    </id>
    <property name="birthday" type="date"/>
    <component name="Name" class="Name"
          <property name="initial" />
          <property name="first" />
          <property name="last" />
     </component>
  </class>
</hibernate-mapping>
```

person table

Column Name	Data Type	Length	Allow Nulls
pid	varchar	20	
birthday	datetime	8	✓
initial	char	1	✓
[first]	varchar	50	✓
[last]	varchar	50	✓

DBS 2

Hibernate Anwendung: Beispiel

```
Session s = factory.openSession();
Transaction tx = null;
    try {       tx = s.beginTransaction();
                Name n = new Name()
                Person p = new Person();
                n.setFirst („Robin“); n.setLast („Hood“);
                p.setName(n); …
                s.save(p);
                tx.commit();

                Query q1 = s.createQuery("from Person");
                 List l = q1.list()
                 //Ausgabe ...
      s.close();
     } catch (HibernateException e) {
      e.printStackTrace();
     }}
```

Zusammenfassung

- OODBS (und ORDBS) unterstützen
  - komplexe Objekte und Objektidentität
  - Typhierarchien und Vererbung
  - Erweiterbarkeit bezüglich Datentypen und Verhalten (BDTs, BDFs)
- Kapselung vs. Flexibilität (Ad-hoc-Anfragemöglichkeiten)
- Bewertung OODBS gegenüber ORDBS
  - einheitliche Bearbeitung transienter und persistenter Daten über objekt-orientierte Programmierschnittstelle, operationale Vollständigkeit
  - hohe Leistung für navigierende Zugriffe, z.B. in Entwurfsanwendungen (CAD)
  - unzureichende Funktionalität/Kompatibilität gegenüber SQL (Sichtenkonzept, Autorisierung, ...); geringe Marktbedeutung
- ODMG-Standardisierung
  - Objektmodell mit Objekten / Literalen, Operationen und Properties
  - symmetrische binäre Beziehungen, Kollektionstypen mit Iteratoren
  - standardisierte Schemabeschreibung über ODL, Anfragesprache OQL
- O/R-Mapping: persistente Objekte mit relationalen DB