AutomatisiertePersistenz: Persistenz-Frameworks

Was ist Persistenz?

- Persistenz bedeutet, dass die **Objekte** die Programmausführung **überdauern**
- Praktisch: Objekte werden in einer **Datenbank** "aufbewahrt"
- Auch Dateisystem möglich
- Nicht: Die Objekte holen sich Ihre Infos aus einer Datenbank,
  Sondern: Die Objekte werde aus der Datenbank geholt

- 3-Schichten-Architektur:
  Datenhaltungsschicht muss alle relevanten Objekte dauerhaft abspeichern – so dass sie das Programmende überleben
- Außerdem können bei großen Anwendungen nicht alle Objekte speicherresident sein
  → Zwischenspeicherung in der DB.

Aufgaben des "Persistenzmechanismus"

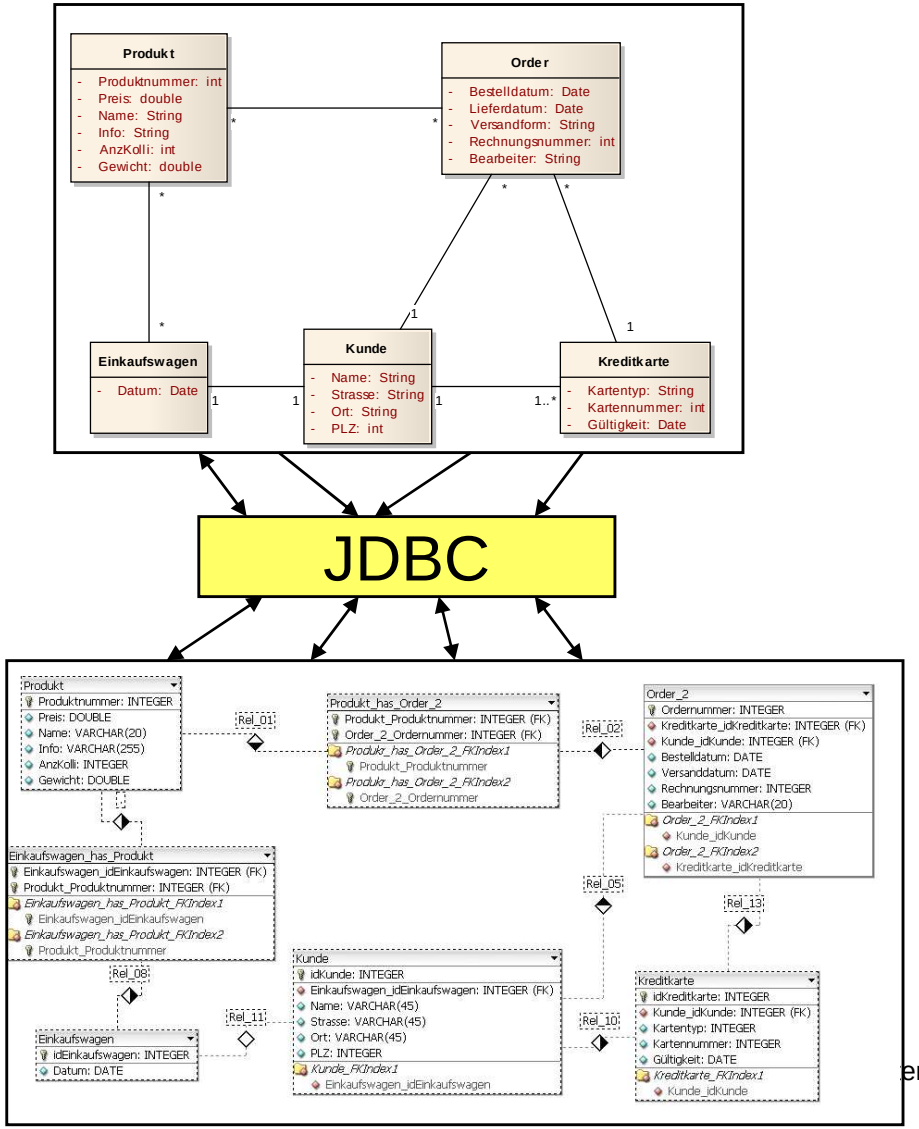
- **OR-Mapping:**
  - Wie werden Objekte auf Datenbanktupel abgebildet?
  - Genauer: Wie wird das Objektmodell auf das Datenmodell abgebildet?

- **DB-Connection:**
  - Wie baut man die Verbindung zur richtigen Datenbank auf?
  - Wie werden Objektanfragen in SQL-Anfragen übersetzt?
  - Wie werden Anfragen ausgeführt und das Ergebnis interpretiert?

- **Objektstruktur laden**
  - Wie werden aus Tupeln Objekte gebaut?
  - Und welche referierten Tupel werden mit geladen?

- **Objektsynchronisation**
  - Wie werden externe und interne Objekte synchron gehalten?

Direkte Datenbank-Anbindung per JDBC

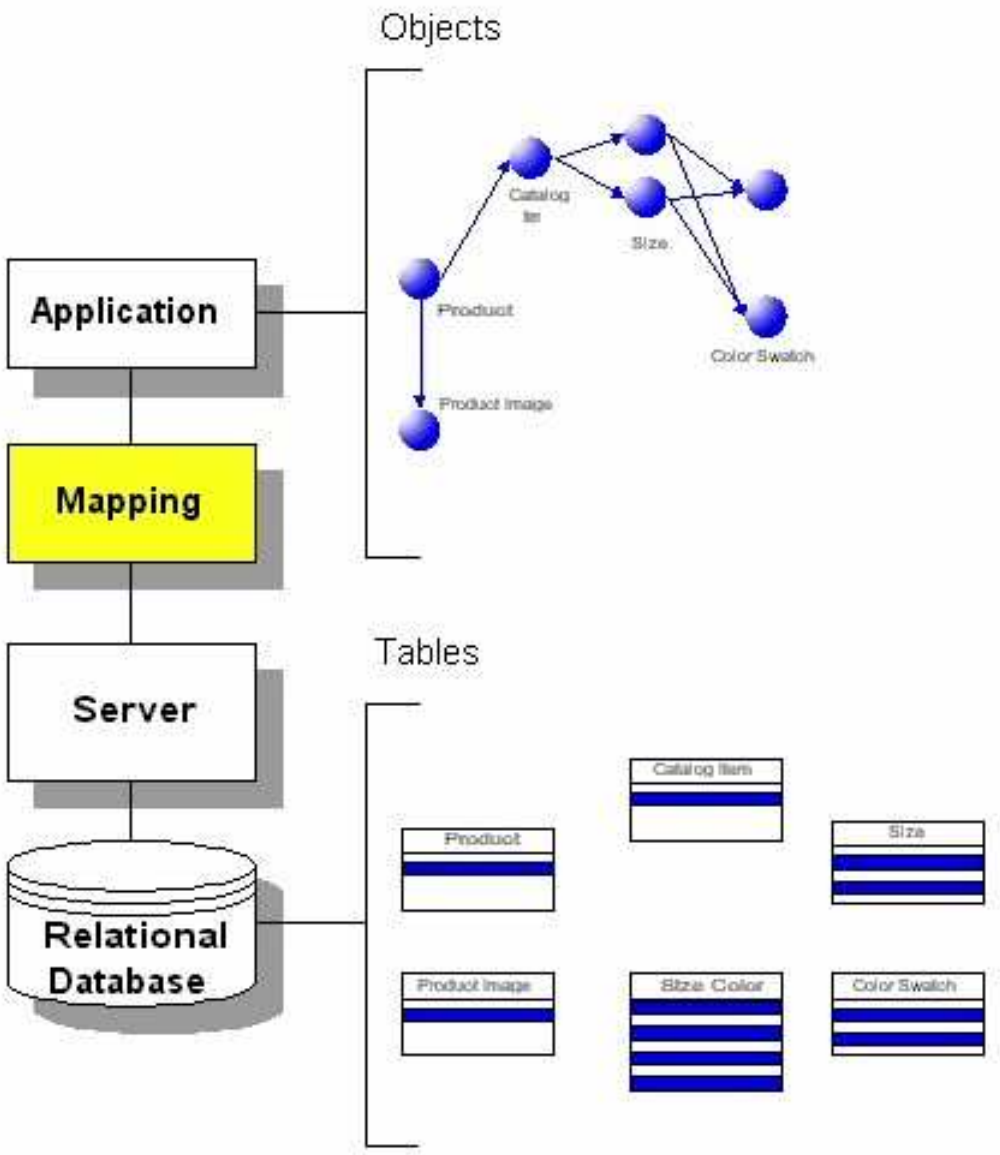


- Beziehung zwischen Klassen und Tabellen nicht erkennbar
- SQL-Verwendung: Datenbank enthält "Daten", nicht Objekte

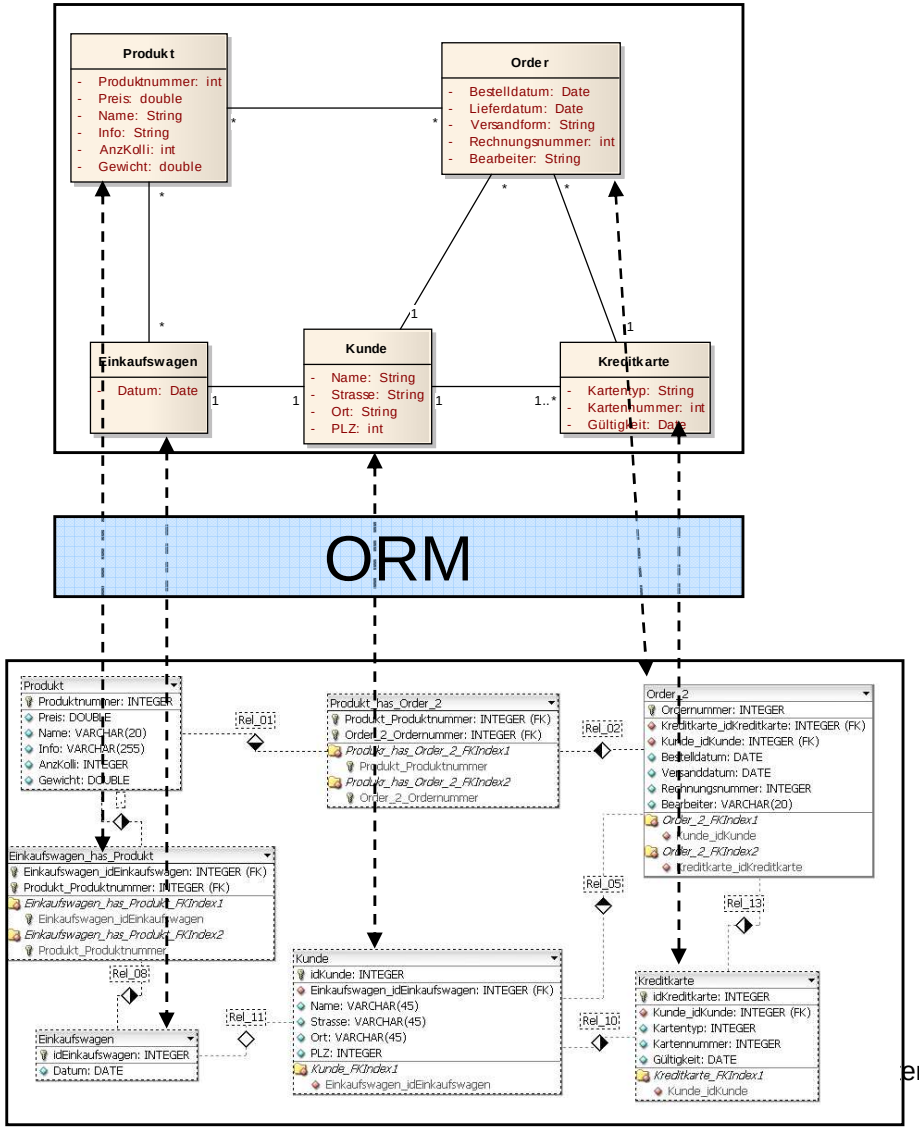
OR-Mapping – auf dem Weg zum Traum

OR-Mapping

- Abbildungsschema zwischen Objektmodell und Datenmodell
- Entitytyp (~"Tablelle") und Klasse sind verwandte Konzepte
- Abbildung "einfacher Objekte", d.h. von Objekten, die keine weiteren Objekte enthalten / referieren, ist trivial:
  - Klasse wird Tabelle
  - Attribute werden Tabellenattribute
- Komplexe Objekte und Objektbeziehungen erfordern komplexere Abbildung



ORM



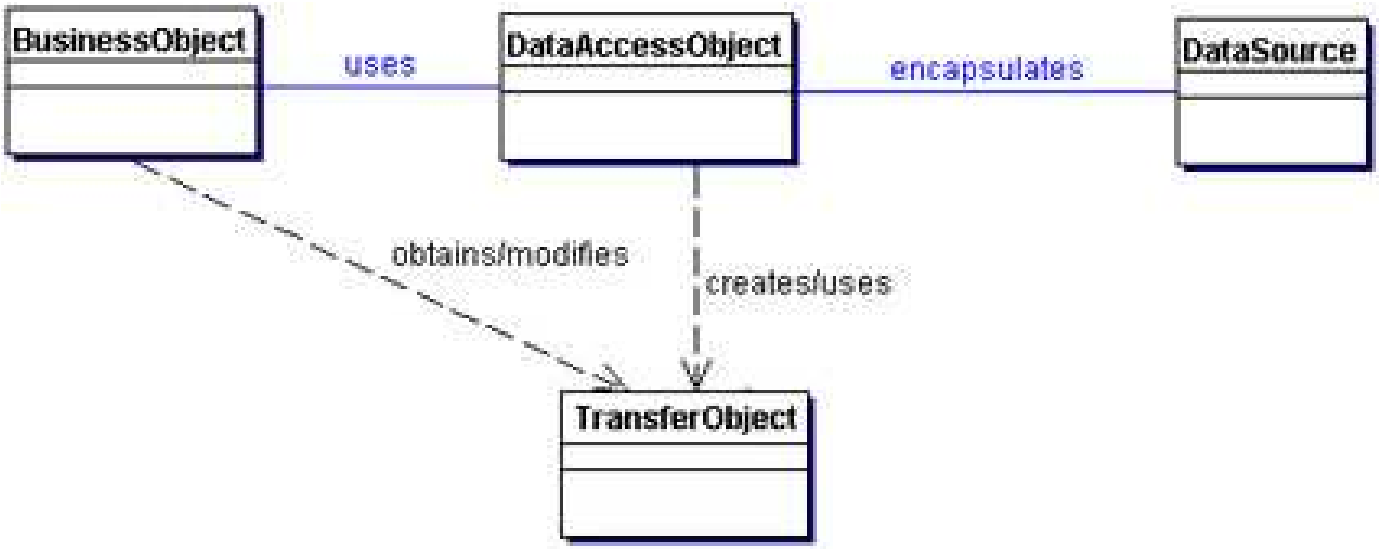
- ORM liefert die Abbildung zwischen Klassen und Tabellen (-gruppen)
- Query Language ermöglicht Suche auf Objektebene
- Es gibt keine DML, sondern Objekte werden manipuliert und dann persistiert

Direkte und indirekte Persistenz

- Direkte Persistenz:
  - jedes persistente Objekt enthält Methoden, um sich selbst zu materialisieren / dematerialisieren / synchronisieren.
  - Wissen über den DB-Zugang und den entsprechenden Teil des Datenmodells sind in jeder Klasse vorhanden.
  - DB-Zugang, Datenmodell und Persistenzstrategie schwer änderbar.

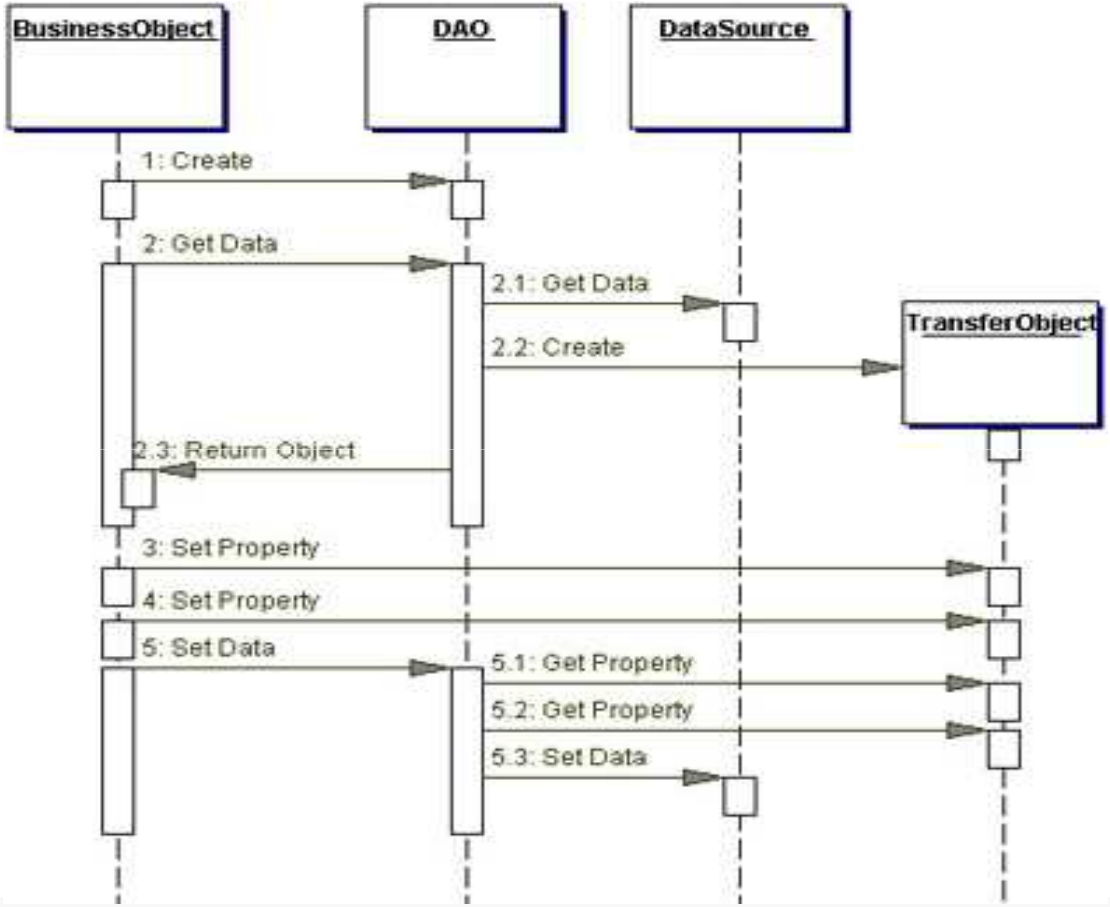
- Indirekte Persistenz
  - Persistenzobjekte sind den Fachobjekten zugeordnet
  - Materialisierung / Dematerialisierung / Synchronisation erfolgt "von außen" als Service für die Fachobjekte.
  - Voraussetzung für transparente Persistenz.

DAO / DTO – minimaler Persistenzapparat



- JEE-Entwurfsmuster
  - DAO – Data Acces Object
  - DTO – Data Transfer Object
- Zweck:
  - indirekte Persistenz.

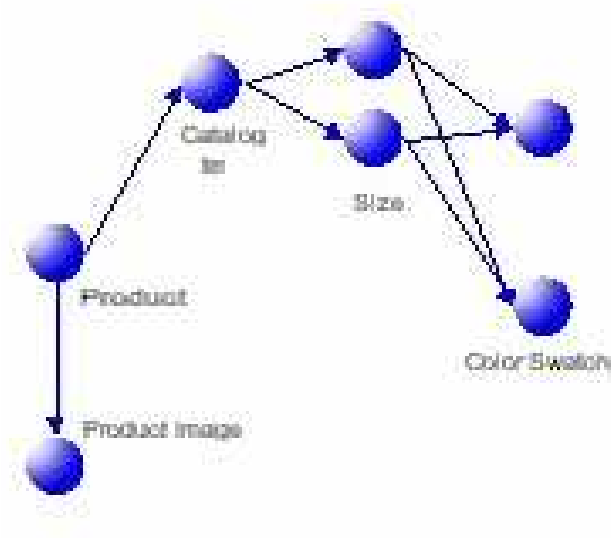
Arbeitsweise des DAO/DTO-Musters



- DAO materialisiert ein DTO, auf das das Fachobjekt zugreift.
- Dematerialisierung bedeutet Zurückschreiben des DTO.
- DAO kennt die Datenquelle (DB) und das ORM.

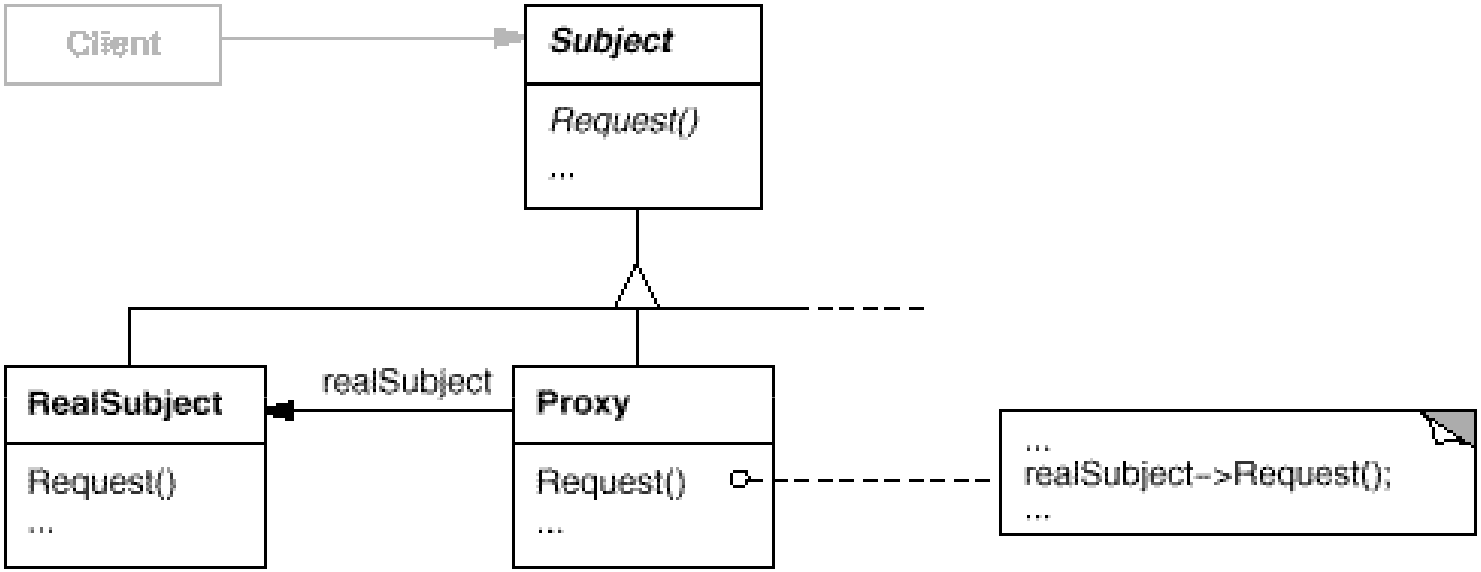
OR-Mapping – Lazy Materialization

- Laden externer Objekte:
  - Was ist mit den assoziierten Objekten?
  - Objektstruktur kann stark verzweigen...



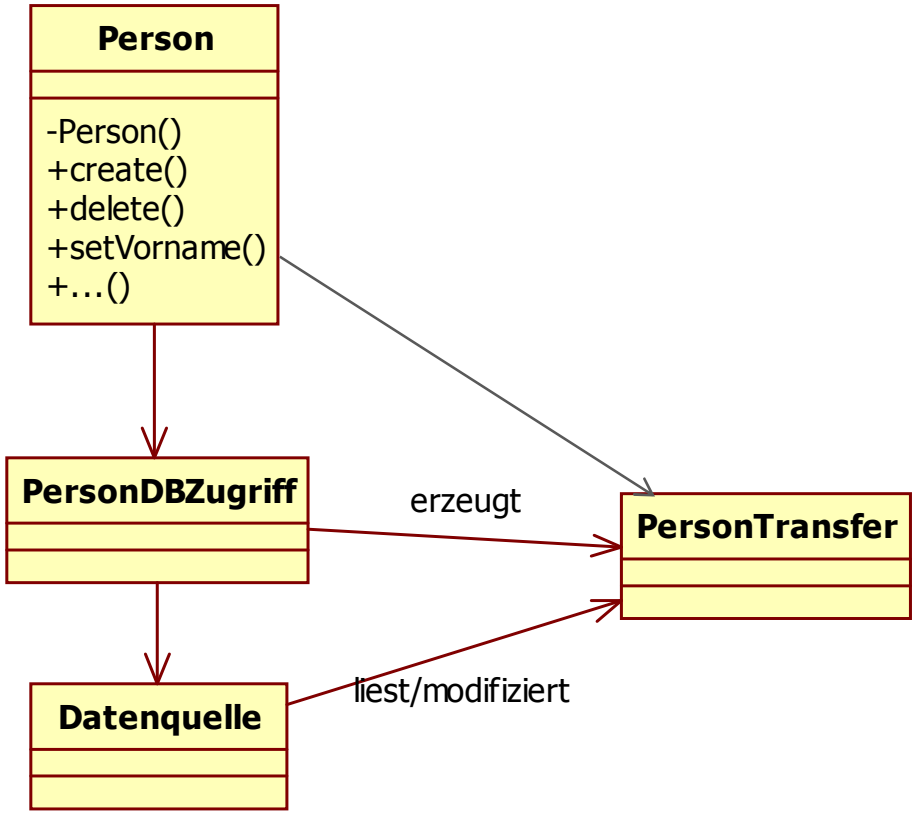
- Proxy-Entwurfsmuster:
  - Objekt erreichbar über ein Stellvertreterobjekt (Proxy)
  - Proxy liegt im Speicher
  - besorgt das Nachladen des Objekts bei Bedarf
  - enthält das OID

Proxy-Entwurfsmuster



- Objektassoziation des Client verweist auf einen Platzhalter
  - Proxy,
  - imitiert Struktur des realen Subjekts.
- Anfrage (request) erzeugt Bedarf:
  - Das reale Subjekt wird geladen
  - und die Anfrage delegiert.

DAO / DTO - Beispiel



Person
ist reales Objekt
im Proxy-Muster

DB-Zugriff ist als
DAO realisiert.

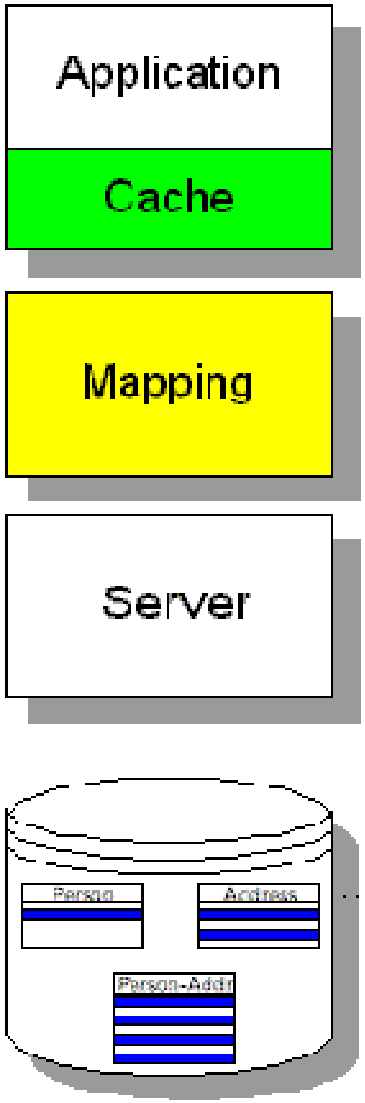
DAO-Code (vereinfacht)

```
public class PersonDBZugriff {
   private Connection con;

   public PersonDBZugriff()
   {    con = getJDBCConnection(myDB, myUser, myPw);  }

   public Person getPerson(int OID)throws SQLException {
        String name, vorname;
        Statement stmtGet = con.createStatment();
        ResultSet res = stmtGet.executeQuery
                ("Select name, vorname from Person"+
                 "where Person.oid= :OID");
        name = res.name;
        vorname = res.vorname;
        stmtGet.close();
        return new Person(name, vorname);  // DTO
}
```

DAO-Code (vereinfacht)

```
public void putPerson(Person person)throws SQLException {
     Statement stmt = con.createStatment();
     ResultSet res = stmt.executeQuery
        ("Select name from Person where Person.oid= :OID");
     String sql;
     if (res==null)
        sql = "Insert into Person (OID, name, vorname)"+
            "Values ("+ person.getOID() + "," +
                        person.name() + ","  +
                        person.vorname() + ");";
     else
         sql = "Update Person set name = " + person.name +
               "vorname = " + person.vorname +
               "where OID = " + person.oid;
     stmt.executeUpdate(sql);
     stmt.close();
}
}
```

Der Traum: Transparente Persistenz



- Animation unter http://www.service-architecture.com/object-relational-mapping/articles/transparent_persistence.html
- Abfolge
  - Application will Person benutzen
  - Cache ist leer
  - Mapping holt Person-tupel aus der DB
  - Person wird im Cache aufgebaut
  - Zugriffsversuch auf Adresse
  - fehlt im Cache
  - Bezug aus der DB
  - Update Adresse
  - Rückschreiben in die DB

Der Transparenz-Traum

- *Völlige Transparenz* ☺:
  - *Persistenz ohne Programmierleistung*
  - *Minimaler Konfigurationsaufwand*

- Nicht alle Objekte müssen persistent sein
  - → Klassen als "persistent" deklarieren:

```
persistent class MyClass { ... }
```

- Nicht alle Attribute müssen persistent sein:

```
persistent int myAttribute;
```

- Alles andere könnte "automatisch ablaufen"

Persistenzframeworks

- Hibernate
  - Der Klassiker (~2002) *Gavin King & Team*
  - Bekannt als ORM-Framework
  - Konzept- und Begriffsbilder (Persistenz, Entity, …)

- JDO - Java Data Objects
  - Spezifikation, einige Referenz-Implementierungen
  - Vereinfachte Persistenz von POJOS
  - D.h. persistente Objekte müssen nichts implementieren

- JPA – Java Persistence API
  - Modernstes Konzept, Spezifikation
  - Bestandteil von JEE (EJB 3.0) – aber herauslösbar
  - Ebenfalls POJO-Persistenz

ORM mit Hibernate

- Sehr frühes ORM-API, etabliert und verbreitet
- Ziel: Abbildung unsichtbar machen
  - DB-Zugang kapseln
  - Zusammenhang zw. Klassen und Tabellen kapseln
  - Statt SQL-Abfragen Objektabfragen
- Funktionsweise:
  - Hibernate-Session-Objekt ist die "Kapsel"
  - Wird DB-spezifisch generiert (XML-Spezifikation)
  - Persistente Objekte werden in XML spezifiziert
  - Session-Methoden arbeiten mit Objekten:
    - *öffnen, schließen: open, close*
    - speichern: save, update, flush
    - abfragen: find; createQuery, list, execute
    - löschen: delete

Hibernate-Deskriptoren

- Konfiguration des DB-Zugangs in hibernate.cfg.xml
- Abbildung auf das relationale Modell per Konfiguration in einzelnen Mapping-Dateien MyClass.hbm.xml
  - Zuordnung Klasse-Tabelle
  - Zuordnung Attribut-Attribut
  - Zuordnung Beziehung (one-to-one, one-to-many,...)

hibernate.cfg.xml

```
<!DOCTYPE …
<hibernate-configuration>
 <session-factory>
      <!-- Database connection settings -->
      <property name="connection.driver_class">org.hsqldb.jdbcDriver
     </property>
      <property name="connection.url">
        jdbc:hsqldb:file:/Users/Data/work/hsqldb/onlineshop/datastore;
        shutdown=true
    </property>
    <property name="connection.username">shop</property>
    <property name="connection.password">pohs</property>
     ….
    <!-- Hibernate XML mapping files -->
    <mapping resource="Produkt.hbm.xml"/>
    <mapping resource="Order.hbm.xml"/>
      ….
 </session-factory>
</hibernate-configuration>
```

Produkt.java

```
public class Produkt {
    private Long id;
    private String name, info;
    private Double preis, gewicht;
    private Collection<Order> hasOrder = new HashSet<Order>();

    // getter und setter
}
```

Produkt.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
      "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
      <class name="onlineshop.model.Produkt" table="produkt">

                  <id name="id" type="long" column="id">
                              <generator class="native"/>
                  </id>

                  <property name="preis"/>
                  <property name="name"/>
                   <property name="info"/>
                   <property name="gewicht"/>

                  <set name="hasOrder" table="produkt_has_order">_
                              <key column="user_id"/>
                              <many-to-many class="hibernateDemo.model.Order"/>
                  </set>
      </class>
</hibernate-mapping>
```

Benutzung von Hibernate:

```
SessionFactory sf = Context.getSessionFactory();
Session session = sf.openSession();

// Objekte laden, verändern
Produkt pr = (Produkt) session.find
      ("from Produkt in class onlineshop.model.Produkt " +
       "where name = ?" , "Schrauberset XXL", Hibernate.STRING);
pr.gewicht += 0.6;   //  neue Verpackung;
// neues Objekt erzeugen und persistieren
Thing thing = new Thing();
session.save(thing);

// alles speichern, auch das aktualisierte pr
session.flush();
session.close();
```

Queries in Hibernate

```
public class ShopServlet extends HttpServlet {
    …
    private Produkt findeProdukt(String name) throws Exception{
            Produkt produkt=null;
            ServletContext servCon=this.getServletContext();
            SessionFactory sessFac=(SessionFactory) servCon.getAttribute("sessionFactory");

            if(sessFac!=null){
                    Session sess=sessFac.openSession();
                    Query q = sess.createQuery
                        ("from Produkt in class onlineshop.model.Produkt "+
                         "where name= :name ");
                    q.setString("name",name);
                    List list=q.list();
                    if  (list!=null && list.size()!=0)   produkt =(Produkt)list.get(0);
            }
            return produkt;
    }
}
```

Was leistet Hibernate

- Automatisierte OR-Abbildung
  - Mapping-Beschreibung als XML-Datei
  - DB-Schema kann generiert werden
  - Mapping kann aus DB-Schema generiert werden
- Abfragesprache
  - find
  - createQuery, list, execute
- Cashing der gelesenen Objekte
  - Verwaltung in der Hibernate-Session
  - Benutzergesteuertes Schreiben durch save, update, flush (Objektbaum bei save und update, gesamter Cache bei flush)

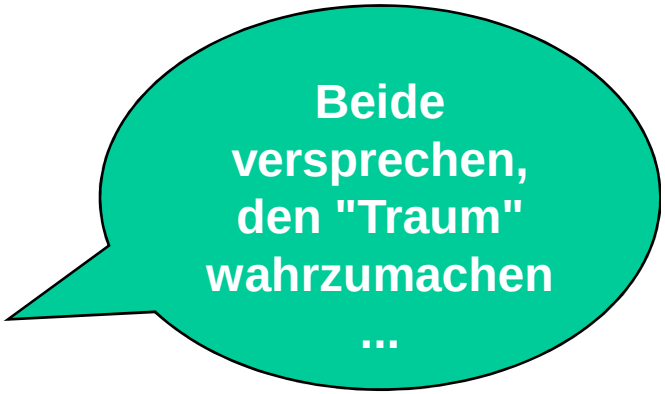
Was fehlt?

- Persistenzverwaltung für beliebige Objekte (nicht nur gelesene).
- Lazy fetching
- *Die Mapping-Beschreibung ist aufwändig*

Persistenz-Automatisierung: JPA und JDO

- POJO = Plain Old Java Object ☺
  in Abgrenzung zu EJB

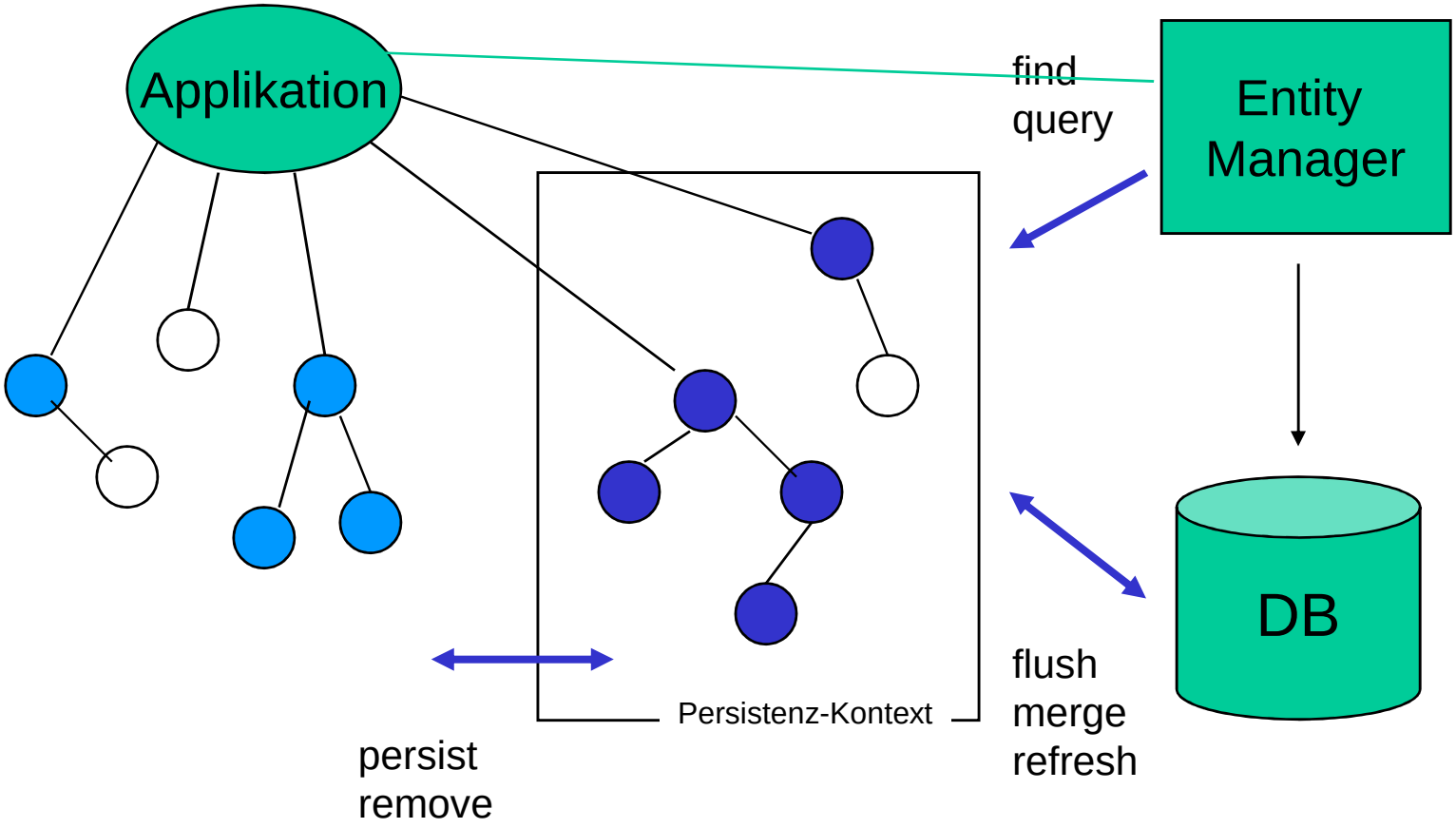
- Zwei konkurrierende Spezifikationen:
- JDO – Java Data Objects
  (seit 2001)
- JPA - Java Persistence API
  (seit EJB 3.0 aus J2EE ausgelagert)



Ausgelagert aus EJB 3.0: JPA

- Java Persistence API
  - Persistenzmodell der EJB 3.0-Spezifikation
- Jede Java-Klasse kann "Persistent Entity" sein
  - wenn sie "will"
- Java-5-Annotationen spezifizieren
  - Persistenz
  - ORM
  - Umgang mit Abhängigkeiten
  - Ladeverhalten
- Persistence Provider - die JPA-Implementierung
- Persistence Unit - das ORM und die DB-Anbindung
- Persitence Context - der Cache der verwalteten Objekte
- Entity-Manager - verwaltet die ‚Objekte im Cache

Entity-Manager



Persistenz-Spezifikation

```
@Entity
public class User {
    private String name;
    private String address;
    @Transient
    private int SessionId;
    ...
}
```

Zugang zum Persistenz-Kontext:

- Referenz auf EntityManager kann durch eine Factory erzeugt werden.
- Der zugehörige Persistenz-Kontext gilt für die Lebensdauer des EntityManagers (Extendend)

```
private EntityManager entityManager =
        EntityManagerFactory.createEntityManager()
```

- In J2EE gilt ein Persistenz-Kontext (per default) für eine Transaktion.
- Referenz auf Entity-Manager wird "injiziert" (Dependency injection), d.h. von der Umgebung (automatisch) gesetzt:

```
public class UserRegistration {
  @PersistenceContext
  private EntityManager entityManager;
  ...
}
```

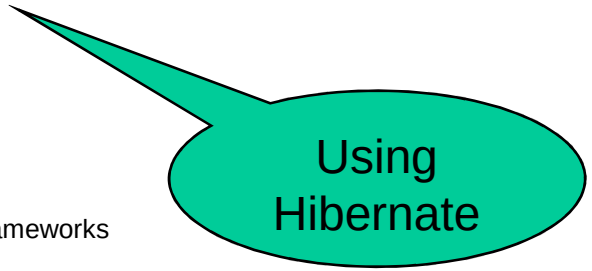
Arbeiten mit dem Persistenz-Kontext

- Instanz (erstmalig) in den Persistenz-Kontext aufnehmen:
  ```
  Artikel bitsatzCC = new Artikel();
  bitsatzCC.setName("CustomConformityBits");
  entityManager.persist(bitsatzCC);
  ```
- Aktuellen Zustand in die DB schreiben:
  ```
  entityManager.merge(bitsatzCC);
  ```
- Instanz mit DB-Werten aktualisieren:
  ```
  entityManager.refresh(bitsatzCC);
  ```
- DB-Eintrag löschen (die Instanz lebt weiter!):
  ```
  entityManager.remove(bitsatzCC);
  ```

JPA-Implementierungen

- Standalone-Implementierung auf JDBC-Basis: **Toplink**
  Referenz-Implementierung ist in Glassfish integriert.
- **Hibernate EntityManager** (mit der Erweiterung Hibernate Annotations) ist ebenfalls eine JPA-Implementierung
  - eigentlich eher das Vorbild für JPA ☺
  - Das Session-Objekt ist der Persistenz-Kontext.
  - Der EntityManager verwaltet das Session-Objekt

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("mysql") ;
EntityManager em = emf.createEntityManager();
em.persist(someEntity);
…
em.close(); ...
//close at application end
emf.close();
```



Konfigurierung von JPA

- OR-Mapping wahlweise per Annotation oder XML-Deskriptor
- Ladeverhalten konfigurierbar:

```
@Entity
@FetchType = LAZY
public class Produkt {....}
```

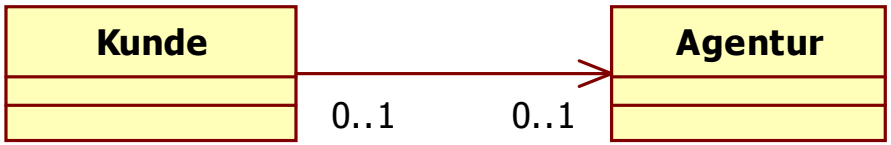
- Persistenz-Kaskadierung

ORM: Attributspezifikation per Annotation

- @Id
  - Primärschlüssel
  - kann generiert werden:
    @Id
    @GeneratedValue
- @Basic
  - Basisdatentyp
- @LOB
  - BLOB oder CLOB nach Bedarf
- @Temporal
  - Zeitwert
- @Embedded
  - eigebettetes Objekt (strukturiertes Objekt mit 1:1-Beziehung)

ORM: Beziehungsspezifikation per Annotation

Unidirektionale 1:1-Beziehung




```
@Entity
public class Kunde {
    private int id;
    private Agentur agentur;
    ...
    @OneToOne
    @JoinColumn(name="AGENTUR-ID")
    public Agentur getAgentur() { return agentur; }
}
```

ORM: Beziehungsspezifikation per Annotation

Bidirektionale 1:1-Beziehung


```
@Entity
public class Kunde {
    private int id;
    private Agentur agentur;
    ...
    @OneToOne
    @JoinColumn(name="AGENTUR-ID")
    public Agentur getAgentur()
    { return agentur; }
}
```

```
@Entity
public class Agentur {
    private int id;
    private Kunde kunde;
    ...
    @OneToOne(mappedBy="agentur")
    public Agentur getKunde()
    { return kunde; }
}
```

ORM: Beziehungsspezifikation per Annotation

Unidirektionale 1:N-Beziehung


```
@Entity
public class Agentur {
    private int id;
    private Collection<Kunde> kunden;
     ...
     @OneToMany(cascade=CascadeType.ALL)
     public Collection<Kunde> getKunden()
     { return kunden; }
    ....
}
```

```
@Entity
public class Kunde {
     private int id;
              ...
}
```

ORM: Beziehungsspezifikation per Annotation

Bidirektionale 1:N-Beziehung


```
@Entity
public class Agentur {
    private int id;
    private Collection<Kunde> kunden;
     ...
     @OneToMany(cascade=CascadeType.ALL
                        mappedBy="agentur")
     public Collection<Kunde> getKunden()
     { return kunden; }
    ....
}
```

```
@Entity
public class Kunde {
     private int id;
     private Agentur agentur;
          ...
     @ManyToOne
     @JoinColumn(name="AGENTUR_I
     public Agentur getAgentur()
     { return agentur; }
}
```

ORM: Beziehungsspezifikation per Annotation

Bidirektionale M:N-Beziehung

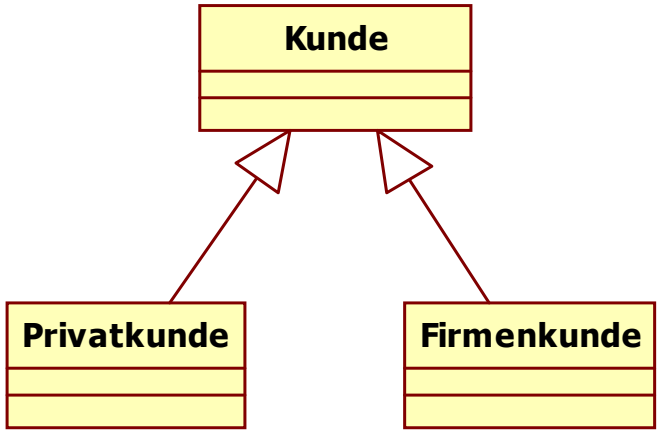
```
@Entity
public class Agentur {
    private int id;
    private Collection<Kunde> kunden;
     ...
    @ManyToMany
    @JoinTable(name="AGENTUR_KUNDE",
          joinColumns=
             @JoinColumn(name="AGENTUR_ID",
                   referencedColumnName="ID"),
          inverseJoinColumns=
              @JoinColumn(name="KUNDE_ID",
                   referencedColumnName="ID") )
    public Collection<Kunde> getKunden()
    { return kunden; }
   ....  }
```


```
@Entity
public class Kunde {
     private int id;
     private Collection<Agentur>
                                agenturen;
          ...
    @ManyToMany (mappedBy=kunden)

     public Collection<Agentur>
                    getAgenturen()
    { return agenturen; }
}
```

ORM: Vererbungsabbildung per Annotation

InheritanceType

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="TYP",
      discriminatorType=CHAR)
@DiscriminatorValue("K")
public abstrct class Kunde {
      private int id;
      ....
}
```




```
@Entity
@DiscriminatorValue("P")
public abstrct class Privatkunde extends
      Kunde {
...
}
```

```
@Entity
@DiscriminatorValue("F")
public abstrct class Fimenkunde extends
      Kunde {
...
}
```

Persistenz: Kaskadierungspezifikation per Annotation

```
@Entity
public class Agentur {
    private int id;
    private Collection<Kunde> kunden;
     ...
     @OneToMany(cascade= {
                CascadeType.PERSIST, CascadeType.MERGE,
                CascadeType.REFRESH, CascadeType.REMOVE }
          mappedBy="agentur")
     public Collection<Kunde> getKunden()
     { return kunden; }
    ....
}
```

Persistenz: Nachladestrategie per Annotation

```
@Entity
public class Agentur {
    private int id;
    private Collection<Kunde> kunden;
     ...
     @OneToMany(cascade= CascadeType.ALL,
          fetchType=FetchType.LAZY,
          mappedBy="agentur")
     public Collection<Kunde> getKunden()
     { return kunden; }
    ....
}
```

JPA-Nachlese: Persistenz

```
@Entity
public class Agentur {
    @Id
    private int id;
    @OneToMany
    private Collection<Kunde> kunden;
    ...
}
```

Konzeptionell perfekte Transparenz!

- Der Entity-Manager kapselt die Cache-Verwaltung und Synchronisation.
- Die weitergehenden Persistenz-Spezifikationen ermöglichen eine hohe Abstraktion (Cascade und Fetch).

- Der vollautomatische Cache bietet nicht die nötige Performance/Stabiltität
  - Transaktionen oder Benutzereingriffe erforderlich

JPA-Nachlese: ORM per Annotation

- genial: Configuration by Default:
  - Tabellen- und Spaltennamen werden aus den Klassen- und Attributnamen abgeleitet,
  - die Spaltentypen werden aus den Attributtypen abgeleitet,
  - Änderung per Spezifikation möglich

- gut: Beziehungsspezifiaktionen aus Navigationssicht
  - allerdings schlägt hier das Datenmodell direkt durch
  - Beziehungsannotationen überfrachten teilweise den Code

- Alles geht auch per XML-Konfiguration
  - oder auch gemischt: Annotation überschreibt XML-Konfiguration

... und dann gibt es noch JDO

- Zweiter Ansatz für ein Persistenz-API
- älter als JPA, ähnlich
- hohe Transparenz
  - Illusion speicherresidenter Objekte
  - unsichtbare Datenquelle
  - keine Anforderungen / Restriktionen beim Domain Object
- "Registrierung" eines beliebigen Objekts beim PersistenceManager durch Aufruf von makePersistent()
- Domain-Klassen müssen das Interface PersistentCapable implementieren
  - meist "nachgerüstet" durch "Enhancement"
- Konfiguration über Property-Dateien

- Von SUN freigegeben, um konkurrierende Ansätze zu vermeiden
- mehrere Referenzimplementierungen.

...und OO-DB-Persistenz

- ORM entfällt ☺
- Persistenzkonzepte, Caching und Synchronisation bleiben.

Java-ORM-Frameworks (open source)

- Hibernate
- Ibatis SQL Maps
- OJB
- Torque
- Castor
- Cayenne
- TJDO
- JDBM
- Prevayler
- JPOX
- Speedo
- Jaxor
- pBeans
- SimpleORM
- Smyle
- XORM
- O/R Broker
- Mr.Persister
- Java Ultra Light Persistence
- JDBCPersistence
- Ammentos
- Velosurf
- PAT
- daozero
- QLOR
- ODAL
- JPersist
- BeanKeeper
- Open JPA
- Super CSV
- SeQuaLite
- Persist
- ...

http://java-source.net/open-source/persistence

Versprochen,
das war's in Sachen Persistenz!