Konzepte der Datenbanktechnologie

Prof. Dr. U. Hoffmann
FH Wedel

Hibernate 2

das letzte Mal

Hibernate Eigenschaften (Überblick)

- Persistenz
- Vererbung
- Änderungsmanagement
- Detached–Objekte
- HQL–Anfragen
- Criterial–Anfragen

Gliederung

Entwicklungsrichtungen

- Top Down
- Bottom Up
- Middle out
- Meet-in-the-middle

APIs

Mapping

- Entity/Value Types
- Value Types
- Entity Types
- n:1–Assoziationen
- Eltern/Kind–Beziehungen
- 1:1–Assoziationen
- POJOs

Hibernate

- Tools

Entwicklungsrichtungen

Je nach Aufgabenstellung unterstützt Hibernate die Entwicklungsrichtungen:

- **Top Down**
  (beginnend bei Java–Klassen)
- **Bottom Up**
  (beginnend mit dem relationalen Datenbankschema)
- **Middle out**
  (beginnend mit der Mapping-Spezifikation)
- **Meet-in-the-middle**
  (beginnend mit Java–Klassen und dem relationalen Datenbankschema)

Top Down

1. Java Klassen werden entworfen
2. Persistenzinformationen hinzufügen, entweder
   - XML–Mapping spezifizieren, oder
   - Annotationen hinzufügen (XDoclet für 1.4)
3. Schema mit `hbm2ddl` (`SchemaExport`) generieren
4. Schema in die Datenbank exportieren

Bottom Up

1. Mit einer bestehenden Datenbank beginnen
2. Mit `Middlegen` das Datenbank–Schema analysieren
   - XML–Mapping–File generieren
3. (XML–Mapping–File anpassen)
4. Java–Objekte mit `hbm2java` generieren

Middle out

1. XML–Mapping–File schreiben
2. Datenbank–Schema mit `hbm2ddl` (`SchemaExport`) erzeugen
3. Java–Objekte mit `hbm2java` erzeugen.

Meet-in-the-middle

1. Java–Objekte und das Datenbank–Schema liegen vor.
2. Nötige Hibernate–XML–Mapping–Files schreiben
3. Notwenige Anpassungen in den Java–Objekten oder dem relatinalen Modell vornehmen.

Deutlich schwereres Vorgehen, als die anderen

Hibernate APIs

Interface Session

- `Transaction beginTransaction()`
- `Object get(Class c, Serializable id)`
- `void delete(Object o)`
- `Query createQuery(String queryString)`
- `void save(Object o)`
- `Connection close()`
- `...`

Interface Transaction

- `void begin()`
- `void commit()`
- `void rollback()`
- `setTimeout(int seconds)`
- `boolean wasCommited()`
- `...`

Objekte speichern

```
Configuration config = new Configuration();
config.addClass(Egg.class);
SessionFactory sessionFactory =
                    config.buildSessionFactory();
Session session = sessionFactory.openSession();
Transaction tx = null;
try {
  tx = session.beginTransaction();
  Egg egg = new Egg();
  Spam spam = new Spam();
  egg.getSpams().setSpam(spam);
  spam.setEgg(egg);
  session.save(egg);
  tx.commit();
  ....
```

Objekte löschen

```
Configuration config = new Configuration();
config.addClass(Egg.class);
SessionFactory sessionFactory =
                    config.buildSessionFactory();
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
int eggID = 1;
Egg egg = (Egg)session.get(Egg.class, new Long(eggID));
session.delete(egg);
tx.commit();
session.close();
```

Query

Interface Session

- ...
- `Query createQuery(String queryString)`
- ...

```
"select o from C as o"
```

Interface Query

- ...
- `List list()`
- `Iterator iterate()`
- ...
- `setParameter(int position, Object val)`
- ...

Objekt–relationales Mapping

- Die Metadaten für das objekt–relationale Mapping werden in Mapping–Dokumenten in XML notiert.
- Endung: `.hbm.xml`
- Inhalt:
  - Beschreibung der persistenten Daten einer Klasse
  - Beziehung der Klasse zu anderen Klassen.

Mapping–Beispiel

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
  "-//Hibernate/Hibernate Mapping DTD//EN"
  "http://hibernate.sourceforge.net/
                        hibernate-mapping-2.0.dtd">
<hibernate-mapping>
<class name="Eggs" table="EGGS">
<id name="id" type="int">
      <meta attribute="scope-set">protected</meta>
      <generator class="native"></generator>
</id>
<property name="spam" type="string" not-null="true"/>
</class>
</hibernate-mapping>
```

Entitäts– und Werte–Typen

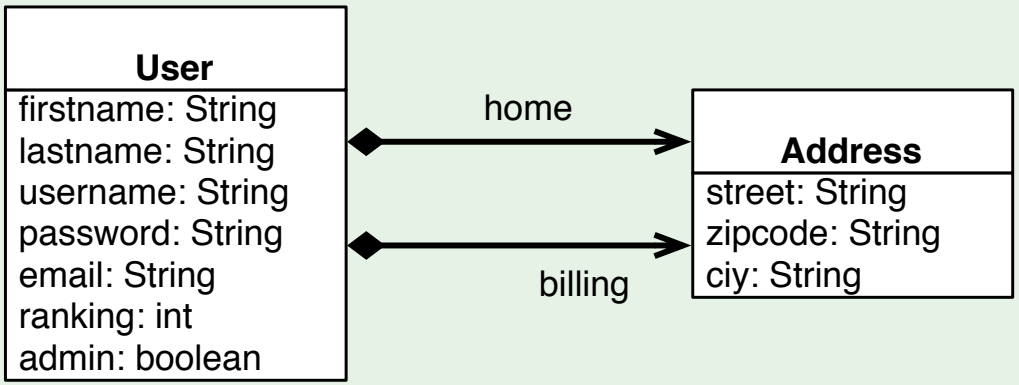
Hibernate unterscheidet grundsätzlich zwischen 2 Arten von Datentypen:

1. **Entity–Objekte**
   - besitzen eine eigene Tabelle
   - sind eigenständige Objekte
2. **Value–Typen**
   Value–Typ–Objekte
   - sind abhängige Objekte
   - werden als Spalten in Tabellen der Entity–Typen gespeichert

Value Types

Beispiel (Benutzer und Adressen)

- Benutzer hat Liefer– und Rechnungsanschrift

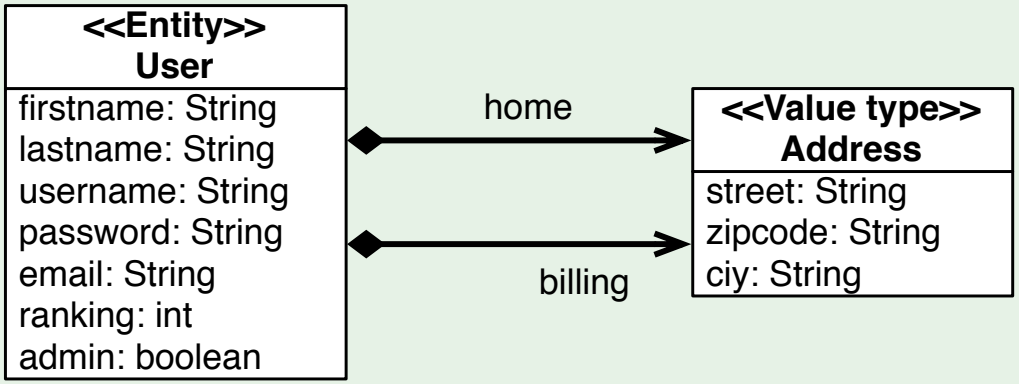


- Lebensdauer der Addressen ist an den jeweiligen User gekoppelt.
- In Java soll das durch User– und Adress–Objekte realisiert werden.
- Objekt-Relationales Mapping soll beide Adressen auf Spalten in der User–Tabelle abbilden.

Value Types

Beispiel (Benutzer und Adressen)

- Benutzer hat Liefer– und Rechnungsanschrift



- In UML durch Stereotypen darstellbar

Value Types

Beispiel (Java–Implementierung von Adressen)

```
public class Address {
    private String street;
    private String zipcode;
    private String city;

    public Address() {}

    public String getStreet() { return street; }
    public void setStreet(String street) {
          this.street = street; }
    public String getZipcode() { return zipcode; }
    public void setZipcode(String zipcode) {
        this.zipcode = zipcode; }
    public String getCity() { return city; }
    public void setCity(String city) {
          this.city = city; }
}
```

Value Types

Beispiel (Mapping von User mit Address–Komponenten)

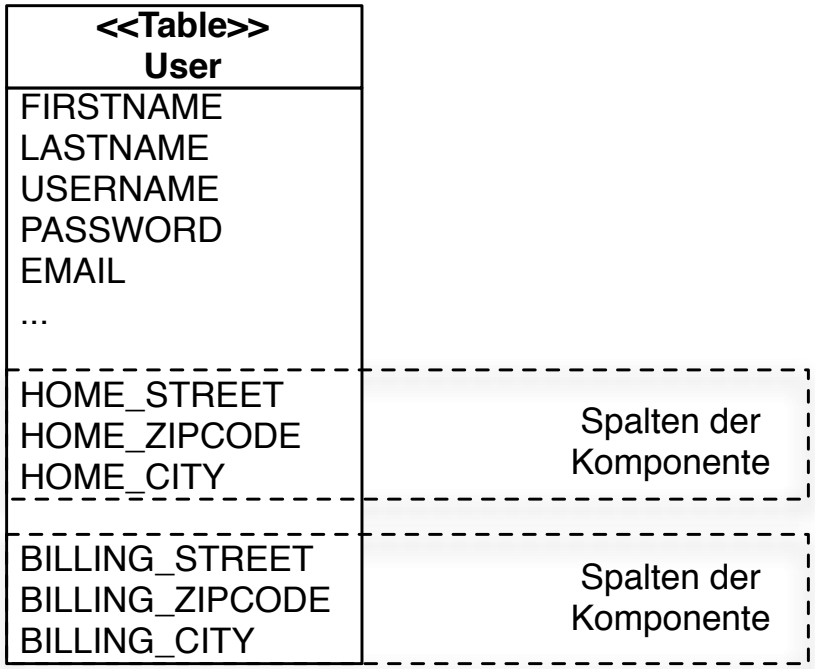
```
<class name="User" table="USER">
 <id name="id" column="USER_ID" type="long">
     <generator class="native"/>
 </id>
 <property name="loginName" column="LOGIN"
                            type="string"/>
 <component name="homeAddress" class="Address">
   <property name="street" type="string"
             column="HOME_STREET" not-null="true"/>
   <property name="city" type="string"
             column="HOME_CITY" not-null="true"/>
   <property name="zipcode" type="string"
             column="HOME_ZIPCODE" not-null="true"/>
 </component>
 ...
</class>
```

Value Types

Beispiel (Mapping von User mit Address–Komponenten)

```
<class name="User" table="USER">
  ...
 <component name="billingAddress" class="Address">
   <property name="street" type="string"
          column="BILLING_STREET" not-null="true"/>
   <property name="city" type="string"
          column="BILLING_CITY" not-null="true"/>
   <property name="zipcode" type="string"
          column="BILLING_ZIPCODE" not-null="true"/>
 </component>
 ...
</class>
```

Value Types



- Unidirektionale Assoziation:
- Keine Navigation von Adressen zu Benutzern
- Bidirektional:
  - Mapping–Eintrag `<parent name="user"/>`
  - Rückwärts–Navigation durch `Adress.getUser()`

Objekt–Beziehungen: Entity Types

Beziehungen zwischen eigenständigen Objekten (Entities)

- 1:1–Beziehungen

```
<one-to-one name="spam" class="Spam">
```

- n:1–Beziehungen

```
<many-to-one name="spam" class="Spam">
```

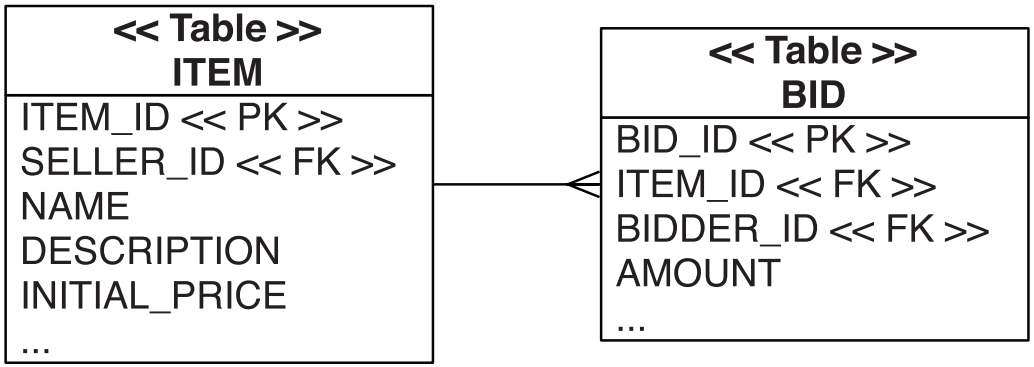
- n:m–Beziehungen

```
<set name="spam" key_column="SPAM_ID">
  <many-to-many class="Spam" column="SPAM_ID">
</set>
```

Abbildung von n:1–Beziehungen

Beispiel (many-to-one)

```
<class
    name="Bid"
    table="BID">
    ...
    <many-to-one
        name="item"
        column="ITEM_ID"
        class="Item"
        not-null="true"/>
</class>
```



Bidirektionale Assoziationen

Wie kann man von Items zu Bids navigieren?

Beispiel (Entity–Klasse Item)

```
public class Item {
    ...
    private Set bids = new HashSet();
    public void setBids(Set bids) {
        this.bids = bids;
    }
    public Set getBids() {
        return bids;
    }
    public void addBid(Bid bid) {
        bid.setItem(this);
        bids.add(bid);
    }
    ...
}
```

Bidirektionale Assoziationen

Beispiel (one-to-many)

```
<class name="Item" table="ITEM">
    ...
    <set name="bids"
         inverse="true">
      <key column="ITEM_ID"/>
      <one-to-many class="Bid"/>
    </set>
</class>
```

- Kein Mapping von Value–Typen sondern von Entity–Typen
- `inverse` gibt an, dass die Assoziation durch die selbe Spalte dargestellt wird wie die Rückrichtung.
- Das relationale Modell bleibt gleich.

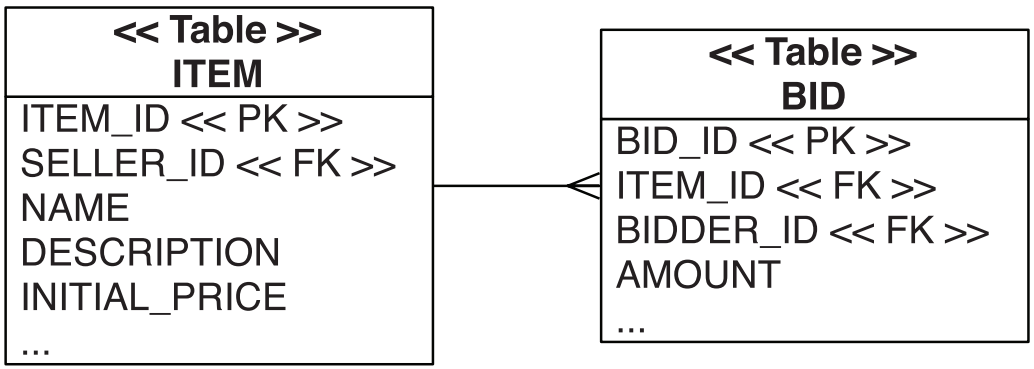


Abbildung von n:1–Beziehungen

Beispiel (Erzeugen von Item– und Bid–Objekten)

```
Item newItem = new Item();
Bid newBid = new Bid();

newItem.addBid(newBid); // Setzt beide Seiten
                        // der Beziehung

session.save(newItem);
session.save(newBid);
```

Kopplung von Entities

- Entities haben standardmäßig unabhängige Lebenszyklen `save` und `delete`–Aufrufe müssen für alle Objekte exlizit gemacht werden.
- Transitive Persistenz (`cascade`–Property) sorgt für gemeinsame Verwaltung von Objekt–Graphen.

Transitive Persistenz

Beispiel (Transitive Persistenz)

```
<class
    name="Item"
    table="ITEM">
    ...
    <set name="bids"
         inverse="true"
         cascade="save-update">
        <key column="ITEM_ID"/>
        <one-to-many class="Bid"/>
    </set>
</class>
```

Transitive Persistenz

Beispiel (Transitive Persistenz)

```
Item newItem = new Item();
Bid newBid = new Bid();

newItem.addBid(newBid); // Setzt beide Seiten
                        // der Beziehung

session.save(newItem);
```

- explizites `save` von `Bid` nicht mehr nötig

Kaskadiertes Löschen

Beispiel (Löschen von Item– und Bid–Objekten)

```
Item anItem = // lade Item-Objekt
// loesche alle referenzierten Bid-Objekte
for (Iterator<Bid> it = anItem.getBids().iterator();
      it.hasNext();) {
    Bid bid = it.next();
    it.remove(); // entferne Referenz aus Collection
    session.delete(bid); // loesche aus Datenbank
}
session.delete(anItem); // Zuletzt: loesche Item
```

Kopplung von Entities

- Entities haben standardmäßig unabhängige Lebenszyklen `save` und `delete`–Aufrufe müssen für alle Objekte exlizit gemacht werden.
- Transitive Persistenz (`cascade`–Property) sorgt für gemeinsame Verwaltung von Objekt–Graphen.

Kaskadiertes Löschen

Beispiel (Kaskadiertes Löschen)

```
<class
    name="Item"
    table="ITEM">
    ...
    <set name="bids"
         inverse="true"
         cascade="save-update, delete">
        <key column="ITEM_ID"/>
        <one-to-many class="Bid"/>
    </set>
</class>
```

Kaskadiertes Löschen

Beispiel (Kaskadiertes Löschen)

```
Item anItem = // lade Item-Objekt
session.delete(anItem);
entityManager.remove(anItem);
```

- explizites `delete` von `Bid`–Objekte nicht mehr nötig

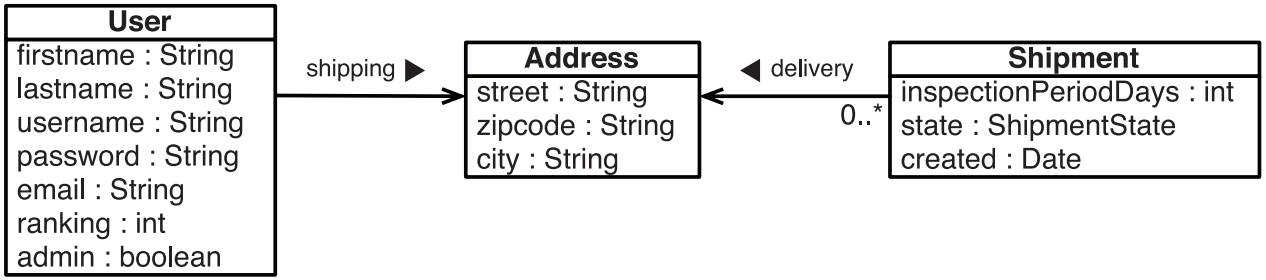
Objekt–Beziehungen: Eltern/Kind–Beziehungen

Teil/Ganzes–Beziehungen

Beispiel (gemeinsame Lebensdauer)

```
<class name="Item">...
 <set name="bids" inverse="true"
    cascade="save-update, delete, delete-orphan">
   <key column="ITEM_ID"/>
   <one-to-many class="Bid"/>
 </set>
</class>
```

1:1–Assoziationen



- Eine Address–Entität hat 1:1–Beziehungen zu User– und Shipment–Entitäten

1:1–Assoziationen mit Primärschlüssel

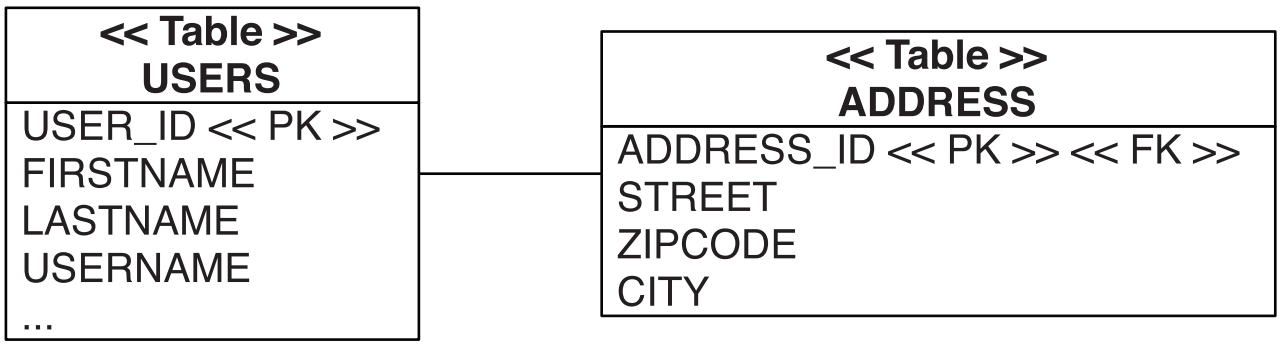
Beispiel (Gemeinsamer Primärschlüssel)

```
public class User {
    ...
    private Address shippingAddress;
    // Getters and setters
}

public class Address {
    ...
     private User user;
    // Getters and setters
}

<one-to-one name="shippingAddress"
            class="Address"
            cascade="save-update"/>
```

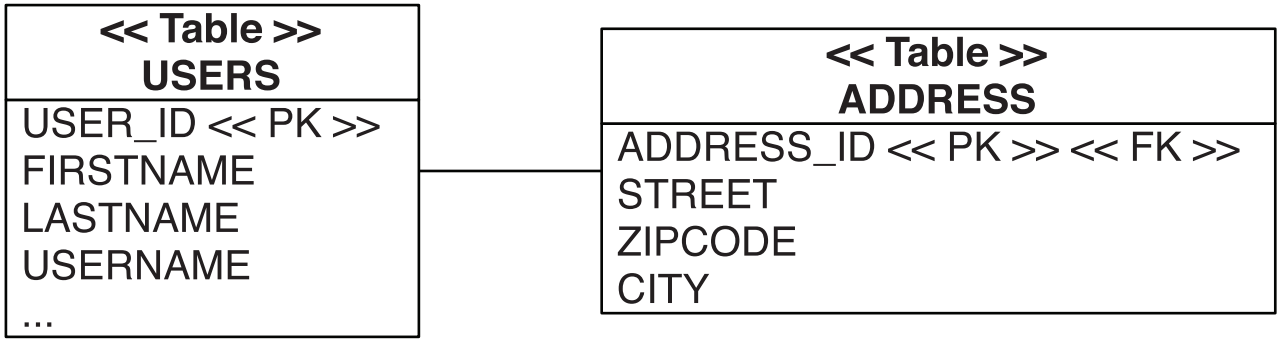
1:1–Assoziationen mit Primärschlüssel

Beispiel (Gemeinsamer Primärschlüssel)

```
<class name="Address" table="ADDRESS">
    <id name="id" column="ADDRESS_ID">
        <generator class="foreign">
            <param name="property">user</param>
        </generator>
    </id>
    ...
    <one-to-one name="user"
                class="User"
                constrained="true"/>
</class>
```



1:1–Assoziationen mit Primärschlüssel



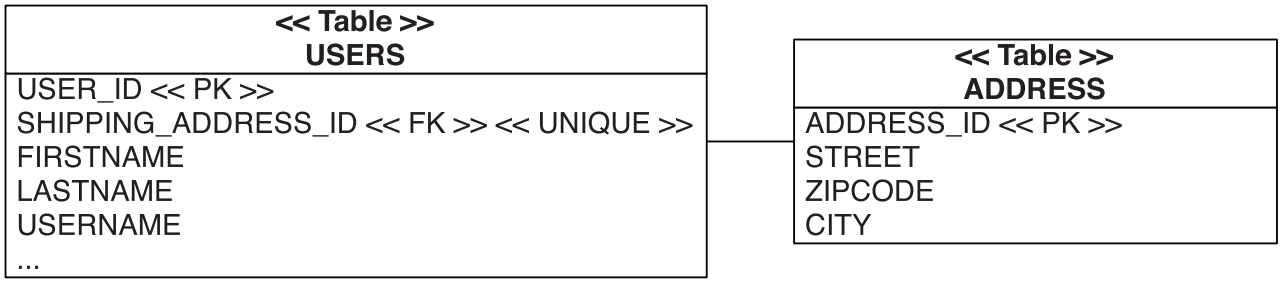
Beispiel (Gemeinsamer Primärschlüssel)

```
User newUser = new User();
Address shippingAddress = new Address();
newUser.setShippingAddress(shippingAddress);
shippingAddress.setUser(newUser);  // Bidirektional
session.save(newUser);
```

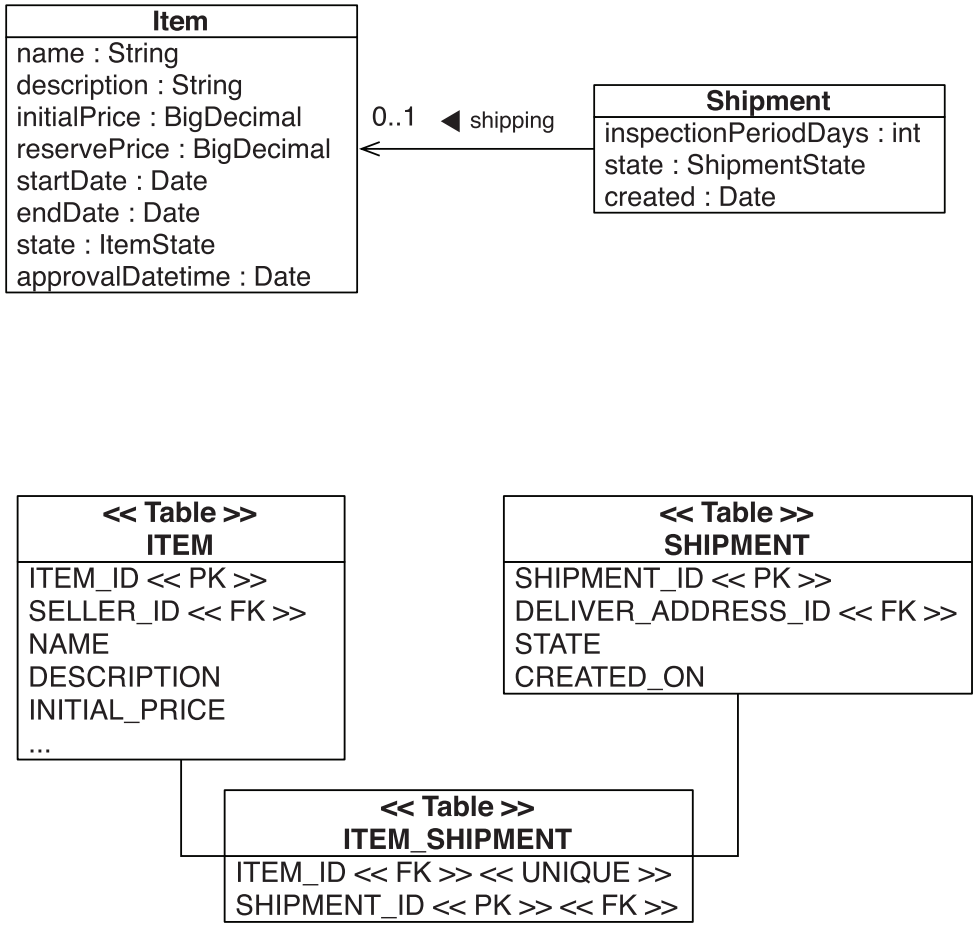
1:1–Assoziationen mit Fremdschlüssel

Beispiel (1:1–Assoziationen mit Fremdschlüssel)

```
<class name="User" table="USERS">
    <many-to-one name="shippingAddress"
                 class="Address"
                 column="SHIPPING_ADDRESS_ID"
                 cascade="save-update"
                 unique="true"/>
</class>
```



1:1–Assoziationen mit Join–Tabelle



1:1–Assoziationen mit Join–Tabelle

Beispiel (Join–Tabelle)

```
<class name="Shipment" table="SHIPMENT">
    <id name="id" column="SHIPMENT_ID">...</id>
    ...
    <join table="ITEM_SHIPMENT" optional="true">
        <key column="SHIPMENT_ID"/>
        <many-to-one name="auction"
                     column="ITEM_ID"
                     not-null="true"
                     unique="true"/>
    </join>
</class>
```

POJOs

- = Plain Old Java Objects
- bestehen aus Geschäftsmethoden (*business methods*) und Eigenschaften (*properties*).
- haben Accessoren (getter und setter) für die Eigenschaften.
  Werden von Hibernate automatisch erkannt, wenn Namenskonventionen eingehalten werden:
  - Property: *varname*
  - Getter: `get`*Varname*
  - Setter: `set`*Varname*
- Einem Java–Objekt wird der Zustand eines aus der Datenbank gelesenen Objekts gegeben:
  - Zugriff auf die Eigenschaften über Accessoren oder
  - direkter Zugriff auf die Eigenschaften
  - Hibernate umgeht Sichtbarkeitsangaben (`private`, `protected`)

Schema–Export: hbm2ddl

Aufruf:

```
java -cp classpath
     net.sf.hibernate.tool.hbm2ddl.SchemaExport
     <options> <mapping_files>

options
--quiet keine Ausgaben - yes/no
--drop Tabellen droppen - yes/no
--text kein Export zur DB - yes/no
--properties Hibernate propertiesfile
--delimiter Trennzeichen für das zu erstellende Skript
```

`hbm2ddl` steht auch über die Ant–Task `hibernatetool` zur Verfügung. Dann können die Parameter direkt als Tag-Attribute angegeben werden (`<task quiet="no" drop="no"....`).

Java–Klassen erzeugen: hbm2java

Aufruf:

```
java -cp classpath
  net.sf.hibernate.tool.hbm2java.CodeGenerator
  <options> <mapping_files>

options
--output Output-Verzeichnis
```

`hbm2java` steht auch über die Ant–Task `hibernatetool` zur Verfügung. Dann können die Parameter direkt als Tag-Attribute angegeben werden (`<task output = "..." ....`).

Ausblick

Entwicklungsrichtungen

- Top Down
- Bottom Up
- Middle out
- Meet-in-the-middle

APIs

Mapping

- Entity/Value Types
- Value Types
- Entity Types
- n:1–Assoziationen
- Eltern/Kind–Beziehungen
- 1:1–Assoziationen
- POJOs

Hibernate

- Tools

- Fragen?
- nächste Woche: mehr über Hibernate