Programmieren II

Dr. Klaus Höppner

Hochschule Darmstadt – Sommersemester 2010

Objekt-relationales Mapping

JPA Hibernate

Objekt-relationales Mapping

Mit JDBC kann Java sich mit relationalen Datenbanken verbinden.

Hierbei entsprechen

- jeder Datensatz einer Zeile in einer Datenbanktabelle und
- die Attribute eines Datensatzes den Tabellenspalten.

Daher ist es prinzipiell möglich, eine Abbildung zwischen den Datensätzen in einer Tabelle und klassischen Java-Objekten (POJO – *plain old Java objects*) herzustellen.

Eine solche Abbildung nennt man *objekt-relationales Mapping (ORM)*.

Beispieltabelle: Customer

Die folgende Tabelle beschreibt einen Kunden mit Adresse, wobei die ID des Kunden automatisch generiert wird (entsprechend des SQL-Dialektes für HSQLDB):

```
CREATE TABLE CUSTOMER(
    ID INTEGER GENERATED BY DEFAULT AS IDENTITY(START WITH 0)
                                                NOT NULL PRIMARY KEY,
    FIRSTNAME VARCHAR(20),
    LASTNAME VARCHAR(20),
    STREET VARCHAR(20),
    CITY VARCHAR(20)
)
```

Korrespondierende Klasse Customer

```
public class Customer {
    private Integer id;
    private String firstname;
    private String lastname;
    private String street;
    private String city;

    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }
```

Korrespondierende Klasse Customer (Forts.)

```
    public String getFirstname() {
        return firstname;
    }
    public void setFirstname(String firstname) {
        this.firstname = firstname;
    }
    // Weitere Getter und Setter analog
    @Override
    public String toString() {
        return String.format(
            "%s %s, %s, %s",
            firstname, lastname, street, city);
    }
}
```

Klasse DBConnection

Das Laden des JDBC-Treibers und der Aufbau der Verbindung wird in eine eigene Klasse ausgelagert:

```
public class DBManager {
    private Connection conn;

    public DBManager() {
        try {
            Class.forName("org.hsqldb.jdbcDriver");
            conn = DriverManager.getConnection(
                "jdbc:hsqldb:file:data/fibu;shutdown=true",
                "SA","");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

Klasse DBConnection (Forts.)

```
    public void close() {
        if (conn!=null) {
            try {
                conn.close();
            } catch (SQLException e) {
                e.printStackTrace();
            } finally {
                conn = null;
            }
        }
    }
}
```

Verknüpfung DB-Tabelle $\leftrightarrow$ Java-Objekt

Nun sollen Java-Objekte *persistent* gemacht werden, indem Objekte vom Typ `Customer` mit der gleichnamigen DB-Tabelle synchronisiert werden können.

Hierzu werden der Klasse `DBManager` zwei neue Methoden hinzugefügt:

`Customer load(Integer id)` Hier werden für eine ID die Werte der Attribute per **`SELECT`** aus der Tabelle geholt und eine neue Instanz von `Customer` erzeugt.

`void persist(Customer c)` Hier werden die Werte der Attribute von `c` in die Tabelle geschrieben, wobei bei leerer ID (**`null`**) ein neuer Datensatz per **`INSERT`** eingefügt, andernfalls die Feldwerte des existierenden Datensatzes per **`UPDATE`** aktualisiert werden.

Methode load von DBManager

```
public Customer load(Integer id) {
    Customer c = null;
    try {
        PreparedStatement st = conn.prepareStatement(
            "select * from CUSTOMER where id=?");
        st.setInt(1, id);
        ResultSet res = st.executeQuery();
        if (res.next()) {
            c = new Customer();
            c.setId(res.getInt("ID"));
            c.setFirstname(res.getString("FIRSTNAME"));
            c.setLastname(res.getString("LASTNAME"));
            c.setStreet(res.getString("STREET"));
            c.setCity(res.getString("CITY"));
        }
        res.close();
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return c;
}
```

Methode persist von DBManager

```
public void persist(Customer c) {
    PreparedStatement st;
    try {
        if (c.getId()==null) {
            st = conn.prepareStatement("insert into CUSTOMER"+
                "(FIRSTNAME,LASTNAME,STREET,CITY) values (?,?,?,?)");
        } else {
            st = conn.prepareStatement("update CUSTOMER set"+
                "FIRSTNAME=?, LASTNAME=?, STREET=?, CITY=? where id=?");
            st.setInt(5, c.getId());
        }
        st.setString(1, c.getFirstname());
        st.setString(2, c.getLastname());
        st.setString(3, c.getStreet());
        st.setString(4, c.getCity());
        st.executeUpdate();
```

Methode persist von DBManager (Forts.)

```
            if (c.getId()==null) {
                st = conn.prepareStatement("call identity()");
                ResultSet res = st.executeQuery();
                res.next();
                c.setId(new Integer(res.getInt(1)));
                res.close();
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
```

Exkurs: `PreparedStatement`

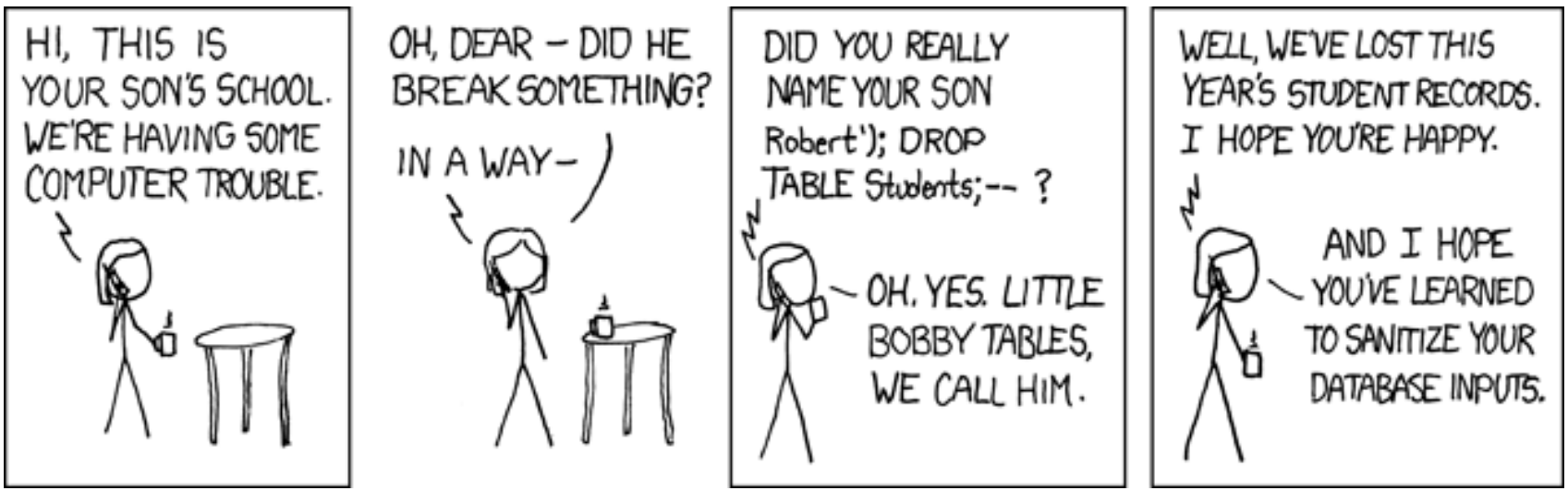
In der Realisierung von `fetch()` und `update()` wurde statt `Statement` jeweils `PreparedStatement` benutzt.

Bei `PreparedStatement` wird beim Anlegen das Statement bereits mit einem SQL-Befehl (i. A. mit Platzhaltern) vorbereitet. Dieser wird von der Datenbank vorkompiliert. Danach können die Platzhalter mit den realen Werte belegt werden.

Vorteile:

- Bei mehrfacher Verwendung (z. B. in Schleifen) ergibt sich ein Laufzeitgewinn, da der Befehl schon vorkompiliert ist.
- Da die Gültigkeit der Parameter überprüft wird und SQL-Sonderzeichen (z. B. ’) geschützt werden, werden so genannte *SQL-Injections* verhindert.

Kleiner Cartoon



Quelle: http://xkcd.com/327/

Zusammenfassung

In diesem Abschnitt wurde anhand einer *einfachen* Tabelle demonstriert, wie ein objekt-relationales Mapping selber realisiert werden kann.

Der Aufwand war aber recht hoch, da individuell für die Klasse die entsprechenden Methoden zur Synchronisation inkl. der entsprechenden SQL-Befehle implementiert wurden. Wegen verschiedener SQL-Dialekte der RDBMS ist die Implementierung auch DB-spezifisch.

Der Aufwand wird noch höher, wenn Tabellen über Schlüssel miteinander verknüpft sind.

Daher werden in der Praxis für ORM spezielle Frameworks benutzt.

Hibernate

Es existieren verschiedene Frameworks, die ein objekt-relationales Mapping realisieren, mittlerweile standadisiert über die *Java Persistence API – JPA*.

Eines der bekanntesten und häufig benutzten ist *Hibernate* (http://www.hibernate.org).

Was ist der Vorteil eines solchen Frameworks und ORM?

Es verbindet die Vorteile der objektorientierten Programmierung mit den Vorteilen einer Datenbank, insbesondere der Persistenz (d. h. dass Datensätze in einem RDBMS nicht flüchtig sind). So kann man persistente Objekte realisieren, die sich weitgehend wie normale Java-Objekte verwenden lassen.

Geschichte

Hibernate wurde von der Firma JBoss entwickelt, die sich mit Open-Source-Software im Bereich Java-Middleware, -Applikationsservern und Message-Brokern beschäftigt.

Mittlerweile wurde JBoss (nach einem Bieterkampf mit Oracle) von RedHat übernommen.

Architektur

Hibernate stellt in der einfachsten Benutzungsform dem Anwender eine Datenbank-Session zur Verfügung, die nach Konfiguration eines Datenbanktyps und der Datenbankparameter geöffnet wird.

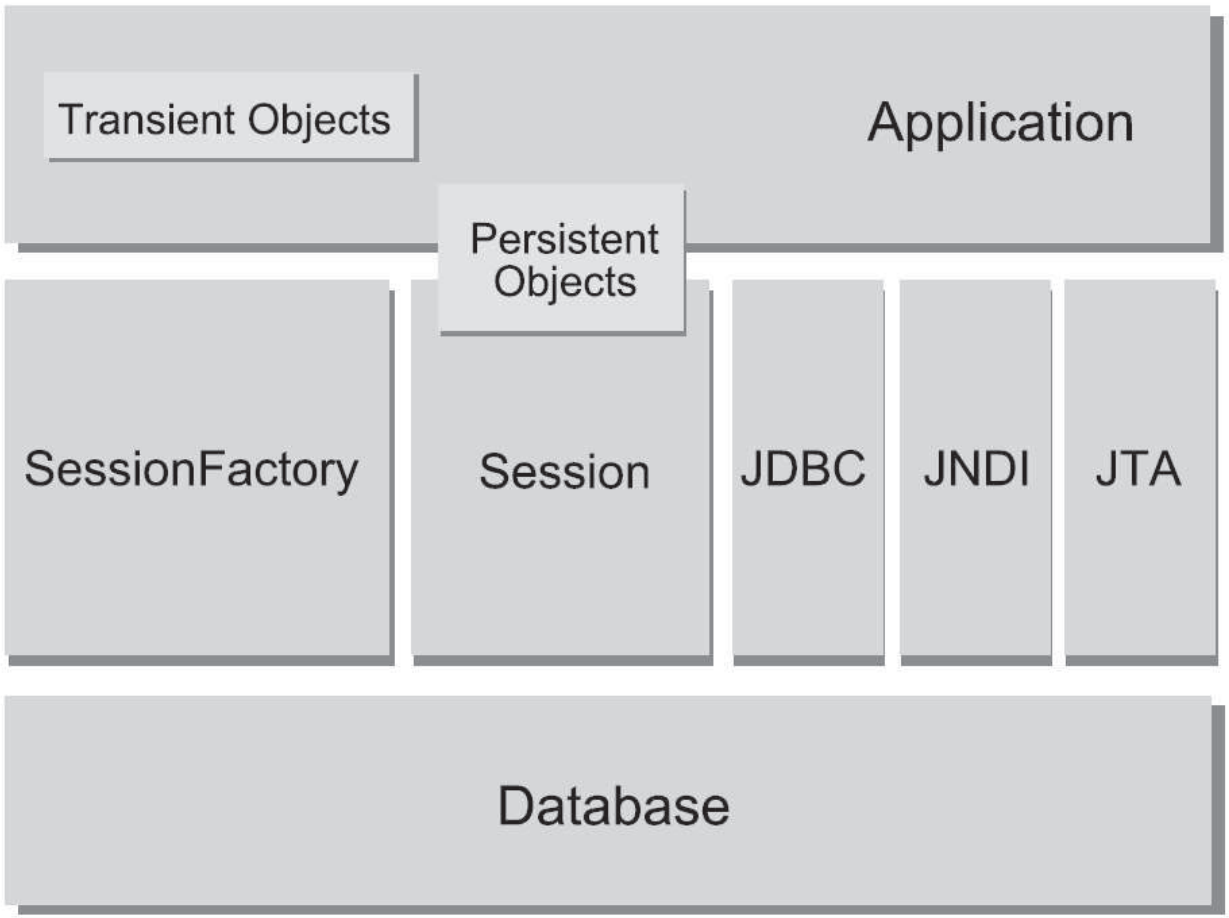
Objekte können sich in drei Zuständen befinden:

transient Das Objekt ist noch nicht mit der Datenbank verknüpft, also flüchtig.

persistent Das Objekt ist im aktuellen Kontext mit der Datenbank verknüpft.

detached Das Objekt war mit der Datenbank verknüpft, der Kontext existiert aber nicht mehr. Daher ist das Objekt aktuell losgelöst.

Schaubild


Transient Objects
Application
Persistent Objects
SessionFactory
Session
JDBC
JNDI
JTA
Database

Konfiguration einer Hibernate-Anwendung

In der Java-SE verwendet man i. A. mit Hibernate einen Entity-Manager, der aus einer so genannten *Persistence Unit* erzeugt wird.

Hierbei erfolgt die Konfiguration der Persistence Unit in der Datei `META-INF/persistence.xml`:

```
<?xml version="1.0" encoding="utf-8" ?>
<persistence  xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistenc
  version="2.0">
  <persistence-unit name="manager1" transaction-type="RESOURCE_LOCAL">
    <properties>
      <property name="javax.persistence.jdbc.driver" value="org.hsqldb.jdbcDriver"/>
      <property name="javax.persistence.jdbc.user" value="sa"/>
      <property name="javax.persistence.jdbc.password" value=""/>
      <property name="javax.persistence.jdbc.url" value="jdbc:hsqldb:file:data/fibu;shutdown=true"
      <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect"/>
      <property name="hibernate.max_fetch_depth" value="3"/>
    </properties>
  </persistence-unit>
</persistence>
```

Anlegen des Entity-Managers

```
import javax.persistence.*;

public class Anwendung {
    public static void main(String[] args) {
        EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("manager1");

        EntityManager em = emf.createEntityManager();

        // Arbeiten mit persistenten Objekten

        em.close();
        emf.close();
    }
}
```

Grundlagen von ORM

Durch ORM werden Klassen mit Tabellen und Instanzen der Klassen mit Datensätzen innerhalb der Tabelle verknüpft.

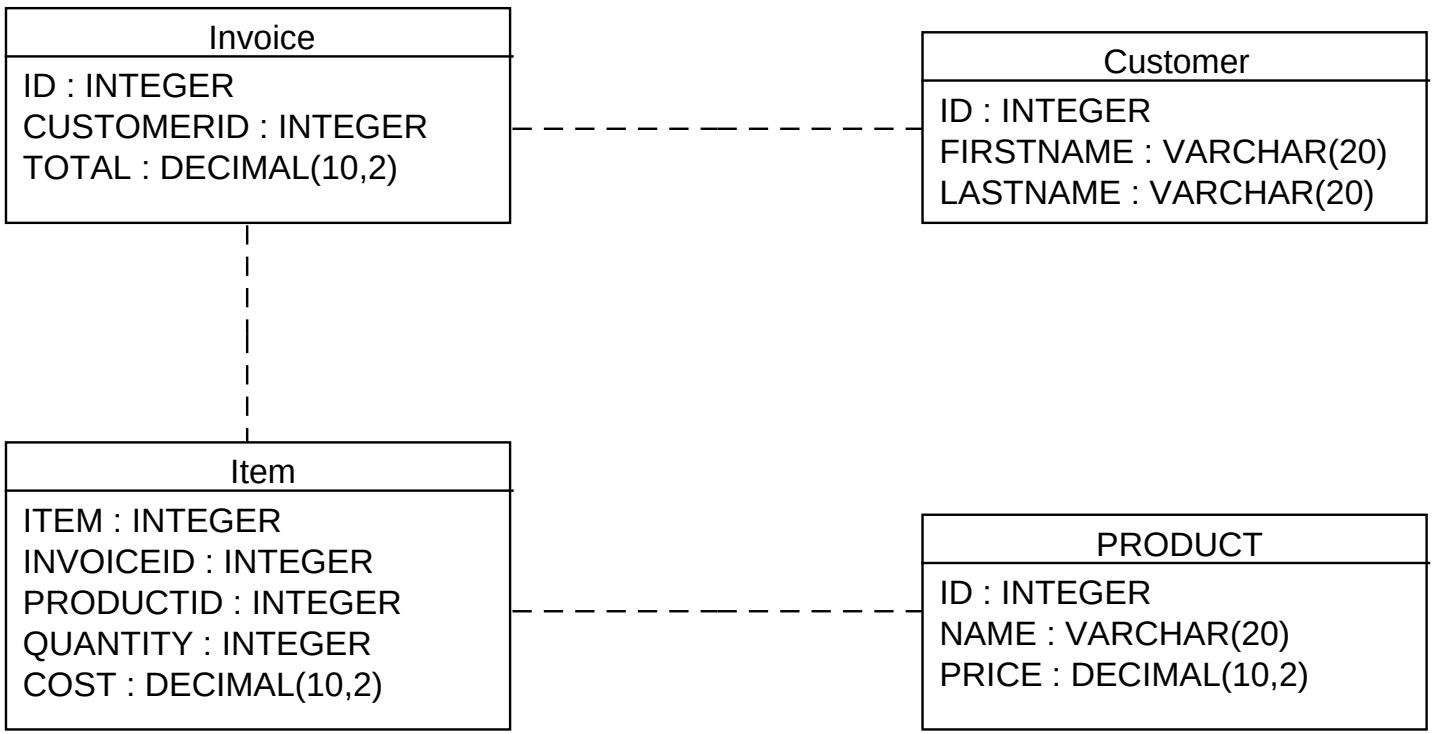
Damit Instanzen eindeutig Datensätzen zugeordnet werden können, muss die Datenbanktabelle einen eindeutigen Primärschlüssel besitzen. Dieser wird dann als `WHERE`-Ausdruck in den abgesetzten SQL-Anweisungen verwendet.
Weiterhin dient die Existenz einer Primär-ID im Objekt der Unterscheidung, ob ein nicht persistentes Objekt transient oder detached ist.

Faustregeln

- Im Allgemeinen wird eine Tabelle der Datenbank einer Java-Klasse zugeordnet.
- Diese Klasse muss einen Standardkonstruktor besitzen.
- Die Attribute der Tabelle werden auf entsprechende Attribute der Klasse abgebildet.
- In der Klasse werden dann Getter und Setter für diese Attribute definiert.
- Die eigentliche Zuordnung zwischen Klasse und Tabelle erfolgt dann über eine XML-Datei, die insbesondere den Primärschlüssel angibt, der zur Identifikation von Elementen verwendet wird, sowie die Namen von Tabellenspalten den Attributen der Klasse zuordnet.

Beispiel: Datenbank Fibu

Hier nun ein Datenbankschema für eine Buchhaltung:



Erster Ansatz: Klasse Invoice

```
import javax.persistence.*;

@Entity
public class Invoice {
    private Integer id;
    private Integer customerid;
    private Double total;

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }
```

Erster Ansatz: Klasse Invoice (Forts.)

```
    public Integer getCustomerid() {
        return customerid;
    }
    public void setCustomerid(Integer customerid) {
        this.customerid = customerid;
    }
    public Double getTotal() {
        return total;
    }
    public void setTotal(Double total) {
        this.total = total;
    }
}
```

Mapping

Während in Hibernate das ORM ursprünglich über XML-Dateien erfolgte, geht dies nun über Annotationen:

- Die Klasse `Invoice` wird über Annotation als Entität gekennzeichnet, wobei per Default angenommen wird, dass die Datenbanktabelle den selben Namen hat.
- Dann wird das Attribut `id` der Java-Klasse als Primärschlüssel markiert.
- Alle weiteren Attribute der Klasse werden automatisch auf die Datenbankspalte gleichen Namens abgebildet.

Zweiter Ansatz für Invoice

In Wirklichkeit möchte man in der Klasse `Invoice` statt der reinen Kundennr. lieber direkt eine Referenz auf den Kunden haben:

```
@Entity
public class Invoice {
    ...
    private Customer customer;
    ...

    @ManyToOne
    @JoinColumn(name="CUSTOMERID")
    public Customer getCustomer() {
        return customer;
    }
    public void setCustomer(Customer customer) {
        this.customer = customer;
    }
}
```

Mapping

Dies ist ein Fall einer Zuordnung *many-to-one*: Mehrere Rechnungen können für denselben Kunden ausgestellt sein.

Diese Abbildung erfolgt über die Annotation `@ManyToOne`, wobei der Name der Tabellenspalte mit dem Fremdschlüssel angegeben werden muss, wenn diese Spalte nicht einfach `FKTAB_ID` heißt, wobei `FKTAB` der Name der verknüpften Tabelle ist.

Die Klasse `Customer` muss natürlich auch entsprechend mit Annotationen für JPA versehen sein.

Mapping der Rechnungspositionen

Einer Rechnung sind in der Regel viele Rechnungsposten zugeordnet. Die Zuordnung erfolgt durch die Spalte `invoiceId` in der Tabelle `Item`.

Hierfür wird in der Klasse `Invoice` nun eine Collection von `Item` neu hinzugefügt, die eine Abbildung vom Typ *one-to-many* ist. Hierbei wird für die Zuordnung die Spalte `invoiceID` aus `Item` als *foreign key* verwendet, anhand dessen die passenden Rechnungsposten ausgewählt werden.
Zusätzlich wird direkt die Tabelle `Product` anhand der Produktnr. des Rechnungsposten in die Klasse `Item` abgebildet.

Klasse Invoice

```
@Entity
public class Invoice {
    private Collection<Item> items;

    @OneToMany(mappedBy="invoice")
    public Collection<Item> getItems() {
        return items;
    }
    public void setItems(Collection<Item> items) {
        this.items = items;
    }
}
```

Klasse Item

```
@Entity
public class Item {
    private Integer id;
    private Invoice invoice;
    private Product product;
    private Integer quantity;

    @Id
    @Column(name="ITEM")
    @GeneratedValue(strategy=GenerationType.AUTO)
    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }
```

Klasse Item (Forts.)

```
@ManyToOne
@JoinColumn(name="INVOICEID")
public Invoice getInvoice() {
    return invoice;
}
public void setInvoice(Invoice invoice) {
    this.invoice = invoice;
}
@ManyToOne
@JoinColumn(name="PRODUCTID")
public Product getProduct() {
    return product;
}
public void setProduct(Product product) {
    this.product = product;
}
```

Klasse Item (Forts.)

```
    public Integer getQuantity() {
        return quantity;
    }
    public void setQuantity(Integer quantity) {
        this.quantity = quantity;
    }
}
```

Klasse Product

```
@Entity
public class Product {
    private Integer id;
    private String name;
    private Double price;

    @Id
    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }
```

Klasse Product (Forts.)

```
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Double getPrice() {
        return price;
    }
    public void setPrice(Double price) {
        this.price = price;
    }
}
```

Anwendung

```
public class Anwendung {
    public static void main(String[] args) {
        EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("manager1");

        EntityManager em = emf.createEntityManager();
        EntityTransaction tx = em.getTransaction();

        tx.begin();
        Customer c = em.find(Customer.class, new Integer(5));
        System.out.println(c);
        c.setCity("Frankfurt");
        tx.commit();
```

Anwendung (Forts.)

```
tx.begin();
c = new Customer();
c.setFirstname("Karl");
c.setLastname("Meier");
c.setStreet("Ahornweg");
c.setCity("Köln");
em.persist(c);
tx.commit();
System.out.println(c.getId());
```

Anwendung (Forts.)

```
        Invoice inv = em.find(Invoice.class, new Integer(2));
        System.out.println(inv.getCustomer());
        for (Item item : inv.getItems()) {
            System.out.format("%d mal %s\n",
                    item.getQuantity(),
                    item.getProduct().getName());
        }

        em.close();
        emf.close();
    }

}
```