Objekt-relationales Mapping mit Hibernate

Überwintern im Eis

Viele Java-Anwendungen müssen ihre Objekte dauerhaft speichern. Relationale Datenbanken eignen sich dafür nur mit zusätzlichem Aufwand. Das Hibernate-Framework übersetzt Java-Objekte in Tabellenform und übernimmt ihre komplette Verwaltung. Thorsten Ehlers, Martin Schmidt

Jeder Java-Entwickler steht irgendwann vor dem Problem, Objekte und ihre Beziehungen persistent zu machen. Obwohl es objektorientierte Datenbanken schon seit langem gibt, dominieren in der Praxis ihre relationalen Verwandten. Die passen aber nicht recht zum Objektmodell, was im Fachjargon Impedance Mismatch heißt. Das Java-Paket Hibernate [1] (auf gut Deutsch: Winterschlaf halten) löst das Problem, indem es eine spezielle Softwarekomponente implementiert, die so genannte Persistenzschicht. Damit bildet es Objekte und ihre Attribute auf Tabellen mit Zeilen und Spalten ab. Diese Abbildung heißt objekt-relationales Mapping (ORM).

Objekte in Tabellen

Ein ORM-System selbst zu entwickeln ist nicht ganz einfach, man läuft Gefahr, einen Großteil der Entwicklungszeit in die Persistenzschicht zu investieren statt an der eigentlichen Aufgabenstellung zu arbeiten. Typischerweise macht das dann 30 Prozent oder mehr des gesamten Projektcodes aus.

Hibernate ist ein fertiges objekt-relationales Mapping-Framework für Java und erfreut sich als freie Software mittlerweile großer Beliebtheit. Das Projekt wird auf Sourceforge geführt und steht unter der Lesser GNU Public License. Die Dokumentation ist ausführlich und die Forenbenutzer leisten schnell qualifizierte Hilfe. Kommerziellen Support und Training bietet die Firma JBoss. Das folgenden Beispiel zeigt anhand einer einfachen Kundendatenbank, wie man Hibernate verwendet. Die Beziehungen zwischen den Tabellen sind in Abbildung 1 dargestellt, die zugehörigen Java-Klassen zeigt Abbildung 2.

Persistente Klassen bilden die Schnittstelle zwischen der Anwendung und der Datenbank. Diese Klassen, die der Java-Bean-Konvention [2] entsprechen müssen, werden von Hibernate auf die Tabellen der Datenbank abgebildet. Eine solche Bean besitzt

- einen Unique Identifier (üblicherweise der Primary Key des Datensatzes aus der Datenbank),
- Properties (die Werte der Spalten aus der zugehörigen Tabelle) und
- Referenzen auf andere Beans (die Beziehungen innerhalb des relationalen Modells).

Exemplare einer Bean können von Hibernate persistent gemacht werden, müssen es aber nicht. Die Beans lassen sich auch unabhängig von Hibernate

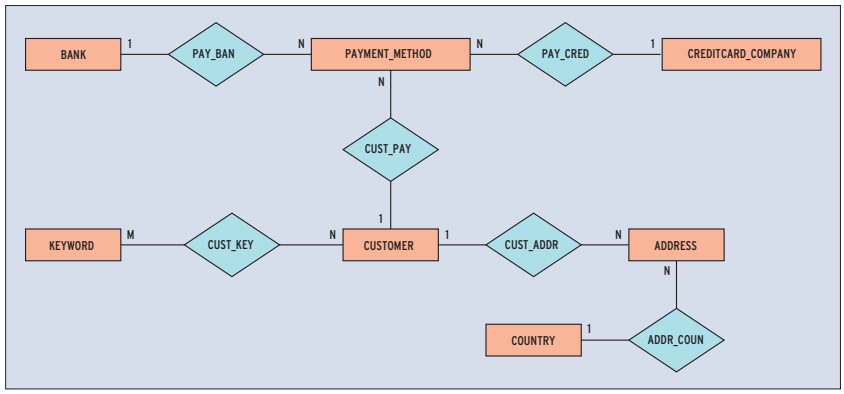

Abbildung 1: Entity-Relationship-Diagramm der Beispieldatenbank: Zu einem Land »COUNTRY« gehören maximal »N« Adressen »ADDRESS« , jede Adresse kann aber nur »1« Land enthalten.

verwenden und entwickeln. Das unterscheidet Hibernate von Technologien wie Container Managed Persistence im J2EE-EJB-Kontext (siehe [3]), bei denen die Beans einen dafür vorgesehenen Container brauchen und spezielle Interfaces implementieren (oder noch schlimmer: spezielle Klassen erweitern) müssen. Im Gegensatz dazu verfolgt Hibernate den Ansatz eines passiven Persistenz-Service, der die Objekte nicht selbst verwaltet.

Bean-Schnittstelle

Listing 1 zeigt eine solche Bean für Adressdaten. Seine Properties müssen nicht als »public« deklariert sein, wie es oft bei anderen ORM-Tools der Fall ist. Hibernate setzt sie über entsprechende Setter- und Getter-Methoden.

Alle Java-Beans, die Hibernate persistent machen soll, registriert der Programmierer bei einer »SessionFactory«. Sie sollte pro Datenbank als Singleton existieren, siehe Listing 2, Zeile 10. Mit ihrer Hilfe erzeugt er Session-Objekte, über die die eigentlichen Datenbankoperationen laufen, optional als Transaktionen. Verbindungen (Connections) erhält die »SessionFactory« von einem »ConnectionProvider«, einem Adapter, der intern meist einen Connection-Pool verwaltet.

Hibernate bringt eine Vielzahl von Adaptern mit. Für den Produktivbetrieb hat sich wegen seiner Stabilität und der sehr guten Konfigurationsmöglichkeiten der DBCP aus dem Apache-Projekt als erste Wahl erwiesen. Dieser und alle übrigen mitgelieferten Adapter werden über eine einfache Property-Datei konfiguriert. Sofern nicht anders vorgegeben, liest Hibernate die Datei »hibernate.properties« in der Wurzel des »CLASSPATH«, wenn es das Configuration-Objekt erzeugt. Wer diesen Mechanismus nicht verwenden möchte, übergibt die gewünschten Optionen manuell in einem Properties-Objekt.

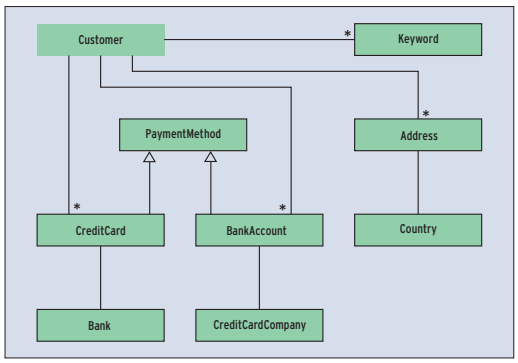

Abbildung 2: Dem Datenbankschema entsprechende Java-Klassen als UML-Diagramm. Die N:M-Relation zwischen »CUSTOMER« und »KEYWORD« aus Abbildung 1 wird von Hibernate auf ein einfaches Set in der Klasse »Customer« abgebildet.

Freie Auswahl an Datenbanken

Die Property »hibernate.dialect« spezifiziert den Datenbankhersteller. Hibernate unterstützt die SQL-Dialekte aller wichtigen Datenbanken wie DB 2, MySQL, Oracle, PostgreSQL oder HypersonicSQL. So kann das Framework den passenden Mechanismus zur Primärschlüssel-Generierung verwenden (also Sequenzen bei Oracle und PostgreSQL und Identity-Typen bei HypersonicSQL und MySQL) und sehr performante SQL-Statements erzeugen. Bei einigen Datenbanken ist zum Beispiel ein Update schneller, wenn nur die geänderten Spalten aktualisiert werden, bei anderen dagegen, wenn sich die ganze Zeile ändert.

Die Property »hibernate.show_sql« – auf »true« gesetzt – bringt Hibernate dazu, sämtliche generierten SQL-Statements mitzuloggen. Das hilft in vielen Fällen bei der Fehlersuche. Listing 3 zeigt eine Beispielkonfiguration für die Datenbank HypersonicSQL.

Eine XML-Datei bestimmt die Persistenz der Beans. Sie beschreibt unter anderem auch, auf welche Properties Hibernate die Tabellenspalten abbildet. Beim Speichern einer Bean erzeugt es automatisch eine ID gemäß der jeweils konfigurierten Methode.

Metadaten geben Hibernate die nötigen Informationen, um Insert-, Update-, Delete- und Select-Anfragen an die Datenbank zu generieren. Das Metadatenformat von Hibernate ist gut lesbar und verfügt über brauchbare Default-Werte. Man kann bei der Erstellung des Konfigurationsobjekts das Mapping entweder für alle Klassen in einer einzigen Datei oder für jede Klasse einzeln übergeben. Hibernate sucht im »CLASS-

Listing 1: »Address.java«

```
01 package com.freiheit.beans;
02 import java.io.Serializable;
03
04 public class Address implements Serializable {
05
06    private Integer _id;
07    private String _firstname;
08    private Country _country;
09    private Customer _customer;
10
11    /* ... weitere Properties ... */
12
13    public Address() {
14    }
15
16    public Integer getId() {
17        return _id;
18    }
19    public void setId(Integer id) {
20        _id = id;
21    }
22
23    public String getFirstname() {
24        return _firstname;
25    }
26    public void setFirstname(String firstname) {
27        _firstname = firstname;
28    }
29
30    /* ... weitere Getter und Setter ... */
31 }
```

Listing 2: »Main.java« (1)

```
01 Configuration configuration = new Configuration()
02    .addClass( Address.class )
03    .addClass( Bank.class )
04    .addClass( CreditCardCompany.class )
05    .addClass( PaymentMethod.class )
06    .addClass( Country.class )
07    .addClass( Customer.class )
08    .addClass( Keyword.class );
09
10 SessionFactory factory = configuration.
   buildSessionFactory();
```

Listing 3: »hibernate.properties«

```
01 hibernate.dialect net.sf.hibernate.dialect.HSQLDialect
02
03 hibernate.connection.driver_class org.hsqldb.jdbcDriver
04 hibernate.connection.username sa
05 hibernate.connection.password
06 hibernate.connection.url jdbc:hsqldb:test
07
08 hibernate.connection.pool_size 1
09 hibernate.statement_cache.size 25
10
11 hibernate.show_sql true
```

PATH« nach einer Datei, die auf ».hbm .xml« statt auf ».class« endet, aber sonst wie das gemappte Objekt heißt.

Das XML-Element »Class« ordnet einer Java-Klasse eine Tabelle zu. In Listing 4 gehört zur Klasse »Address« die Tabelle »ADDRESS« (Zeile 6). Das zusätzliche Attribut »mutable« gibt an, ob Hibernate die Tabelle nur lesen (Wert »false«) oder auch schreiben darf.

Das Element »id« definiert die Tabellenspalte, die den Primärschlüssel enthält, und ihre Zuordnung zu einer Bean-Property. Das Attribut »unsaved-value« steht dafür, ob eine Bean bereits in der Datenbank vorhanden ist oder nicht. Im Beispiel existiert sie noch nicht, wenn »id« gleich »NULL« ist. Das Kindelement »generator« gibt an, wie Hibernate den Primärschlüssel beim Persistieren neuer Beans erzeugt. Die »property«-Elemente schließlich ordnen die Bean-Properties den entsprechenden Datenbankspalten und -typen zu.

Listing 4: »Address.hbm.xml«

```
01 <?xml version="1.0"?>
02 <!DOCTYPE hibernate-mapping PUBLIC
03         "-//Hibernate/Hibernate Mapping DTD//EN"
04         "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
05 <hibernate-mapping>
06     <class name="com.freiheit.beans.Address" table="ADDRESS" mutable="true">
07
08         <id name="id" type="integer" column="ID" unsaved-value="null">
09             <generator class="identity" />
10         </id>
11
12         <many-to-one name="country" class="com.freiheit.beans.Country"
13                      column="COUNTRY_ID" />
14         <many-to-one name="customer" class="com.freiheit.beans.Customer"
15                      column="CUSTOMER_ID" not-null="true" />
16
17         <property name="firstname" column="FIRSTNAME" type="string" />
18         <property name="creationDate" column="CREATION_DATE" type="calendar" />
19         <property name="changingDate" column="CHANGING_DATE" type="calendar" />
20
21         <!-- ... weitere Properties ... -->
22     </class>
23 </hibernate-mapping>
```

Listing 5: »Customer.java«

```
01 public class Customer implements Serializable
   {
02
03     private Integer _id;
04     private String _firstname;
05     private Set _addresses;
06     private Set _bankAccounts;
07     private Set _creditCards;
08     private Set _keywords;
09
10     /* ... weitere Properties ... */
11
12     public Customer() {
13     }
14
15     /* ... Getter und Setter ... */
16
17     public void addAddress( Address address )
       {
18         address.setCustomer( this );
19         if ( _addresses == null ) {
20             _addresses = new HashSet();
21         }
22         _addresses.add( address );
23     }
24 }
```

Listing 6: »Main.java« (2)

```
01 Customer customer = new Customer();
02 customer.setFirstname("Hans");
03 customer.setLastname("Wurst");
04 ...
05
06 Address address = new Address();
07 address.setPostalCode("12345");
08 address.setCity("Musterhausen");
09 address.setCountry(new Country(1));
10 ...
11
12 customer.addAddress(address);
13
14 Session session = factory.openSession();
15 session.saveOrUpdate( customer );
16 session.flush();
17 session.connection().commit();
```

Vielfache Beziehungen

Um Foreign-Key-Verknüpfungen auf Objektreferenzen abzubilden, steht das Element »many-to-one« zur Verfügung:

```
<many-to-one name="country"
   class="com.freiheit.beans.Country"
   column="COUNTRY_ID" />
```

Hibernate ermittelt anhand der angegebenen Klasse und der zugehörigen Konfiguration die Tabelle, die die Zeile mit dem Primary Key »COUNTRY_ID« enthält. Mit den ermittelten Daten erzeugt es ein neues Objekt und übergibt eine Referenz darauf an das ursprünglich erzeugte Objekt. Wenn eine Zeile einer Tabelle von mehreren Zeilen einer anderen Tabelle referenziert wird, bildet Hibernate dies mit Hilfe des Elements »one-to-many« auf eine Collection ab:

```
<set name="addresses" inverse="true"
       cascade="all-delete-orphan">
    <key column="CUSTOMER_ID" />
    <one-to-many class="com.freiheit.
       beans.Address" />
</set>
```

Wie auch bei der Many-to-One-Verknüpfung ermittelt Hibernate automatisch die betroffene Tabelle und lädt die Datensätze. Da dieser Vorgang unter Umständen sehr lange dauern kann, lässt er sich mit dem Parameter »lazy« auf den Zeitpunkt verschieben, zu dem die Applikation die Daten braucht.

Collections inklusive

Hibernate unterstützt alle Standard-Collection-Typen. Allerdings brauchen Listen noch eine zusätzliche Tabellenspalte für den Index. Dem Attribut »inverse« kommt besondere Bedeutung zu: Es besagt, dass bei der bidirektionalen Verknüpfung zwischen »Customer«- und »Address«-Objekten das »Customer«-Objekt als Erstes erzeugt werden muss, wenn es noch nicht persistent ist.

Die verknüpften Objekte werden im »Customer«-Objekt als Collections vom Typ »Set« abgebildet. Listing 5 zeigt dies ausschnittweise (Zeilen 5 bis 8 und 19

bis 22). Die Methode »addAddress()« stellt die bidirektionale Verknüpfung auf Java-Ebene her.

Listing 6 zeigt in einem Ausschnitt des Hauptprogramms, wie sich nun die Objektdaten mit den neuen Klassen setzen lassen: bei jedem Objekt einfach über die zum Attribut gehörige Set-Methode. Die Methode »saveOrUpdate()« erkennt automatisch, ob ein Insert oder ein Update nötig ist. Im Beispiel sind das Customer- und das referenzierte Adressobjekt noch nicht in der Datenbank vorhanden. Hibernate macht erst das »Customer«-Objekt persistent und speichert dann das referenzierte »Address«-Objekt, wobei es zusätzlich den Fremdschlüssel auf das »Customer«-Objekt einfügt.

Polymorphie mit Tabellen

Mit Hilfe des Elements »discriminator« lässt sich die Polymorphie des Objektmodells mit dem flachen relationalen Modell verbinden. Eine Tabelle nimmt

Listing 7: »PaymentMethod.hbm.xml«

```
01 <class name="com.freiheit.beans.PaymentMethod" table="PAYMENT_METHOD"
02        mutable="true">
03
04    <id name="id" column="ID" unsaved-value="any">
05        <generator class="identity" />
06    </id>
07
08    <discriminator column="TYPE" />
09
10    <many-to-one name="customer" class="com.freiheit.beans.Customer"
11                 column="customer_id" not-null="true" />
12
13    ...
14
15    <subclass name="com.freiheit.beans.CreditCard" discriminator-value="1">
16        <many-to-one name="creditCardCompany"
17                     class="com.freiheit.beans.CreditCardCompany"
18                     column="CREDITCARD_COMPANY_ID" />
19        <property name="validMonth" column="VALID_MONTH" type="integer" />
20        <property name="validYear" column="VALID_YEAR" type="integer" />
21    </subclass>
22
23    <subclass name="com.freiheit.beans.BankAccount" discriminator-value="2">
24        <many-to-one name="bank" class="com.freiheit.beans.Bank"
25                     column="BANK_ID" />
26    </subclass>
27 </class>
```

die verschiedenen Klassen des Objektmodells auf. Neben diesem als Table per Class Hierarchy bezeichneten Mapping kann Polymorphie auch noch mit Table per Subclass oder Table per Concrete Class auf mehrere Tabellen abgebildet werden.

Die Tabelle »PAYMENT_METHOD« speichert Zahlungsarten, im Beispiel Konto- oder Kreditkartendaten (Listing 7, Zeile 1). Das Element »discriminator« (Zeile 8) weist Hibernate an, je nach Wert des Felds »TYPE« verschiedene Objekte zu erzeugen, die für eine Zahlungsart stehen. Die Attribute »discriminator-value« in »subclass« (Zeilen 15 und 23) legen fest, welche Java-Bean für welchen Wert steht: Falls das Feld den Wert 1 besitzt, erzeugt Hibernate ein Objekt »CreditCard«, bei 2 einen »BankAccount«. Besitzen die Objekte unterschiedliche Parameter, lässt sich das mit weiteren »property«-Elementen innerhalb des »subclass«-Elements spezifizieren.

Die Klasse »Customer« besitzt Collections für »BankAccount« und »CreditCard«. Im entsprechenden Hibernate-Mapping unterscheidet das Attribut »where« die beiden polymorphen Typen, Listing 8, Zeilen 1 und 6. Listing 9 demonstriert, wie das Hauptprogramm diese Klassen verwendet.

Auch Many-to-Many-Verknüpfungen bildet Hibernate auf Collections ab. Dafür spezifiziert der Programmierer einfach nur die Tabelle, die die Relation herstellt, im Attribut »table« (Listing 10, Zeile 1). In der Anwendung unterscheidet sich die neu definierte Collection nicht von den vorher gezeigten, siehe Listing 11.

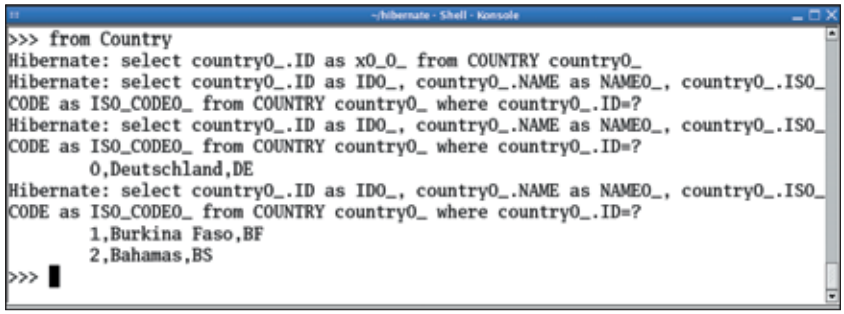

Abbildung 3: Die interaktive Abfrage »from Country« zeigt ID, Name und Kürzel aller Länder.

Bequeme Anfragen

Für Anfragen an die Datenbank gibt es die Hibernate Query Language (HQL), die das Framework nach SQL übersetzt und zur Laufzeit als Prepared Statements aufbereitet. Alles was Hibernate nicht als HQL erkennt, reicht es unverändert an die Datenbank weiter. Somit ist es möglich, spezielle Funktionen der Datenbank zu nutzen. HQL ähnelt syntaktisch SQL, ist aber objektorientiert. Die einfachste Abfrage gibt alle Länder mit Datenbank-ID, Namen und Kurzzeichen aus (Abbildung 3).

Listing 12 verwendet eine ähnliche Abfrage in der Java-Methode »session.createQuery()« (Zeile 2) und iteriert für die Ausgabe der Ergebnisse über die zurückgelieferte Liste (Zeilen 4 bis 7). Durch die Select-Klausel lässt sich einschränken, welche Beans oder Properties Hibernate ausgibt. Folgendes Statement zeigt eine Liste der Vornamen aller Kunden:

```
select c.firstname from Customer as c
```

Mit Hilfe der Where-Klausel lassen sich Einschränkungen definieren, die ähnlich wie in SQL funktionieren. Dieses Statement zeigt eine Liste aller Kunden an, deren Account nicht mehr aktiv ist:

```
from Customer as c where c.active = false
```

Um mit dieser Abfrage den Account-Status von außen zu setzen, ist sie folgendermaßen zu parametrisieren:

```
from Customer as c where c.active = :active
```

Beim erzeugten Query-Objekt ist der Wert des Parameters »active« mit Hilfe der Methode »setBoolean()« zu setzen, Listing 13, Zeile 3.

Natürlich kann man mit Hibernate auch joinen, also Abfragen über zwei oder mehr Tabellen laufen lassen. Die folgende Query ermittelt zum Beispiel alle Kunden, die eine Eurocard besitzen:

```
select c from Customer as c join
   c.creditCards as card where
   card.creditCardCompany.name='Eurocard'
```

Auch polymorphe Anfragen sind möglich. Eine Liste aller Zahlungsarten für einen Kunden liefert Hibernate mit folgender Anfrage:

```
from PaymentMethod as p where
   p.customer.id=42;
```

Eine Liste aller Kreditkarten des Kunden zeigt Hibernate mit Angabe der konkreten Subklasse der gemappten Bean:

```
from CreditCard as p where
   p.customer.id=42;
```

Listing 8: »Customer.hbm.xml«

```
01 <set name="bankAccounts" inverse="true" where="type=2" cascade="all-delete-orphan">
02    <key column="CUSTOMER_ID" />
03       <one-to-many class="com.freiheit.beans.BankAccount" />
04 </set>
05
06 <set name="creditCards" inverse="true" where="type=1" cascade="all-delete-orphan">
07    <key column="CUSTOMER_ID" />
08    <one-to-many class="com.freiheit.beans.CreditCard" />
09 </set>
```

Listing 9: »Main.java« (3)

```
02 ...
03 BankAcount bankAccount = new BankAccount();
04 bankAccount.setNumber( "12345" );
05 bankAccount.setOwner( "Hans Hansen" );
06 bankAccount.setBank( new Bank( 1 ) );
07
08 CreditCard creditCard = new CreditCard();
09 creditCard.setNumber( "5212345678901234" );
10 creditCard.setOwner( "Hans Hansen" );
11 creditCard.setValidMonth( 1 );
12 creditCard.setValidYear( 2009 );
13 creditCard.setCreditCardCompany( new
       CreditCardCompany( 2 ) );
14
15 customer.addBankAccount( bankAccount );
16 customer.addCreditCard( creditCard );
```

Neben den vorgestellten Funktionen bietet Hibernate weitere Features, die für den produktiven Einsatz oft unverzichtbar sind, etwa Transaktionen und optimistisches Locking. Mit dem Callback-Interface »Lifecycle« führt es vor jeder Datenbankoperation in der Bean zusätzliche Kommandos aus oder verhindert den Datenbankzugriff ganz.

Zusammenspiel

Inzwischen sind Tools entstanden, die dem Entwickler die Arbeit erleichtern, indem sie aus der Datenbankdeklaration Mappings für Hibernate erzeugen oder umgekehrt. Auch eine Unterstützung für XDoclet (siehe „Coffee-Shop“ in dieser Ausgabe) und Eclipse ist verfügbar. Viele Projekte bestätigen, dass Hibernate auch unter Hochlast einwandfrei funktioniert. So wird es auch von Bookzilla.de verwendet, dem Online-Buchhändler, der seine Provisionen an die FSF Europe spendet [4]. Die aktive Community erweitert und verbessert Hibernate ständig. Um keine neuen Funktionen zu verpassen, lohnt sich also ein regelmäßiger Blick auf die Website. *(ofr)*

Infos

[1] Hibernate: [http://www.hibernate.org/]
[2] Java-Beans-Spezifikation: [http://java.sun.com/products/javabeans/docs/spec.html]
[3] Bernhard Bablok, „Enterprise Java Beans“: Linux-Magazin 04/01, S. 152: [http://www.linux-magazin.de/Artikel/ausgabe/2001/04/Coffeshop/coffeeshop.html]
[4] Bookzilla: [http://www.bookzilla.de/]
[5] Listings zu diesem Artikels: [http://www.linux-magazin.de/Service/Listings/2004/09/Hibernate]

Die Autoren

Martin Schmidt und Thorsten Ehlers entwickeln Software bei Freiheit.com Technologies GmbH in den Bereichen Business-Anwendungen, Systemintegration und Robotics. Beide gehören zum Kernentwicklerteam von Bookzilla.de.

Listing 10: »Customer.hbm.xml«

```
01 <set name="keywords" table="KEYWORD_CUSTOMER_
   RELATION" cascade="all-delete-orphan">
02     <key column="CUSTOMER_ID" />
03     <many-to-many column="KEYWORD_ID"
   class="com.freiheit.beans.Keyword" />
04 </set>
```

Listing 11: »Main.java« (4)

```
01 Customer customer = new Customer();
02 ...
03 Keyword keyword = new Keyword("Stammkunde");
04 customer.addKeyword(keyword);
```

Listing 12: »Main.java« (5)

```
01 Session session = factory.openSession();
02 Query query = session.createQuery("from Country country");
03
04 List result = query.list();
05 for ( int i = 0; i < result.size(); i++ ) {
06     System.out.println( result.get( i ) );
07 }
```

Listing 13: »Main.java« (6)

```
01 query = session.createQuery("from Customer
02    as c where c.active = :active");
03 query.setBoolean("active", false);
```