HANSER

Richard Oates, Thomas Langer, Stefan Wille, Torsten Lueckow,
Gerald Bachlmayr

Spring & Hibernate

Eine praxisbezogene Einführung

ISBN-10: 3-446-41213-1
ISBN-13: 978-3-446-41213-2

Leseprobe

Weitere Informationen oder Bestellungen unter
http://www.hanser.de/978-3-446-41213-2
sowie im Buchhandel.

Kapitel 4

Einführung in Hibernate

In letzter Zeit taucht der Begriff Hibernate immer häufiger in verschiedenen Fachartikeln und anderen Veröffentlichungen auf. Gleichzeitig entstehen mehr und mehr Software-Architekturen, die Hibernate als Persistenz-Framework einsetzen. In diesem Kapitel wollen wir Ihnen die Aufgaben und Möglichkeiten von Hibernate kurz erläutern. Dazu werden wir auch einen ersten Blick auf die Architektur von Hibernate und deren Bestandteile werfen. Ziel dieses Kapitel ist es, dass Sie ein Verständnis dafür bekommen wie Hibernate arbeitet.

4.1 Hibernate als O/R-Mapper

Zu jeder Software-Architektur gehört fast immer eine relationale Datenbank, die für die Persistenz der verschiedenen Stamm- bzw. Bewegungsdaten verantwortlich ist. Für die Manipulation dieser Datenbestände (Einfügen, Bearbeiten und Löschen) sowie für die Abfrage der Daten steht mit der SQL (Structured Query Language) ein sehr mächtiges Sprachinstrument zur Verfügung. Jeder einzelne Entwickler muss für die Abbildung dieser relationalen Daten auf seine objektorientierten Daten innerhalb der Anwendung sorgen. Ein Framework, das diese Aufgabe automatisiert übernimmt, wird O/R-Mapper genannt. Mit der Zeit sind verschiedenste O/R-Mapper entstanden, die zum Teil einen sehr generischen Ansatz verfolgen und somit für viele fachliche und technische Umgebungen geeignet sind.

Wir wollen uns in diesem Buch mit Hibernate beschäftigen. Hibernate, ein Open-Source-Produkt, ist einer der bekanntesten O/R-Mapper und bietet dem Entwickler eine objektorientierte Sicht auf Tabellen und Beziehungen in einem relationalen Datenbank-Management-System. Doch welche Vorteile bietet uns Hibernate als O/R-Mapper und warum sollten wir uns gerade diesem Produkt widmen?

- Hibernate generiert die entsprechenden SQL-Anweisungen für uns und befreit uns von der manuellen Behandlung der JDBC-Ergebnisse. Bei der Gene-

rierung der SQL-Anweisungen verwendet Hibernate Datenbankdialekte und abstrahiert auf diese Weise ein konkretes Datenbankprodukt. Der Gewinn liegt in einer einfachen Portabilität bezüglich der verschiedenen Datenbanken. Allerdings versucht Hibernate nicht, die gesamte Stärke von SQL zu verstecken. Der Entwickler hat jederzeit die Möglichkeit, an geeigneten Stellen selbst den Datenbankzugriff direkt zu optimieren und noch besser an seine Bedürfnisse anzupassen.

- Im Gegensatz zu vielen anderen Persistenz-Frameworks können die verschiedenen fachlichen Klassen als einfache POJOs (Plain Old Java Objects) mit einem Default-Konstruktor realisiert werden. Sie müssen keine Hibernate-spezifische Basisklasse implementieren oder erweitern. Es ist also nicht notwendig, das fachliche Objekt-Modell von Hibernate abhängig zu machen.
- Hibernate bietet technisch ausgefeilte Abfrage-Möglichkeiten, um die für den jeweiligen Anwendungsfall benötigten Daten zu erhalten. Dabei kann sich der Entwickler auf die an SQL angelehnte objektorientierte Abfragesprache HQL (Hibernate Query Language) stützen oder mithilfe der Criteria-API programmatisch entsprechende Abfragen aufbauen. Ein Highlight ist sicherlich die Möglichkeit, mithilfe von Example-Objekten eine Abfrage abzusetzen, die Objekte findet, die ähnlich dem Example-Objekt sind. Sollten diese umfangreichen Möglichkeiten nicht ausreichen, ist es auch weiterhin erlaubt, direkt eine SQL-Abfrage an das System zu senden.
- Hibernate ist in der Lage, aus dem O/R-Mapping das Datenbankschema für eine bestimmte Datenbank zu generieren. Umgekehrt existieren Werkzeuge, die das Mapping aus einem bestehenden Datenbankschema erzeugen. Der letztere Weg ist naturgemäß holpriger, aber für nicht komplett exotische Datenbankschemata durchaus gangbar. Hier helfen die umfangreichen Konfigurationsmöglichkeiten von Hibernate.
- Gerade im Bereich von Datenbanken spielen Begriffe wie Transaktionen, Pooling und Caching eine große Rolle. Hibernate bietet hier die Möglichkeit, mit verschiedensten Implementierungen zusammenzuarbeiten. Hibernate stellt für die unterschiedlichen Aufgaben Schnittstellen zur Verfügung, über die externe Dienste angebunden werden können.
- Generell spielt es für Hibernate keine Rolle, ob es innerhalb einer Managed-Umgebung (z. B. JBoss) oder in einer Non-Managed-Umgebung eingesetzt wird. Allerdings bringt Hibernate eine herausragende JEE-Integration mit. Durch die Tatsache, dass Hibernate aktiv den EJB 3.0-Prozess begleitet, stellt es bereits heute eine erste Preview-Implementierung der EJB 3.0-Persistence-Provider-API bereit und kann somit als Persistence-Provider innerhalb eines EJB 3.0-Containers dienen. Hibernate kann innerhalb einer JEE-Architektur auch mithilfe der JMX-Integration konfiguriert und gesteuert werden.

4.2 Überblick über die Hibernate-Architektur

Die Abbildung 4.1 zeigt einen Überblick über die verschiedenen Konzepte, die von Hibernate verwendet werden. In diesem Abschnitt wollen wir Ihnen diese ein wenig näher bringen, damit Sie eine grobe Vorstellung davon haben, was sich hinter den einzelnen Begriffen versteckt.

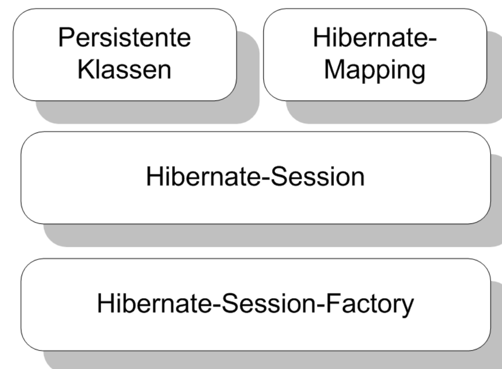

Abbildung 4.1: High-Level-Überblick über die Hibernate-API

4.2.1 Persistente Klassen

Die persistenten Klassen spielen bei einem O/R-Mapper eine zentrale Rolle, obwohl sie vom Aufbau sehr einfach sind. Sie enthalten oftmals nur eine Sammlung von Eigenschaften und entsprechenden Getter- und Setter-Methoden und folgen somit der Java-Beans-Spezifikation. Allerdings müssen die Zugriffsmethoden im Gegensatz zu dieser Spezifikation nicht *public* sein, da Hibernate sie auch verwenden kann, wenn sie *private* sind. Damit Hibernate die Klassen später auch instanziieren kann, müssen sie einen Default-Konstruktor besitzen.

Diese Klassen enthalten die Daten, die in der Datenbank gespeichert werden soll und stellen somit unser Domain-Modell dar. Als einfaches Beispiel stellen wir uns eine Pizza-Klasse vor, die neben einer eindeutigen Identifikationsnummer noch einen Namen besitzt. Wie bereits erwähnt, handelt es sich bei den persistenten Klassen um einfache Java-Objekte, die wir von keiner Hibernate-Klasse ableiten müssen.

Optional stellt Hibernate zwei Interfaces bereit, die von einer persistenten Klasse implementiert werden können. Die Schnittstelle `org.hibernate.classic.Lifecycle` ermöglicht dem Entwickler, bei einer Klasse eventuell gewünschte Initialisierungs- bzw. Aufräumarbeiten durchzuführen, nach dem die Instanz von Hibernate gespeichert oder geladen wurde und bevor sie gelöscht oder aktualisiert wird. Soll eine Instanz einer persistenten Klasse vor dem Speichern in der Datenbank noch die Möglichkeit erhalten, ihre Parameter mithilfe bestimmter Prüfungen zu validieren, dann kann das Interface `org.hibernate.classic.Validatable` implementiert werden.

In der Praxis würden wir Ihnen aber zunächst davon abraten, diese Interfaces zu verwenden, da die persistenten Klassen damit eine direkte Abhängigkeit zu Hibernate erhalten.

4.2.2 Die Hibernate-Session

Die Hibernate-`Session`[1] ist für den Entwickler die primäre Schnittstelle (Primary API) zu Hibernate. Mithilfe der `Session` können Objekte aus der Datenbank gelesen bzw. Änderungen an Objekten persistiert werden. Die folgende Tabelle zeigt die wichtigsten Operationen, die über eine `Session` ausgelöst werden können:

Tabelle 4.1: Die wichtigsten Methoden der Hibernate-Session

Funktion	Methode	Beschreibung
SELECT	Session.load(Class, Object)	Liest ein Objekt anhand des Primärschlüssels aus der Datenbank.
SELECT	Session.createQuery(String)	Erstellt eine Abfrage.
INSERT	Session.save(Object)	Speichert ein neues Objekt in der Datenbank.
DELETE	Session.delete(Object)	Löscht ein Objekt aus der Datenbank.
INSERT UPDATE DELETE	Session.flush()	Synchronisiert den Status der Objekte mit der Datenbank.

Neben der Funktion als Bindeglied zwischen der Datenbank und unserer Anwendung stellt die `Session` auch eine Factory für `Transaction`-Instanzen bereit. Gleichzeitig ist eine konkrete `Session`-Instanz nicht thread-safe und darf nur von einem Thread verwendet werden.

4.2.3 Hibernate-Session-Factory

Die `SessionFactory`[2] lädt und hält alle O/R-Mappings. Sie ist der Erzeuger der bereits erwähnten `Session`-Klasse. Im Gegensatz zur `Session` enthält unsere Anwendung genau eine `SessionFactory`, die als thread-safe und unveränderlich angesehen werden kann. Das bedeutet, dass die `SessionFactory` beim Start der Anwendung einmalig instanziiert und konfiguriert wird. Die dafür notwendige Konfiguration sollte im Klassenpfad als Property-Datei (z. B. *hibernate.properties*) oder als XML-Datei (z. B. *hibernate.cfg.xml*) vorliegen. Innerhalb dieser Datei werden die Datenbank-Verbindung sowie verschiedene weitere Einstellungen konfiguriert, was wir uns in der Folge noch genauer ansehen werden.

[1] `org.hibernate.Session`
[2] `org.hibernate.SessionFactory`

Zusätzlich ist die `SessionFactory` in der Lage, als Daten-Cache zwischen verschiedenen Transaktionen zu fungieren. Für diese Funktionalität greift Hibernate auf verschiedene Cache-Provider, wie z. B. EHCache, OSCache oder JBoss-Tree-Cache, zurück. Dieser Cache wird als Second-Level-Cache bezeichnet, da sich bereits innerhalb der Hibernate-Session der sogenannte First-Level-Cache befindet. Die Tatsache, dass der Second-Level-Cache direkt an die Session-Factory gebunden ist, hat zur Folge, dass dieser Cache so lange existiert, wie die Session-Factory lebt.

4.2.4 Hibernate-Mapping

Die Grundidee von O/R-Mapping ist, objektorientierte Klassen und ihre Eigenschaften auf Tabellen und Spalten einer relationalen Datenbank abzubilden. Vereinfacht gesagt, stellt ein O/R-Mapper einen Datensatz aus der Datenbank durch ein entsprechendes Objekt dar. Indem wir Änderungen am Objekt vornehmen, manipulieren wir letztlich den entsprechenden Inhalt einer Datenzeile.

Ein Merkmal von O/R-Mappern ist, dass sie keine starre 1:1-Beziehung von Klasse zu Tabelle und Eigenschaft zu Spalte erzwingen. Sie bieten an dieser Stelle eine erhebliche Flexibilität. Wie diese Abbildung aussieht, definiert der Entwickler im sogenannten Mapping. Es bildet die Brücke zwischen der Java- und der Datenbankwelt und bekommt dadurch die entscheidende Rolle innerhalb der Hibernate-Konfiguration.

Die Abbildung 4.2 veranschaulicht ein einfaches Mapping für unsere `Pizza`-Klasse.

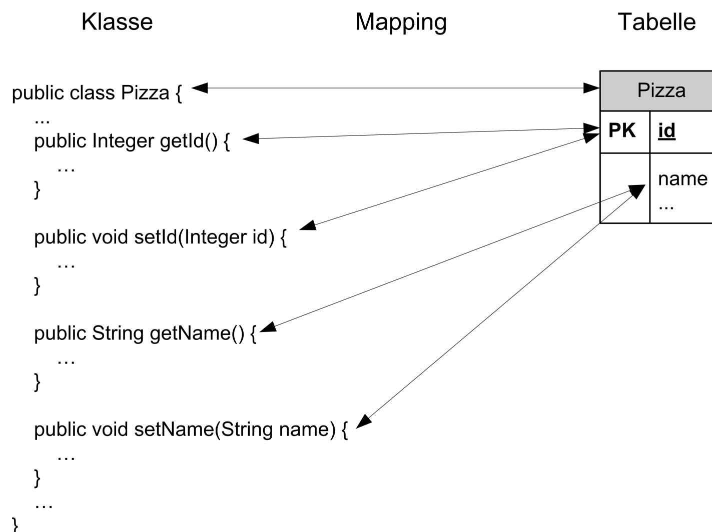

Abbildung 4.2: Abbildung zwischen Klassen und Tabellen durch das Mapping

In der Grafik bildet das Mapping die `Pizza`-Klasse auf die `pizza`-Tabelle und die Eigenschaften `id` und `name` auf die gleichnamigen Spalten ab.

Bei Hibernate legen wir unter anderen folgende Information im Mapping ab:

- Die Abbildung von Klassen auf Tabellen
- Die Abbildung von Eigenschaften auf Spalten
- Beschränkungen für Eigenschaften und Spalten, z. B. Not Null, Unique usw.
- Die Beziehungen zwischen den Tabellen und wie diese mit Java repräsentiert werden sollen – als Objektreferenz, als Collection usw.

Außer für das eigentliche O/R-Mapping verwenden die Hibernate-Tools, die als Zusatzpaket verfügbar sind, diese Daten auch zur Erstellung des Datenbank-Schemas und zur Generierung von Java-Klassen.

4.3 Hibernate in der Praxis

Was bedeutet all das für unsere Anwendung? Als Beispiel benutzen wir die `Pizza`-Klasse aus Opiz. Sie wird unsere erste persistente Klasse. `Pizza` präsentiert sich im folgenden Listing als einfache Java-Bean mit zwei Eigenschaften.

```
public class Pizza {
    private Integer id;
    private String name;

    /** Konstruktor für eine Pizza.  */
    public Pizza() {
    }

    /** Der Primary Key.  */
    public Integer getId() {
        return id;
    }

    public void setId(Integer anId) {
        this.id = anId;
    }

    /** Der Name der Pizza.  */
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

}
```

Damit Hibernate eine Java-Klasse als persistente Klasse benutzen kann, müssen wir ein paar Spielregeln einhalten:

- Die Klasse muss einen Default-Konstruktor oder gar keinen Konstruktor haben, um Hibernate die Instanziierung per Reflection zu ermöglichen. Die Sichtbarkeit (*public*, *private*, ...) spielt hier keine Rolle.

- Die Klasse muss die Property-Konvention einhalten. Ein Property ist ein abstraktes Konzept mit einem Namen und einem Typ. Eine Klasse hat ein Property `xyz`, wenn sie die Methoden `setXyz` und `getXyz` bzw. bei einem Boolean `isXyz` statt `getXyz` mit einem entsprechenden Typ bereitstellt. Die Werte der Properties halten wir typischerweise in Instanzvariablen, die als *private* markiert sind.
- Jede persistente Klasse benötigt zur Identifizierung ihrer Objekte ein spezielles Property, das sich Identifier-Property nennt und den Primärschlüssel aufnimmt.
- Persistente Klassen benötigen keine gemeinsame Basisklasse und müssen auch kein spezielles Interface implementieren.

Wenn wir die `Pizza`-Klasse auf die Spielregeln untersuchen, so stellen wir fest, dass alle Regeln eingehalten wurden. Wir haben einen Default-Konstruktor. Die beiden Properties `id` und `name` haben die nötigen Getter und Setter und mit dem Property `id` wird ein Identifier-Property zur Verfügung gestellt.

Dass persistente Klassen keine gemeinsame Basisklasse und kein gemeinsames Interface benötigen, heißt aber nicht, dass dies verboten wäre. Es kann z. B. durchaus sinnvoll sein, eine gemeinsame Basisklasse, wie `AbstractPersistent` mit Standard-Properties einzuführen. In diesem Buch machen wir davon allerdings keinen Gebrauch.

4.3.1 Mapping mit Annotations

Als nächsten Schritt wollen wir ein Mapping für die `Pizza`-Klasse definieren. Das folgende Bild zeigt die Tabelle, auf die wir unsere `Pizza`-Klasse abbilden:

pizza	
PK	id
	name

Die Tabelle `pizza` hat zwei Spalten. Die `id`-Spalte ist numerisch und bildet den Primärschlüssel. In der Grafik ist sie deshalb mit dem Kürzel „PK" gekennzeichnet. Die zweite Spalte `name` ist vom Typ `varchar`.

Die entsprechende SQL-Anweisung, um die Tabelle zu erzeugen, lautet:

```
create table pizza (
   id numeric(8) not null,
   name varchar(127),
   primary key (id))
```

In den Tests überlassen wir es Hibernate, aus den Mapping-Daten das Datenbankschema zu generieren. Wir könnten diese Tabelle mit der SQL-Anweisung natürlich auch von Hand erstellen.

Das zu erstellende Mapping soll Folgendes festlegen:

- Die `Pizza`-Klasse wird auf die `pizza`-Tabelle gemappt.
- `id` ist der Primärschlüssel bzw. Identifier.
- Das `id`-Property wird auf die `id`-Spalte gemappt.
- Entsprechend wird `name` auf `name` gemappt.

Um diese Informationen Hibernate zu vermitteln, bieten sich uns zwei verschiedene Möglichkeiten. Der klassische Weg für ein Hibernate-Mapping ist eine XML-Datei, die zu jeder persistenten Klasse die entsprechende Information enthält. Am Ende des nächsten Kapitels gibt es dazu ein Beispiel.

Im Buch verwenden wir den moderneren Weg über Java 5-Annotations. Mit ihrer Hilfe lässt sich das Mapping direkt in den Java-Klassen angeben. Wir bevorzugen diesen Ansatz aus drei Gründen:

- Wir können uns auf eine Datei beschränken, anstatt zwischen Java- und Mapping-Datei hin und her wechseln zu müssen.
- Wir müssen die Property-Namen nur an einer Stelle angeben, nicht an zwei. Dadurch beseitigen wir eine Fehlerquelle.
- Annotations sind beim Java Persistence API (JPA), auf dem EJB3 basiert, das Mittel der Wahl und die meisten mit Hibernate verwendeten Annotations stammen direkt aus dem JPA-Standard. Damit sind unsere gemappten Klassen weitgehend JPA-konform.

Nachteilig an dem Annotations-Ansatz ist, dass wir Getter- und Setter-Methoden sowie `equals`, `hashCode` und `toString` selbst schreiben und pflegen müssen. Die Vorteile überwiegen aber die Nachteile.

Erweitern wir also die `Pizza`-Klasse um Mapping-Annotations:

```
@Entity
@Table(name = "pizza")
public class Pizza {
    private Integer id;
    private String name;

    /** Konstruktor. */
    public Pizza() {
    }

    /** Der Primary Key. */
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    public Integer getId() {
        return id;
    }

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

```
    /** Der Name der Pizza. */
    @Column(name = "name")
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Damit Hibernate die `Pizza`-Klasse als persistent erkennt, markieren wir sie mit der `@Entity`-Annotation. Dies ist eine Standard-Annotation aus der JPA und kommt aus dem Package *javax.persistence*, so wie die meisten Mapping-Annotations. Mit der nächsten Annotation *@Table* geben wir den Namen der Tabelle an, auf die die Klasse gemappt werden soll. Weil Klasse und Tabelle den gleichen Namen haben, könnten wir in diesem Fall auf die Angabe einer Tabelle mit `@Table` verzichten.

Danach markieren wir die Getter-Methode für das Property `id` mit `@Id`. Dadurch erkennt Hibernate `id` als das Identifier-Property. Weil Hibernate per Voreinstellung annimmt, dass die Property- und Spaltennamen gleich sind, brauchen wir den Spaltennamen hier nicht anzugeben. Wichtig ist hier noch einmal, dass die Hibernate-Annotation für ein Property bei der Getter-Methode angegeben wird und nicht beim Klassenfeld.

`@GeneratedValue` definiert, dass wir den Primärschlüssel nicht selbst bereitstellen, sondern von Hibernate automatisch generieren lassen wollen. Das Verfahren dafür lassen wir Hibernate selbst aussuchen (*strategy = AUTO*). Die verschiedenen Primärschlüsselgenerierungsverfahren betrachten wir im nächsten Kapitel genauer.

Danach mappen wir das `name`-Property. Per Voreinstellung sind alle Properties persistent. Wir bräuchten deshalb für `name` keine Mapping-Informationen angeben. Zur Veranschaulichung des Mapping-Konzepts geben wir trotzdem die `@Column`-Annotation an, die den Spaltennamen festlegt.

4.3.2 Konfiguration der Session-Factory

Nun haben wir eine kommentierte `Pizza`-Klasse mit der nötigen Mapping-Information und eine passende Tabelle. Jetzt fehlt uns nur noch eine Sache. Hibernate benötigt eine Datei, die die oben beschriebene `SessionFactory` konfiguriert. Diese enthält die Parameter für die Datenbank-Verbindung (JDBC-Treiber, URL, Username und Password). Zusätzlich müssen wir dort noch unsere gemappten Klassen auflisten, damit Hibernate mit ihnen arbeiten kann. Üblicherweise heißt die Datei *hibernate.cfg.xml* und liegt im *CLASSPATH* im Wurzelverzeichnis, sodass die `SessionFactory` sie dort automatisch finden kann.

Hier finden Sie ein Beispiel für die *hibernate.cfg.xml*:

```
1 <!DOCTYPE hibernate-configuration PUBLIC
2     "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
3     "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
```

```
4  <hibernate-configuration>
5      <session-factory>
6
7          <!-- Datenbank Connection Einstellungen -->
8          <property name="connection.driver_class">
               org.hsqldb.jdbcDriver</property>
9          <property name="connection.url">jdbc:hsqldb:mem:pizza</property>
10         <property name="connection.username">sa</property>
11         <property name="connection.password"></property>
12
13         <property name="hibernate.dialect">
               org.hibernate.dialect.HSQLDialect</property>
14
15         <!-- Zusaetzliche Hibernate-Properties -->
16         <property name="hibernate.show_sql">true</property>
17         <property name="hibernate.format_sql">true</property>
18         <property name="hibernate.hbm2ddl.auto">create</property>
19
20         <!-- Auflistung der gemappten Klassen -->
21         <mapping class="de.hanser.buch.opiz.domain.Pizza"/>
22
23     </session-factory>
24 </hibernate-configuration>
```

Zunächst definiert die Datei die DTD, damit der XML-Parser den Inhalt validieren kann. Dann kommt im Element *<session-factory>* eine Folge von *<property>*-Elementen, deren Bedeutung exakt den Einträgen einer *properties*-Datei entspricht: Sie definieren jeweils ein Name/Wert-Paar, also ein Property. Die ersten Einträge definieren die Eigenschaften der JDBC-Verbindung:

- *connection.driver_class* – der Klassenname des JDBC-Treibers
- *connection.url* – die URL
- *connection.username* – der Benutzername
- *connection.password* – das Passwort

In dem Beispiel verwenden wir eine HSQL-Datenbank mit den entsprechenden Einstellungen (Zeilen 8–11).

Das nächste *<property>*-Element *hibernate.dialect* definiert den SQL-Dialekt, den Hibernate mit der Datenbank sprechen soll. Über den Dialekt passt sich Hibernate an die Besonderheiten der jeweiligen SQL-Implementation an. Der Dialekt für Oracle 8 setzt zum Beispiel die in Oracle proprietäre Syntax für Outer Joins um.

Mit dem nächsten *<property>*-Element *hibernate.show_sql* schalten wir das Logging der von Hibernate generierten SQL-Anweisungen ein. Das folgende Property *hibernate.format_sql* sorgt dabei für eine gut lesbare Darstellung mit Einrückungen.

Das letzte *<property>*-Element *hibernate.hbm2ddl.auto* lässt Hibernate bei der Initialisierung der `SessionFactory` das Datenbankschema neu erzeugen. Dies ist für uns besonders nützlich, weil wir in diesem Buch häufiger Änderungen am Schema vornehmen. Für eine Produktionssituation ist das aber gefährlich, weil bei dem Neuerzeugen der Tabellen alle Daten verloren gehen.

Nach den Properties listen wir mit dem Element <*mapping*> alle persistenten Klassen auf, um sie Hibernate bekannt zu machen. Da wir bis jetzt nur eine persistente Klasse haben, haben wir auch nur ein <*mapping*>-Element.

> ***Achtung!***
> Die angegebene Konfiguration verwendet die HSQL-Datenbank in der Memory-Variante. Das heißt, die Datenbank speichert ihre Tabellen im RAM. Dadurch sind alle Operationen sehr schnell, was für Tests wunderbar ist. In der Konsequenz sind aber nach dem Ende der Tests auch alle gespeicherten Daten wieder verloren. Sie haben deshalb keine Möglichkeit, sich mit einem separaten Tool die Tabelleninhalte anzusehen.

4.3.3 Verzeichnisstruktur

Damit haben wir alle benötigten Teile zusammen. Nun soll noch kurz die Verzeichnisstruktur unseres Projekts zusammengefasst werden.

- Der Quelltext für die `Pizza`-Klasse *Pizza.java* liegt im Source-Verzeichnis in einem Unterverzeichnis entsprechend dem Package-Namen *de.hanser.buch.opiz.domain*.
- *hibernate.cfg.xml* liegt im Source-Verzeichnis im Wurzelverzeichnis, damit die Datei beim Compiler-Lauf im *CLASSPATH* landet.
- Testklassen, die darauf aufbauen, erscheinen ebenfalls im Source-Verzeichnis oder in einem separaten Test-Verzeichnis.

Abbildung 4.3 auf der nächsten Seite stellt unsere Verzeichnisstruktur grafisch dar, wobei wir *src* als Source- und *test* als separates Test-Verzeichnis verwenden. Diese Struktur finden Sie auch im Begleit-Quelltext wieder. Als Grundlage für ein Eclipse-Projekt können Sie das in Kapitel 2 beschriebene Basis-Projekt importieren.

4.3.4 Initialisierung der Session-Factory

Als Nächstes wollen wir eine `SessionFactory` initialisieren und eine `Session` öffnen. Dazu laden wir zunächst die Konfiguration aus der Datei *hibernate.cfg.xml*:

```
AnnotationConfiguration configuration = new AnnotationConfiguration();
configuration.configure();
```

Der Dateiname ist bei Hibernate auf *hibernate.cfg.xml* voreingestellt. Nun können wir die `SessionFactory` initialisieren:

```
SessionFactory sessionFactory = configuration.buildSessionFactory();
```

Für unsere Tests verwenden wir auch gerne folgende Kurzschreibweise:

```
SessionFactory sessionFactory = new
    AnnotationConfiguration().configure().buildSessionFactory();
```

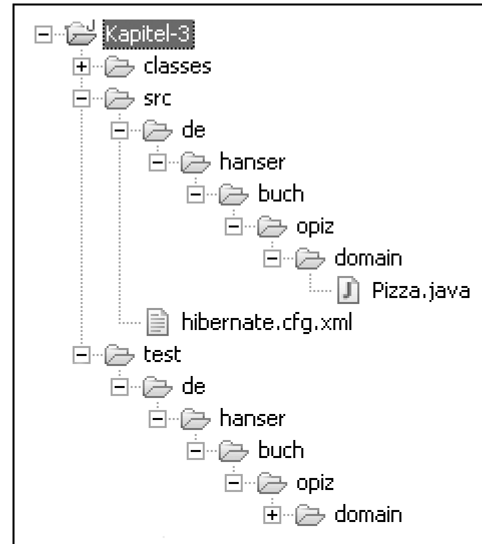

Abbildung 4.3: Verzeichnisstruktur unseres Projekts

Als Nächstes können wir mit der `SessionFactory` eine `Session`, die zentrale Schnittstelle zu Hibernate, öffnen:

```
Session session = sessionFactory.openSession();
```

Jetzt steht der Benutzung unserer gemappten Klasse nichts mehr im Wege. In den nächsten Abschnitten sehen wir uns an, wie wir mit der `Session` die grundlegenden Datenbank-Operationen, die sogenannten CRUD-Operationen (Create, Read, Update, Delete), ausführen.

4.3.5 Speichern eines Objekts

In unserem ersten Test wollen wir eine Pizza in die Datenbank einfügen. Wir formulieren den Test mit JUnit. Dazu legen wir uns einen JUnit-Test in einer Klasse *de.hanser.buch.opiz.domain.SaveTest* an und starten mit einer leeren Testmethode:

```
public class SaveTest extends TestCase {
    public void testSave() {
    }
}
```

In dieses JUnit-Fragment schreiben wir unseren Code.

Als Erstes öffnen wir eine `Session` wie oben:

```
Session session = sessionFactory.openSession();
```

Alle Aktionen mit einer Session sollen bei Hibernate innerhalb einer Transaktion ablaufen. Deshalb öffnen wir eine Transaktion:

```
Transaction transaction = session.beginTransaction();
```

Als Nächstes erzeugen wir das `Pizza`-Objekt, das wir in die Datenbank einfügen wollen:

```
Pizza pizza = new Pizza();
pizza.setName("Thunfisch");
```

Weil der Primärschlüssel von Hibernate generiert wird, belegen wir nur das `name`-Property, während der Wert für `id` *null* bleibt.

Nun wird es interessant: Wir speichern das `Pizza`-Objekt. Dafür bietet das `Session`-API die Methode `save`:

```
Serializable save(Object object) throws org.hibernate.HibernateException
```

Die Methode `save` fügt ein Objekt in die Datenbank ein, das dort noch nicht existiert. Dazu erzeugt die Methode ein `INSERT`- Statement. Da wir automatisch generierte Primärschlüssel verwenden, generiert `save` vorher einen Identifier und weist ihn dem Identifier-Property zu. Wir erweitern also unseren Test um einen Aufruf von `save`:

```
session.save(pizza);
```

Wenn die `save`-Methode ihre Arbeit gemacht hat, darf das `id`-Property nun nicht mehr *null* sein:

```
assertNotNull(pizza.getId());
```

Das war es im Wesentlichen. Am Ende müssen wir lediglich die Transaktion festschreiben und die `Session` sowie die `SessionFactory` schließen:

```
transaction.commit();
session.close();
sessionFactory.close();
```

Damit Sie das Ganze im Zusammenhang sehen können, finden Sie im folgenden Listing den vollständigen Quelltext.

Listing 4.1: Die `SaveTest`-Klasse

```
1  package de.hanser.buch.opiz.domain;
2
3  import junit.framework.TestCase;
4  import org.hibernate.*;
5  import org.hibernate.cfg.AnnotationConfiguration;
6
7  public class SaveTest extends TestCase {
8
9      public void testSave() {
10
11         AnnotationConfiguration configuration = new
               AnnotationConfiguration();
12         configuration.configure();
13         SessionFactory sessionFactory =
               configuration.buildSessionFactory();
14         Session session = sessionFactory.openSession();
15         Transaction transaction = session.beginTransaction();
16
```

```
17          Pizza pizza = new Pizza();
18          pizza.setName("Thunfisch");
19
20          session.save(pizza);
21
22          assertNotNull(pizza.getId());
23
24          transaction.commit();
25          session.close();
26          sessionFactory.close();
27      }
28  }
```

Sie finden diesen Test auch im Begleit-Quelltext unter dem Namen *SaveTest.java*. Am besten probieren Sie ihn selbst aus.

> ***Achtung!***
> Beim Speichern versucht Hibernate, anhand des Identifier-Properties zu entscheiden, ob ein Objekt bereits in der Datenbank existiert und ob deshalb ein INSERT oder ein UPDATE nötig ist. Bei referenzbasierten Typen wie *Long* und *Integer* gilt per Voreinstellung *null* als sogenannter Unsaved Value, also als Marker für ungespeicherte Objekte. Für einfache Typen wie *long* und *int* wird die Sache unnötig kompliziert, weshalb wir in diesem Buch nur nullable Typen für den Identifier verwenden.

Hibernate arbeitet asynchron

Aktionen, die die Datenbank verändern, führt Hibernate asynchron aus. Wenn wir auf der `Session` beispielsweise `save` aufrufen, erzeugt Hibernate das dazugehörige `INSERT`-Statement in der Regel nicht sofort. Stattdessen werden die ausgeführten Aktionen in einer internen Buchführung vermerkt. Erst später führt Hibernate sie bei einer speziellen Aktion namens Flush aus.

Ein Flush lässt sich zum einen manuell über die `Session`-Methode `flush` auslösen:

```
void flush() throws HibernateException;
```

Zum anderen hat die Session einen Flush-Mode, der bestimmt, zu welchen Zeitpunkten Hibernate automatisch einen Flush auslöst. Der Flush-Mode lässt sich an der `Session` mit der Eigenschaft `flushMode` einstellen:

```
void session.setFlushMode(FlushMode flushMode)
FlushMode getFlushMode()
```

Die möglichen Werte für `flushMode` lauten:

- `ALWAYS`: Vor jeder Abfrage erfolgt ein Flush.
- `AUTO`: Vor manchen Abfragen erfolgt ein Flush, um sicherzustellen, dass konsistente Daten geliefert werden.
- `COMMIT`: Vor jedem Commit erfolgt ein flush.

- MANUAL: Nur bei einem expliziten flush-Aufruf erfolgt ein Flush, sonst nie.

Die Voreinstellung ist AUTO.

> ***Achtung!***
> Hibernate verzögert die Ausführung von SQL-Anweisungen, weil dadurch häufig mehrere Aktionen in einem einzigen Batch-Statement ausgeführt werden können. Dies kann die Performance wesentlich verbessern. Bei der Vermischung von Hibernate-Code mit JDBC-Code kann das allerdings auch verwirren, weil JDBC – anders als Hibernate – SQL-Anweisungen synchron behandelt.

4.3.6 Laden eines Objekts

Mit Hibernate können wir Objekte natürlich nicht nur speichern, sondern auch laden. Das Session-API bietet dafür die Methode load. Sie lädt ein Objekt anhand seines Primärschlüssels:

```
Object load(Class aClass, Serializable id) throws HibernateException
```

Der erste Parameter ist die Klasse des Objekts, das wir laden wollen. Der zweite Parameter ist der Primärschlüssel.

In diesem Buch beschränken wir uns auf den Fall, bei dem der Primärschlüssel aus einem primitiven Java-Typ gebildet wird, insbesondere einem Integer oder Long. In diesem Fall genügt der Aufruf:

```
Long id = 5;
Pizza p = (Pizza) session.load(Pizza.class, id);
```

In dem Beispiel lädt die Session ein Pizza-Objekt mit dem Primärschlüssel 5. Dazu setzt die Session ein SELECT-Statement ab. Wenn Hibernate keinen passenden Datensatz findet, bekommen wir eine HibernateException.

Alternativ zur load-Methode bietet die Session auch die Methode get:

```
Object get(Class aClass, Serializable id) throws HibernateException
```

Ihr Verhalten entspricht dem von load, nur dass sie *null* liefert, anstatt eine Exception zu werfen, wenn es keinen Datensatz mit dem übergebenen Primärschlüssel gibt.

> ***Tipp***
> Wenn Hibernate eine HibernateException wirft, ist die Session in einem undefinierten Zustand und muss geschlossen werden – so schreibt es die Referenzdokumentation vor. Deshalb ist es in der Regel sinnvoller, get statt load zu verwenden – außer, wir sehen ein fehlgeschlagenes load als einen Systemfehler an.

Der folgende Code-Ausschnitt zeigt, wie Sie die `get`-Methode in einem Test ausprobieren können:

```
....
Transaction transaction = session.beginTransaction();
Pizza pizza = new Pizza();
pizza.setName("Thunfisch");
session.save(pizza);
Integer id = pizza.getId();
transaction.commit();

transaction = session.beginTransaction();
Pizza reloadedPizza = (Pizza) session.get(Pizza.class, id);
assertNotNull(reloadedPizza);
assertEquals("Thunfisch", reloadedPizza.getName());
transaction.commit();
...
```

In der ersten Transaktion speichert das Beispiel ein `Pizza`-Objekt, um es dann in der zweiten Transaktion mit `get` wieder zu laden.

Nachdem Sie im vorherigen Abschnitt in Listing 4.1 einen vollständigen Beispieltest gesehen haben, der ein `Pizza`- Objekt in die Datenbank einfügt, dürfte Ihnen klar sein, wie Sie ihn um das Code-Fragment von oben erweitern. Sie finden außerdem im Begleit-Quelltext eine Testklasse `GetTest`, die eigenständig lauffähig ist und `get` demonstriert.

4.3.7 Suchen eines Objekts

Natürlich bieten relationale Datenbanken sehr viel leistungsfähigere Möglichkeiten für den Zugriff als nur den Weg über den Primärschlüssel. SQL erlaubt die Formulierung komplexer und umfangreicher Abfragen. Und so kommt auch Hibernate mit einer Abfragesprache namens HQL (Hibernate Query Language) daher. HQL wurde ursprünglich locker als Erweiterung von EJBQL konzipiert, welche wiederum auf SQL basiert. Deshalb sehen HQL-Abfragen heute wie ein objektorientiertes SQL aus. HQL bietet alle wesentlichen Features von SQL, ist in einigen Belangen aber deutlich mächtiger und bequemer zu benutzen.

Hier ein erstes Beispiel für eine typische HQL-Abfrage:

```
select p from Pizza p where p.name like 'M%'
```

Diese Abfrage sieht exakt wie ihr SQL-Pendant aus. Allerdings interpretiert Hibernate sie mit den Namen der gemappten Klassen und deren Properties. Deshalb bezieht sich „Pizza“ auf die Klasse `Pizza` und „p.name“ auf das Property `name` der `Pizza`-Klasse. Mit etwas SQL-Erfahrung ist es leicht, diese Abfrage zu interpretieren: Sie liefert alle Pizzen, deren Name mit ‚M‘ anfängt.

Um eine HQL-Abfrage auszuführen, bietet die `Session` uns die `createQuery`-Methode:

```
Query createQuery(String queryString) throws HibernateException
```

Diese Methode erzeugt aus unserem Abfrage-String ein `Query`-Objekt. Darauf können wir verschiedene Eigenschaften setzen, z.B. `Query`-Parameter. Anschließend werten wir über die `list`- Methode die Abfrage aus:

```
List list() throws HibernateException
```

Bei der `list`-Methode setzt Hibernate den Abfrage-String intern in ein oder teilweise sogar mehrere `SELECT`-Statements um. Das Ergebnis ist eine `List` von Objekten.

Hier ein konkretes Beispiel, das den Abfrage-String von oben ausführt:

```
Query query = session.createQuery(
        "select p from Pizza p where p.name like 'M%'");
List objects = query.list();
```

Der Typ der Ergebnisliste richtet sich nach dem Abfrage-String. In unserem Beispiel liefert `list Pizza`-Objekte zurück. Hier zeigt sich ein Vorteil des O/R-Mappings: Bei JDBC müssten wir das JDBC-`ResultSet` manuell in Objekte umwandeln. Hibernate erledigt das für uns automatisch.

> *Achtung*
> Weil HQL auf Klassen und Properties basiert, ist dort für Namen die Groß- und Kleinschreibung relevant – anders als bei SQL. Wenn Sie die Schreibweise nicht beachten, wird Ihnen Hibernate eine Exception präsentieren, weil es einen entsprechenden Namen nicht kennt.

Wenn wir höchstens ein Element als Ergebnis erwarten, können wir auch die Methode `uniqueResult` verwenden:

```
Object uniqueResult() throws HibernateException
```

`uniqueResult` liefert das einzige Abfrageergebnis zurück, oder *null*, wenn das Ergebnis leer ist. Das ist in manchen Situationen kürzer als `list`. Wenn es mehr als ein Ergebnis gibt, wirft `uniqueResult` eine `HibernateException`.

HQL bietet die aus JDBC bekannten Fragezeichen als Platzhalter für Abfrage-Parameter. Zum Beispiel:

```
select p from Pizza p where p.name = ?
```

Die konkreten Werte für die Platzhalter setzen wir an dem `Query`-Objekt:

```
Query query = session.createQuery (
                  "select p from Pizza p where p.name = ?");
query.setString(0, "Salami");
List objects = query.list();
```

In dem Beispiel setzt `setString` den ersten Abfrage-Parameter, nämlich den mit dem Index 0, auf den Wert *„Salami“*. In Hibernate beginnen die Indizes von Abfrage-Parametern bei 0 – anders als in JDBC. Die `Query`-Schnittstelle stellt uns entsprechende Methoden, `setInteger`, `setBigDecimal` usw., für andere Typen bereit.

Neben den Fragezeichen, die auch als anonyme Platzhalter bezeichnet werden, bietet Hibernate auch benannte Platzhalter. Wir markieren sie in einer HQL-Abfrage durch einen Namen mit vorangestelltem Doppelpunkt:

```
select p from Pizza p where p.name = :name
```

Um den Wert des Platzhalters anzugeben, gibt es wieder Methoden, `setString` usw., im `Query`-API, die statt des Parameter-Indexes den Platzhalternamen erwarten:

```
Query query = session.createQuery (
                 "select p from Pizza p where p.name = :name");
query.setString("name", "Salami");
List objects = query.list();
```

Ein vollständiges Beispiel finden Sie im Begleit-Quelltext in der Klasse `QueryTest`. Kapitel 7 geht genauer auf HQL und die `Query`-Schnittstelle ein.

4.3.8 Verändern eines Objekts

Der Weg, um ein persistentes Objekt in Hibernate zu verändern, ist für viele Entwickler zunächst überraschend. Die Intuition sagt, dass man zuerst ein Objekt lädt, es mit den Setter-Methoden verändert und dann mit einer Methode wie `save` wieder speichert. So funktioniert es aber nicht!

Ein Objekt, das mit `save` in der Datenbank gespeichert wird, bleibt mit seiner `Session` verbunden, bis sie geschlossen wird. Wenn wir nun Änderungen an dem Objekt vornehmen, erkennt Hibernate das automatisch und speichert die Änderungen durch ein `UPDATE`-Statement in der Datenbank. Dies nennt sich automatisches Dirty-Checking. Ähnliches wie bei `save` gilt für Objekte, die wir mit `load` oder `get` laden. Sie bleiben ebenfalls mit der `Session` verbunden, und damit gilt auch für sie das automatische Dirty-Checking.

Konkret verändern wir ein `Pizza`-Objekt mit folgendem Fragment:

```
Pizza pizza = (Pizza) session.load(Pizza.class, id);
pizza.setName("Salami");
```

Wir laden ein `Pizza`-Objekt. Dieses bleibt mit der `Session` assoziiert. Dann verändern wir das Objekt, indem wir den Namen setzen. Hibernate überprüft beim nächsten Flush den Objekt-Zustand und erzeugt ein `UPDATE`-Statement.

Dass Objekte mit der `Session` assoziiert werden und über deren Lebenszeit auch bleiben, ist bei Hibernate ein zentrales Konzept. Wenn die Session geschlossen wird, löst Hibernate die Objekte von der Session. Sie heißen dann „detached“. Für solche Objekte gelten andere Spielregeln – mehr dazu in Abschnitt 4.4. Deshalb sollten wir uns immer darüber im Klaren sein, wie die Lebensdauer der benutzten `Session` ist.

Zum automatischen Dirty-Checking finden Sie ein vollständiges Beispiel im Begleit-Quelltext als *UpdateTest.java*.

4.3.9 Löschen eines Objekts

Um ein Objekt aus der Datenbank zu löschen, bietet Hibernate uns die `Session`-Methode `delete`:

```
void delete(Object object) throws HibernateException
```

`delete` löst ein SQL-`DELETE`-Statement aus. Hier ein Beispiel mit der `Pizza`-Klasse:

```
Pizza pizza = (Pizza) session.load(Pizza.class, 5);
session.delete(pizza);
```

Das Beispiel lädt ein `Pizza`-Objekt und löscht es anschließend. Um ein Objekt löschen zu können, muss es nicht mit der `Session` verbunden sein.

Ein vollständiges Beispiel finden Sie im Begleit-Quelltext als *DeleteTest.java*.

4.3.10 Exception-Handling

In den bisherigen Beispielen haben wir die `HibernateException`-Exceptions, die die einzelnen `Session`-Methoden werfen können, einfach ignoriert. In produktionstauglichem Code können wir uns das leider nicht erlauben, weil sonst bei einer Exception die Session weiter geöffnet bleibt und Ressourcen unnötig verschwendet werden. Wir müssen deshalb die Session schließen, wenn eine Exception auftritt. Außerdem müssen wir einen Rollback auf der Transaktion auslösen. Daraus ergibt sich für `Session`-basierten Code folgendes Muster:

```
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();
try {
    // Aktionen mit der Session durchführen
    ...
    transaction.commit();
}
catch (Throwable t) {
    transaction.rollback();
    throw t;
} finally {
    session.close();
}
```

Dieses Muster schließt durch das *try . . . finally* zuverlässig die Session und führt im Fehlerfall ein Rollback aus.

Eigentlich müssten wir so eine Code-Sequenz für jede Hibernate-basierte Aktion in den Quelltext kopieren. Das ist natürlich sehr unschön. Sobald wir in Kapitel 7 die Hibernate-Unterstützung von Spring kennengelernt haben, steht uns ein besserer Ansatz zur Verfügung. Deshalb verzichten wir in den reinen Hibernate-Beispielen zu Gunsten der Lesbarkeit auf das Exception-Handling.

4.3.11 Logging

In manchen Situationen, sei es beim Debugging, Testen oder Profilen, ist es hilfreich, sich die von Hibernate generierten SQL-Statements anschauen zu können.

Ein schneller Weg zum Erfolg ist das Setzen des Parameters *show_sql* an der Konfiguration für die Session-Factory. Wenn wir auch noch den Parameter *format_sql* auf *true* setzen, geschieht die Ausgabe auch noch in gut lesbarer Form.

```
<prop key="hibernate.show_sql">true</prop>
<prop key="hibernate.format_sql">true</prop>
```

Diese Einstellung ist nur für Testzwecke geeignet und funktioniert auch eher aus rein historischen Gründen, denn Hibernate schreibt mit diesen Einstellungen direkt in `System.out`, was uns wenig Steuerungsmöglichkeiten bei der Ausgabe einräumt.

Alternativ nutzt Hibernate das Apache-Jakarta Commons Logging[3], ein Art Portal-API für verschiedene Logging-Implementationen. In unserem Beispielprojekt verwenden wir die Implementation von *log4j*[4]. Die Konfiguration von *log4j* befindet sich standardmäßig in der Datei *log4j.xml*, die im Klassenpfad bereitgestellt wird.

Um das Loggen der SQL-Statements in Hibernate einzuschalten, muss der Logger *org.hibernate.SQL* in den `DEBUG`-Level versetzt werden:

```
<logger name="org.hibernate.SQL">
    <level value="DEBUG"/>
</logger>
```

Einen weiterer hilfreicher Logger bei der Fehlersuche ist der *org.hibernate.type*. Befindet sich dieser Logger im `DEBUG`-Level, so werden bei Prepared-Statements auch die Werte der gebundenen Variablen herausgeschrieben.

```
<logger name="org.hibernate.type">
    <level value="DEBUG"/>
</logger>
```

Zu guter Letzt muss in der *log4j*-Konfiguration noch ein *appender* existieren, der Logging-Ausgaben im `DEBUG`-Level durchlässt. Die folgende Konfiguration würde solche Ausgaben in die Konsole schreiben.

```
<appender name="STDOUT" class="org.apache.log4j.ConsoleAppender">
    <param name="threshold" value="DEBUG"/>
    <layout class="org.apache.log4j.PatternLayout">
        <param name="ConversionPattern" value="%d %p [%c] - %m%n"/>
    </layout>
</appender>
```

Weitere Möglichkeiten zur Steuerung der eigentlichen Ausgabe des Loggers entnehmen Sie bitte der Dokumentation zu *log4j*.

[3] http://jakarta.apache.org/commons/logging
[4] http://logging.apache.org/log4j

4.4 Persistency-Lifecycle

Jedes Objekt hat aus Sicht von Hibernate einen von drei Zuständen: *persistent*, *detached* und *transient*. Deren Unterschiede sind entscheidend für das Verhalten von Hibernate.

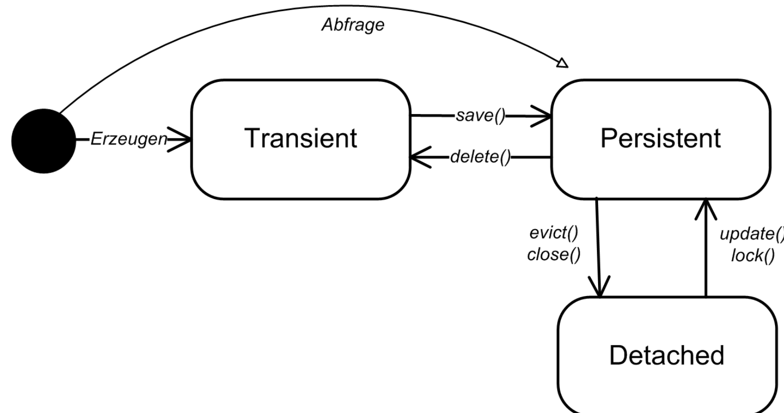

Abbildung 4.4: Lifecycle eines Hibernate-Objekts

Die Objekte, die der Hibernate-`Session` noch nicht bekannt sind, z. B. weil sie gerade mit dem `new`-Operator erzeugt worden sind, werden als *transiente* Objekte bezeichnet. Diese Objekte haben natürlich noch keine Datenbank-Repräsentation. Sobald diese Objekte nicht mehr von anderen Objekten referenziert werden, schlägt der Garbage-Collector von Java zu und unsere Objekte sind für immer verloren.

Wie in Abbildung 4.4 zu sehen ist, gibt es noch eine weitere Möglichkeit ein *transientes* Objekt zu erhalten. Sollten wir mithilfe der `delete`-Methode ein Objekt aus der Datenbank entfernen, verliert es seine Repräsentation in der Datenbank und wird wieder wie ein normales transientes Objekt behandelt.

Die Objekte, die Hibernate in der aktuellen Session bekannt sind, werden als *persistente* Objekte bezeichnet. Sie haben eine Repräsentation in der Datenbank und einen Wert für ihren Primärschlüssel. Der Zustand dieser Objekte wird von Hibernate verfolgt, das heißt ein explizites Speichern von Änderungen ist nicht notwendig. Wir erhalten ein persistentes Objekt, wenn wir z. B. unser neu angelegtes transientes Objekt mithilfe der `save`-Methode bei der `Session` persistieren. Alle Objekte, die als Ergebnis einer Abfrage von Hibernate geliefert werden, sind auch persistent. Am Ende einer Transaktion bzw. spätestens beim Schließen einer Session wird ihr Zustand mit der Repräsentation in der Datenbank abgeglichen.

Ein persistentes Objekt kann von der Hibernate-Session wieder getrennt werden. In diesem Fall spricht man von einem *detached* Objekt. Dabei können wir zwischen dem expliziten Entfernen über die Session-Methode `evict` oder dem impliziten Trennen am Ende (`close`) der Session unterscheiden. In beiden Fällen werden unsere Objekte nicht mehr von Hibernate überwacht, das heißt Änderungen an

den Objekten führen nicht mehr automatisch zu einer Anpassung in der Datenbank. Somit können wir diese Objekte ohne Bedenken an ein Frontend übergeben. Das besondere an detached Objekten ist die Tatsache, dass wir diese Objekte wieder mit einer anderen Hibernate-Session verbinden können. Dafür steht an der Hibernate-Session die `lock`-Methode zur Verfügung, wodurch ein *detached* Objekt wieder unter die Kontrolle von Hibernate gelangt (vgl. Abschnitt 9.3.2). Auch die Session-Methode `update` erzielt das gleiche Ergebnis, wobei hier das Aktualisieren des Objekts im Mittelpunkt steht (vgl. Abschnitt 7.8.3).