Seminararbeit zum Seminar Datenbanksysteme im SS 2008

Objekt-relationales Mapping mit Hibernate

Studiengang Informatik (Bachelor)
Fachbereich Mathematik-Naturwissenschaften-Informatik
Fachhochschule Gießen-Friedberg

Michael Eckel
3. Juni 2008

Betreuer: Prof. Dr. V. Klement

Inhaltsverzeichnis

1. Einführung

In vielen Java-Anwendungen ist es nötig, Daten (Objekte) dauerhaft zu speichern. In einer normalen Java-Anwendung werden Objekte aber nur vorübergehend im Hauptspeicher erzeugt und sind nur solange gültig wie die Anwendung läuft. Wird die Anwendung beendet, sind alle zuvor erzeugten Objekte nicht mehr vorhanden. In den meisten Anwendungen ist es aber erwünscht, dass Objekte dauerhaft gespeichert werden und somit nach dem Beenden und dem erneuten Starten der Anwendung noch zur Verfügung stehen. Für die dauerhafte Speicherung haben sich Datenbanksysteme bewährt, welche im Gegensatz zu anderen Speicherungsarten viele Vorteile bieten, wie z. B. gleichzeitigen Zugriff durch mehrere Anwendungen / Benutzer, Sicherstellung der Datenintegrität und Datenunabhängigkeit, um einige zu nennen.

Obwohl es objektorientierte Datenbanken schon lange gibt, findet man in der Praxis meist relationale Datenbanken, welche Daten in Tabellenform speichern. Relationale Datenbanken sind performanter als ihre objektorientierten Konkurrenten. Außerdem sind sie sehr robust, was damit zusammenhängt, dass sie auf der mathematischen Theorie der Relationen basieren. Will man nun Objekte dauerhaft in einer relationalen Datenbank speichern (persistent[1] machen), sieht man auf den ersten Blick, zumindest bei einfachen Objekten, eigentlich kein Problem darin. Man legt eine neue Tabelle an und für jedes Attribut legt man eine Tabellenspalte an. Doch wie sieht es bei ganzen Objekt-Netzwerken oder bei Vererbung aus? Relationale Datenbanken bieten meist keine Unterstützung für diese Probleme und man muss selbst Hand anlegen. Man kommt schnell darauf, dass sich die beiden Welten – die Objekt-Welt und die relationale Welt – nicht vertragen. Man nennt dieses Problem die „objekt-relationale Paradigmen-Unvereinbarkeit“. In der Praxis begegnet man dem Problem oft mit einer selbst-geschriebenen Persistenz-Schicht, um den Code für persistente Daten zentral zu halten. Eine solche Persistenz-Schicht beinhaltet im Schnitt 30% des Quellcodes vom gesamten Projekt. Ändert sich etwas an der Datenbank oder will man auf ein anderes Datenbanksystem migrieren, muss man den Code der Persistenz-Schicht anfassen und an das neue Datenbanksystem anpassen. Man sieht sofort, dass diese Technik nicht praktikabel ist und man händeringend nach einer besseren Lösung sucht. Es gibt mehrere Ansätze zwischen

[1] Die „Persistenz“ (aus lat. *persistere*: „verharren“) bezeichnet allgemein etwas mit dauerhafter Beschaffenheit oder Beharrlichkeit, das langfristige Fortbestehen einer Sache. Das zugehörige Adjektiv lautet „persistent“ [Wik08].

der Objekt-Welt und der relationalen Welt eine Brücke zu schlagen. Die momentan beste Lösung dafür ist objekt-relationales Mapping (kurz *O-R/M* oder einfach *ORM*). Und genau dieser Aufgabe widmet sich Hibernate – dem objekt-relationalen Mapping. Hibernate ist nicht das einzige Tool für ORM, doch Hibernate ist Open-Source. Grund genug, das Ganze genauer zu betrachten.

In den folgenden Kapiteln werde ich deshalb näher auf Aspekte rund um Hibernate eingehen. In Kapitel 2 werde ich zunächst einen genaueren Blick in die Theorie – Überlegungen, Grundlagen – von ORM werfen, wobei die zu bewältigenden Problemstellungen beim ORM betrachtet werden. Im folgenden Kapitel werde ich dann auf die Grundlagen von Hibernate eingehen, wobei u.a. Informationen und Tipps gegeben werden über den Aufbau eines Projektes mit Hibernate. Auch verschiedene Entwicklungsszenarien werden besprochen. In Kapitel 4 folgt mit einem Beispiel-Projekt die Praxis, also der Einsatz von Hibernate in einem Java-Projekt. Dort werden die durchzuführenden Schritte bei der Entwicklung eines Projektes mit Hibernate erklärt. In Kapitel 5 werde ich einen Ausblick geben auf weitere Features von Hibernate. Das letzte Kapitel ist eine Zusammenfassung und gibt nochmal die in dieser Seminararbeit diskutierten Aspekte wieder.

2. Theoretische Grundlagen von ORM

In diesem Kapitel geht es um die grundsätzlichen Dinge, mit denen sich ein ORM-Tool auseinandersetzt. Dabei gehe ich in Abschnitt 2.1 noch etwas genauer auf verschiedene Möglichkeiten der persistenten Speicherung ein. In Abschnitt 2.2 gehe ich dann auf die diversen Probleme ein, die ein ORM-Tool wie Hibernate lösen muss.

2.1. Persistenz-Mechanismen

In der Einführung in Kapitel 1 wurde bereits erwähnt, was Persistenz ist. Es handelt sich um das dauerhafte Speichern von Daten und speziell in Java um das dauerhafte Speichern von Objekten. Auch wurden in der Einführung zwei Persistenz-Mechanismen erwähnt, und zwar das Speichern der Daten in einer objektorientierten Datenbank und das Speichern der Daten in einer relationalen Datenbank. Diese beiden sind nicht die einzigen Persistenz-Mechanismen. Es gibt noch weitere, welche im Folgenden diskutiert werden.

2.1.1. Serialisierung

Serialisierung ist ein bereits in Java eingebauter Persistenz-Mechanismus. Um ein Objekt zu serialisieren, muss die entsprechende Klasse das Interface `Serializable` implementieren. Mit der Methode `serialize()` des Objekts kann man das Objekt dann serialisieren. Auch ein ganzes Netzwerk von Objekten kann man so serialisieren. Man hat dann sozusagen einen Schnappschuss des momentanen Zustands der Anwendung. Doch der große Nachteil dieser Art der persistenten Speicherung ist, dass man in dem durch die Serialisierung entstandenen Datenstrom nicht selektiv auf Teile dieses Datenstroms (z. B. ein bestimmtes Objekt) zugreifen kann, ohne den gesamten Datenstrom zu deserialisieren. Das Ganze wird dann sehr unperformant für Datenbanksysteme, welche ein hohes Maß an gleichzeitigem Zugriff gewährleisten sollen.

2.1.2. XML

Die Daten in hierarchischer Form in XML zu speichern bietet den Vorteil, dass man auf Teile, u. a. die einzelnen Objekte, selektiv zugreifen kann, ohne die ganze Datei komplett laden zu müssen. Allerdings ist dies auch keine gute Lösung, da XML z. B. keine Vererbung unterstützt. Die Daten können nur hierarchisch gespeichert werden und man stößt beim persistenten Speichern mit XML schließlich auf das Problem der „objekt-hierarchischen Paradigmen-Unvereinbarkeit“.

2.2. Probleme beim Objekt-relationalen Mapping

Wie in der Einführung bereits erwähnt, ist ORM die brauchbarste Lösung, um Daten persistent in einer relationalen Datenbank zu speichern. Aufgrund der objekt-relationalen Paradigmen-Unvereinbarkeit muss ein ORM-Tool einige Probleme diesbezüglich lösen.

2.2.1. Das Problem der Vererbung

In vielen Domain-Modellen wird vom objektorientierten Feature der Vererbung Gebrauch gemacht. Da die meisten relationalen Datenbanksysteme keine Vererbung unterstützen, und wenn doch, diese Unterstützung nicht einheitlich ist, betrachten wir uns im Folgenden eine Lösung für dieses Problem. Als Beispiel nehmen wir ein einfaches Klassendiagramm (siehe Abbildung 2.1). Als Superklasse dient hier die Klasse `Mitarbeiter` mit den Attributen `persNr` (Personalnummer) und `name`. Als Subklassen haben wir die Klasse `Manager` mit dem zusätzlichen Attribut `abtNr` (Abteilungsnummer) und die Klasse `Ingenieur` mit dem zusätzlichen Attribut `gebiet`.

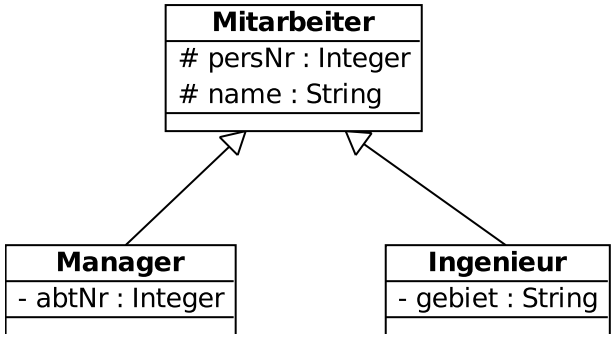

Abbildung 2.1.: Klassendiagramm – Klasse Mitarbeiter

Um diese Vererbungshierachie in einer relationalen Datenbank zu realisieren kann man folgenden Ansatz gehen: Eine gemeinsame Tabelle für alle Klassen plus zugehörige Views für die Superklasse und jede Subklasse. In SQL formuliert sieht das Ganze dann so aus wie in Listing 2.1 (als SQL-Dialekt und Datenbanksystem verwende ich hier PostgreSQL).

Listing 2.1: SQL-Code – Vererbung

```
1  create table M (
2      persNr integer primary key,
3      typ char(1) not null,
4      name varchar(40),
5      abtNr integer,
6      gebiet varchar(30)
7  );
8
9  create view Mitarbeiter as
10     select persNr, name
11     from M;
12
13 create view Manager as
14     select persNr, typ, name, abtNr
15     from M
16     where Typ = 'M';
17
18 create view Ingenieur as
19     select persNr, typ, name, gebiet
20     from M
21     where Typ = 'I';
```

Diese ist nicht die einzige Möglichkeit, wie man Vererbung in einer relationalen Datenbank realisieren kann. Hier soll lediglich anhand eines praktischen Beispiels gezeigt werden, dass und wie dies möglich ist. Neben dem Problem der Vererbung gibt es eine Menge weiterer Probleme, die ein ORM-Tool lösen muss. Wir schauen uns deshalb als nächstes das Problem der Identität an.

2.2.2. Das Problem der Identität

In Java gibt es zwei Arten von Identität/Gleichheit. Zum einen ist dies die Identität von Objekten, welche durch die Hauptspeicher-Adresse (Referenz) gegeben ist und mit dem ==-Operator geprüft werden kann. Zum anderen ist dies die Gleichheit, welche durch die Implementierung der Methode `equals()` festgelegt wird. Man nennt diese Gleichheit auch Zustandsgleichheit.

In relationalen Datenbanksystemen ist die Identität eines Datensatzes (Tabellenzeile) durch den Primärschlüssel festgelegt. Weder `equals()` noch == ist natürlich

äquivalent zum Wert des Primärschlüssels. Ein weiteres Problem ist ist mit der Wahl des Primärschlüssels verbunden. Wir betrachten ein Beispiel (siehe Listing 2.2). Wir sehen hier eine sehr abgespeckte Version eines Online-Diskussions-Forums (im Folgenden einfach Online-Forum genannt). Das Beispiel beinhaltet nur zwei Tabellen, was für ein echtes Online-Forum wahrscheinlich zu wenige sind. Aber es soll hier nur zur Demonstration eines bestimmten Sachverhalts dienen.

Listing 2.2: SQL-Code – Benutzer- und Beitrag-Tabelle für Online-Forum (1)

```
1  create table Benutzer (
2      loginname varchar(12) primary key,
3      vorname varchar(30),
4      nachname varchar(30),
5      email varchar(60)
6  );
7
8  create table Beitrag (
9      betreff varchar(20),
10     text varchar(100),
11     loginname varchar(12) references Benutzer (loginname)
12 );
```

Wir sehen hier die Tabelle `Benutzer`, welche einen Benutzer in einem Online-Forum repräsentieren soll. Des weiteren sehen wir die Tabelle `Beitrag` mit einer Fremdschlüssel-Beziehung auf die Tabelle `Benutzer`. Die Tabelle `Beitrag` speichert ins Forum geschriebene Beiträge der Benutzer. Die Wahl des Primärschlüssels in der Tabelle `Benutzer` ist auf den Login-Namen als natürlicher Schlüssel gefallen, da dieser in jedem Fall eindeutig sein muss. Das Problem tritt jetzt auf, wenn man den Login-Namen ändern möchte. Man muss dazu nicht nur das Feld `loginname` des Datensatzes in `Benutzer` ändern, sondern auch die Fremdschlüsselspalte in jedem betroffenen Datensatz in `Beitrag`. Dieses Problem tritt bei sehr vielen natürlichen Schlüsseln auf, sodass zur Lösung des Problems ein *Surrogatschlüssel* empfohlen wird. Ein Surrogatschlüssel (künstlicher Schlüssel, Ersatzschlüssel oder auch Stellvertreterschlüssel) dient als Ersatz, wenn man keinen geeigneten natürlichen Schlüssel finden kann. An dieser Stelle teilen sich die Meinungen der Experten. Die einen sagen, dass man nur Surrogatschlüssel einführen sollte, wenn man keinen geeigneten natürlichen Schlüssel finden kann, die anderen raten dazu, immer einen Surrogatschlüssel zu verwenden. Ein solcher Surrogatschlüssel hat für den Anwender keine Bedeutung, er dient lediglich der Identifikation von Daten innerhalb des Softwaresystems. Ein Surrogatschlüssel sollte einen Ganzzahl-Typ sein mit großem Bereich, in PostgreSQL z. B. `bigint`. Wir ändern die Tabellen nun so ab, dass sie je einen Surrogatschlüssel bekommen. Auch die Tabelle `Beitrag`, welche noch keinen Primärschlüssel hat, bekommt einen künstlichen Primärschlüssel. Wir schauen uns nun die veränderten Tabellen in Listing 2.3 an.

Listing 2.3: SQL-Code – Benutzer- und Beitrag-Tabelle für Online-Forum (2)

```
1  create table Benutzer (
2      benutzer_id bigint not null primary key,
3      loginname varchar(12) unique,
4      vorname varchar(30),
5      nachname varchar(30),
6      email varchar(60)
7  );
8
9  create table Beitrag (
10     beitrag_id bigint not null primary key,
11     betreff varchar(20),
12     text varchar(100),
13     benutzer_id bigint references Benutzer (benutzer_id)
14 );
```

Die Spalten `benutzer_id` und `beitrag_id` wurden nur zugunsten des Datenmodells eingeführt und enthalten vom System generierte eindeutige Werte. Normalerweise sollten diese Spalten im Domain-Modell nicht auftauchen, da es eine technische Sache ist und nichts mit der Geschäftslogik zu tun hat. Aber die Primärschlüssel-Eigenschaft (auch Identifikator-Eigenschaft) ist ein Sonderfall und man sollte sie generell mit in die Klassen aufnehmen. Eine weitere Frage ist auch, ob die Getter- und Setter-Methode einer solchen Identifikator-Eigenschaft `private` oder `public` gemacht werden sollte. Eine gute Lösung dafür ist, die Getter-Methode `public` zu machen und die Setter-Methode `private`. In Web-Anwendungen wird die Identifikator-Eigenschaft nämlich oft benutzt, z. B. bei einer Auflistung von Daten aus der Datenbank für den Client mit der Möglichkeit auf eine Zeile dieser Liste mit der Maus zu klicken und diese somit zu bearbeiten oder ähnliches. In den anklickbaren Verweis (URL) schreibt man als Parameter den Wert des Primärschlüssels vom Datensatz. So kann man auf der Server-Seite den Datensatz identifizieren.

Im nächsten Abschnitt betrachten wir uns weitere Probleme, mit denen sich ein ORM-Tool beschäftigen muss – die mit Assoziationen verbundenen Probleme.

2.2.3. Mit Assoziationen verbundene Probleme

„Das Mapping von Assoziationen und die Verwaltung von Entity-Assoziationen sind die zentralen Konzepte in jeder Lösung für eine Objektpersistenz.“ [BK07]. In Java und allen anderen objektorientierten Sprachen werden Assoziationen über Objektreferenzen dargestellt. In einer relationalen Datenbank werden dafür Kopien von Schlüsseln anderer Tabellen (Fremdschlüssel) verwendet, um eine Verknüpfung herzustellen. Objektreferenzen sind Zeiger und deshalb direktional – von einem Objekt zum anderen. Will man Objekte so assoziieren, dass sie in beide Richtungen

navigierbar sind, muss man in jedem Objekt eine Referenz auf das jeweils andere Objekt anlegen, ähnlich folgendem Java-Code in Listing 2.4.

Listing 2.4: Java-Code – Bidirektionale Assoziation

```
1 public class Person {
2     private Set adressen;
3     ...
4 }
5
6 public class Adresse {
7     private Person person;
8     ...
9 }
```

Fremdschlüsselassoziationen dagegen sind nicht direktional. Navigation hat im relationalen Datenmodell keine Bedeutung, man kann mit Hilfe von Tabellen-Joins beliebige Datenassoziationen erstellen. „Die Herausforderung besteht darin, den Brückenschlag zwischen einem vollkommen offenen Datenmodell, das von der Applikation unabhängig ist, die mit den Daten arbeitet, und einem applikationsabhängigen Navigationsmodell zu bewerkstelligen, einer eingeschränkten Sicht der Assoziationen, die von dieser speziellen Applikation benötigt werden.“ [BK07]. Zwei Java-Klassen können z. B. auf einfache Weise eine *many-to-many*-Assoziation bilden (siehe Listing 2.5).

Listing 2.5: Java-Code – *many-to-many*-Assoziation

```
1 public class Buch {
2     private Set autoren;
3     ...
4 }
5
6 public class Autor {
7     private Set buecher;
8     ...
9 }
```

Tabellenassoziationen können niemals direkt eine *many-to-many*-Assoziation aufweisen, dafür wird eine Zwischentablle benötigt. Tabellenassoziationen können nur *one-to-many* (bzw. *many-to-one*) oder *one-to-one* sein.

2.2.4. Das Problem der Datennavigation

In Java greift man auf Objekte zu, indem man von Objekt zu Objekt navigiert, etwa so: `telefonbuch.getPerson("Hans Meier").getTelNr()` oder ähnlich. Leider ist diese Art, Daten aus einer relationalen Datenbank zu holen, nicht sehr

effizient. Um die Performance zu verbessern, ist die beste Möglichkeit, die Anzahl der Anfragen an die Datenbank zu minimieren, was in der Praxis bedeutet, die Anzahl der SQL-Queries zu verringern. Eine effiziente Weise, Daten aus mehreren Tabellen der Datenbank gleichzeitig zu holen, geht über Tabellen-Joins. Es müssen alle Tabellen eingeschlossen werden, über die in der Java-Anweisung in Form von Objekten navigiert wird. Das setzt voraus, dass man vorher wissen muss, auf welche Teile man zugreift und dann nur die betroffenen Teile aus der Datenbank holt. Holt man stattdessen bei jedem „Weiterhangeln“ das entsprechend nächste Objekt und führt dazu jedesmal eine Datenbankabfrage durch, ist dies sehr ineffizient und führt zum *n+1 selects*-Problem. „Diese unterschiedliche Art, auf Objekte in Java und in einer relationalen Datenbak zuzugreifen, ist vielleicht die am häufigsten auftretende Quelle von Performanceproblemen in Java-Applikationen.“ [BK07].

3. Grundlagen von Hibernate

In diesem Kapitel beschäftigen wir uns mit den grundlegenden Dingen rund um Hibernate, bevor wir uns in Kapitel 4 angucken wie man ein Hibernate-Projekt durchführt.

Wie inzwischen bekannt sein dürfte, ist Hibernate[1] ein Tool, welches das objekt-relationale Mapping bewerkstelligt. Doch wie genau und wo kann man sich Hibernate vorstellen in Bezug auf die Java-Applikation und das Datenbanksystem? Abbildung 3.1 zeigt ein Schalenmodell. Der Kern dieses Schalen-Modells ist die Datenbank. Dort werden Daten persistent gespeichert. Damit Hibernate (oder auch Java direkt) mit der Datenbank sprechen kann, wird ein JDBC-Treiber für die Datenbank benötigt. In der äußeren Schale befindet sich die *Java Persistence API*, kurz: *JPA*. Diese bietet u.a. standardisierte Schnittstellen in Form von Annotationen, um mit einem ORM-Tool zu kommunizieren, welches nicht unbedingt Hibernate sein muss. Ich habe die JPA-Schale hier nur der Vollständigkeit halber aufgeführt. In dieser Seminararbeit verwende ich Hibernate immer direkt ohne JPA-Annotationen.

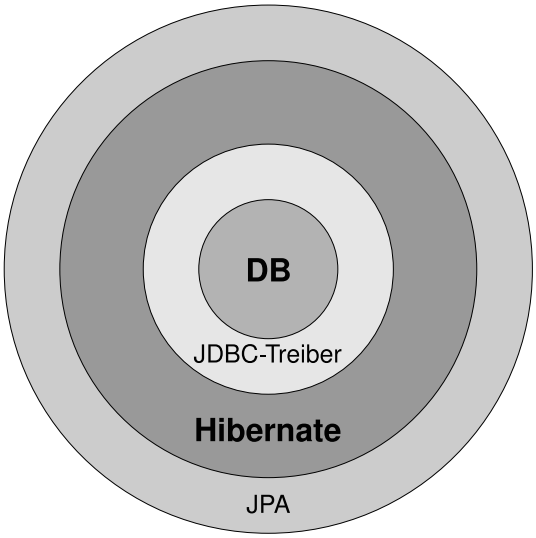

Abbildung 3.1.: Schalenmodell Hibernate

[1] Übrigens: „Hibernate“ heißt wörtlich aus dem Englischen ins Deutsche übersetzt „Winterschlaf halten“

3.1. Das Persistenz-Schicht-Modell

„In einer mittleren oder großen Applikation ist es normalerweise sinnvoll, Klassen nach Funktionbereich zu organisieren.“ [BK07]. Wie dies aussehen kann zeigt Abbildung 3.2. Hier im Diagramm ganz oben hat man die Präsentationsschicht, welche für die Darstellung von Daten für den Benutzer zuständig ist. Dann die Business-Schicht, in der die Geschäftslogik steckt und zu guter letzt die Persistenzschicht, in der die Klassen, die sich mit der Persistenz beschäftigen, zentral untergebracht sind. Wichtig sind die Abhängigkeiten der Schichten. Die Präsentationsschicht darf nur auf die Business-Schicht zugreifen, die Business-Schicht nur auf die Persistenzschicht und ganz wichtig: nur die Persistenzschicht darf auf die Datenbank zugreifen! Außerdem gibt es noch einen Satz Hilfsklassen, welche in jeder Schicht benutzt werden können und sich nicht an die Abhängigkeiten halten.

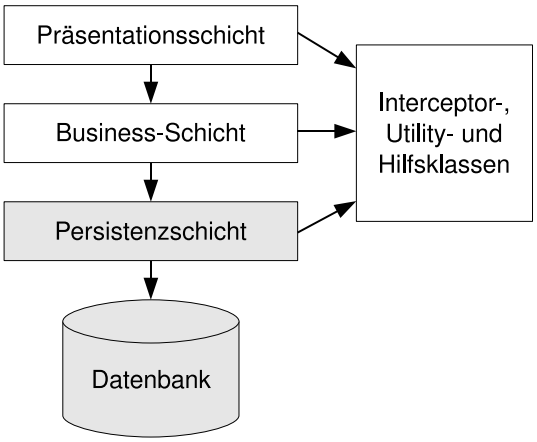

Abbildung 3.2.: Schichten-Diagramm einer mittleren oder großen Applikation

3.2. Entwicklungsszenarien

Die Art und Weise wie man an ein Projekt rangehen kann, ist nicht immer gleich. Mal hat man ein vorhandenes Datenbankschema, mal ein vorhandenes Domain-Modell; manchmal auch beides oder garnichts. Deshalb betrachten wir im Folgenden verschiedene Entwicklungsszenarien/-prozesse.

Top down: Beim Top-down-Entwicklungsprozess hat man bereits ein Domain-Modell und die Implementierung dessen in Java. Ein Datenbankschema ist noch nicht vorhanden. Deshalb ist der Top-down-Entwicklungsprozess für Java-Entwickler der komfortabelste von allen. Es müssen anschließend Mapping-Metadaten erstellt werden, anhand denen man sich mit dem Hibernate-Tool `hbm2ddl` dann ein Datenbankschema erstellen lassen kann.

Bottom up: Im Gegensatz zum Top-down-Entwicklungsprozess hat man beim Bottom-up-Entwicklungsprozess ein vorhandenes Datenbankschema und noch kein Domain-Modell. Mit dem Hibernate-Tool `hbm2hbmxml` kann man sich aus dem vorhandenen Datenbankschema dann XML-Mapping-Dateien erzeugen lassen. Aus diesen Metadaten kann man sich mit dem Tool `hbm2java` dann Java-Persistenz-Klassen generieren lassen.

Middle out: Dieser Entwicklungsprozess wird von manchen Entwicklern bevorzugt, da man die XML-Mapping-Metadaten per Hand schreibt und sich daraus mit den oben genannten Tools sowohl das Datenbankschema als auch die Java-Persistenz-Klassen generieren lässt. Dieser Entwicklungsstil kann allerdings nur erfahrenen Hibernate-Experten empfohlen werden.

Meet in the middle: Dieses Entwicklungsszenario ist das schwierigste von allen. Es ist die Kombination aus vorhandenem Datenbankschema und vorhandenen Java-Klassen. Die Hibernate-Tools können hier nicht viel helfen. Man kommt (sehr wahrscheinlich) um eine Überarbeitung der Java-Klassen und/oder des Datenbankschemas nicht herum. Dieses Szenario kommt zum Glück sehr selten vor.

3.3. Verbindungspooling und Hibernate-Schnittstellen

Es ist nicht zu empfehlen, für jede Anfrage einer Java-Applikation an die Datenbank eine Verbindung aufzubauen und danach wieder zu beenden. Dies zehrt sehr an der Performance des Datenbanksystems. Stattdessen ist es für eine Java-Applikation sinnvoll, einen Pool von Verbindungen zu verwalten. Bei jeder Anfrage an die Datenbank bekommt die Java-Applikation eine Verbindung aus dem Verbindungspool, welcher offene Verbindungen zur Datenbank zur Verfügung hält. Ist die Anfrage beendet, wird die Verbindung zur Datenbank nicht geschlossen, sondern an der Verbindungspool zurückgegeben. Die Verbindungen im Verbindungspool werden beim Start des Programms oder bei der ersten Verwendung aufgebaut. Ein bewährtes Open-Source-Tool für das Verbindungspooling ist die Software C3P0. Verbindungspooling kann man direkt so – ohne Hibernate – einsetzen (siehe Abbildung 3.3).

Mit Hibernate ändert sich das Bild etwas. Man verwendet jetzt nicht mehr direkt den Verbindungspool, sondern die Hibernate-APIs `Session`, `Transaction` und `Query`, wie in Abbildung 3.4 dargestellt. Um das Verbindungspooling kümmert sich jetzt Hibernate. Im Folgenden werden kurz die Aufgaben der Hibernate-APIs beschrieben:

Session: Eine Hibernate-`Session` repräsentiert eine bestimmte Arbeitseinheit mit der Datenbank. Es ist ein nicht threadsicheres Objekt, das deshalb nur von ei-

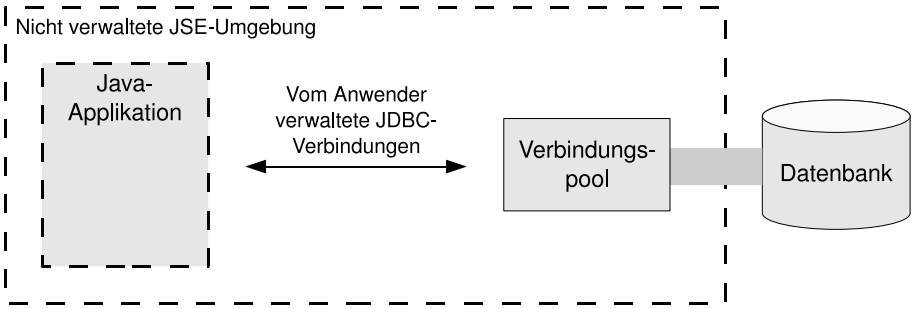

Abbildung 3.3.: Verbindungspooling ohne Hibernate

nem Thread benutzt werden sollte. Eine Session besitzt außerdem eine Queue für SQL-Anweisungen und eine Map mit allen von der `Session` überwachten Persistenz-Instanzen.

Transaction: Dient zum Setzen von Transaktionsgrenzen in der Appikation. Diese Transaktionsgrenzen können allerdings auch über andere Schnittstellen wie das JTA-Interface gesetzt werden.

Query: Mit einer Query, kann man Anfragen an die Datenbank senden, entweder in der Sprache von Hibernate HQL[2] oder in normalem SQL. Außerdem unterstützen Quries Platzhalter für Argumente.

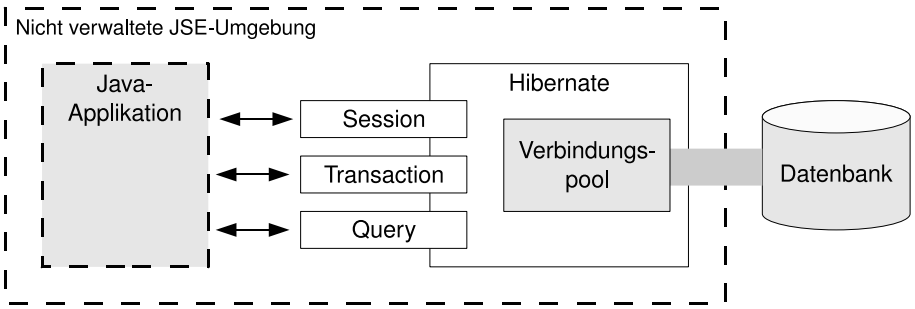

Abbildung 3.4.: Verbindungspooling mit Hibernate

3.4. Der Persistenz-Kontext

Alle Objekte in Java sind normalerweise transient, was das Gegenteil von persistent ist. Das heißt, sie sind nach dem Beenden der Applikation nicht mehr vorhanden. Mit der Benutzung von Hibernate ändert sich dieses Modell. Es gibt nun weitere Zustände, in denen ein Objekt sein kann. Abbildung 3.5 stellt diese Zustände dar.

[2]HQL heißt *Hibernate Query Language* und ähnelt SQL. Der wesentliche Unterschied ist aber, dass HQL sich auf Objekte bezieht, während SQL auf Tabellen ausgerichtet ist.

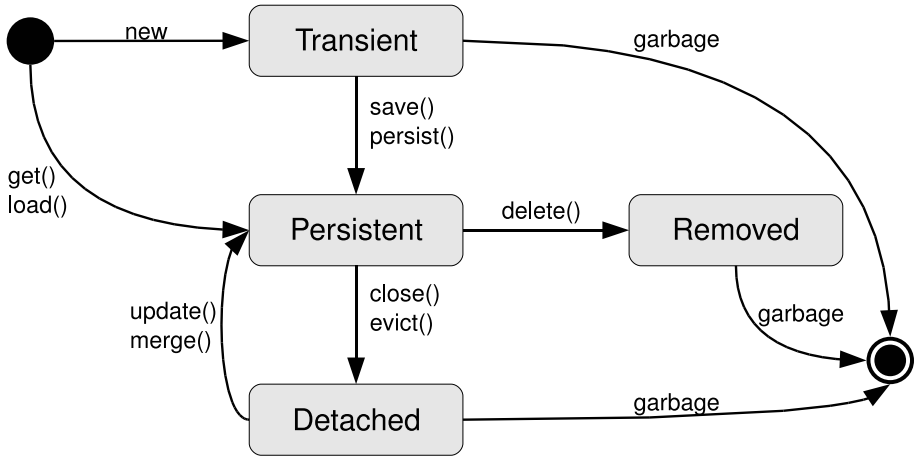

Abbildung 3.5.: Der Persistenz-Kontext

Es gibt nun die Zustände *Transient*, *Persistent*, *Detached* und *Removed*. Abbildung 3.5 zeigt auch die verschiedenen Methoden des `Session`-Objektes, mit denen Zustandsübergänge veranlasst werden. Wenn ich in diesem Abschnitt von Methoden rede, sind die Methoden des `Session`-Objektes gemeint. Erzeugt man in einer Java-Appikation ein neues Objekt (`new ...()`), ist es zunächst transient. Mit der Methode `save()` oder `persist()` des `Session`-Objektes kann das transiente Objekt in den Zustand *Persistent* übergehen, oder anders gesagt: es wird in den Persistenz-Kontext aufgenommen. Objekte, die in den Persistenz-Kontext aufgenommen werden können, müssen Hibernate bekannt sein und es muss ein Mapping für sie existieren (siehe Kapitel 4 Abschnitt 4.3.2). Ist ein Objekt im Persistenz-Kontext, wacht Hibernate über es und alle Änderungen am Objekt, welche in Java-üblicher Weise geschehen (also mit Gettern und Settern), werden persistent gemacht. Das heißt nicht, dass Hibernate diese Änderungen sofort in die Datenbank schreibt. Die Änderungen können aus Performanz-Gründen auch zuerst in einem Hibernate-internen Cache zwischengespeichert werden und zu gegebenem Zeitpunkt z. B. „in einem Rutsch“ in die Datenbank geschrieben werden. Ist ein Objekt im Zustand *Persistent*, kann man es z. B. mit der Methode `evict()` *Detached* („losgelöst“) machen. Dieses Loslösen wird auch automatisch veranlasst, wenn die Session mit der Methode `close()` geschlossen wird. Hibernate wacht nun nicht mehr über dieses Objekt und Änderungen werden nicht mehr persistent gemacht. Das Objekt hat zwar noch eine Entsprechung in der Datenbank, sein Zustand kann jedoch veraltet sein. Ändert man das *Detached*-Objekt und ruft die Methode `update()` auf, kann man das Objekt so wieder in den Persistenz-Kontext befördern. Dabei werden die (geänderten) Werte aus dem Objekt in der Datenbank persistent gemacht (also der entsprechende Datensatz aktualisiert). Will man ein Objekt aus dem Persistenz-Kontext löschen und den entsprechenden Datensatz aus der Datenbank löschen, nimmt man die Methode `delete()`. Die Pfeile im Diagramm, die mit „garbage“ beschriftet sind, bedeuten, dass das Objekt jederzeit

vom Garbage Collector zerstört werden kann – natürlich nur dann, wenn keine Referenz mehr darauf existiert. Man kann auch direkt ein persistentes Objekt bzw. eine ganze Collection von persistenten Objekten aus der Datenbank holen. Dazu dienen die Methoden `get()` und `load()` des `Session`-Objektes. Außerdem kann man dafür die Query-Schnittstelle verwenden.

Trotzdem Hibernate dem Entwickler viel Arbeit abnimmt, muss der Entwickler, um eine performante Applikation zu entwickeln, sich auskennen mit:

- der Applikation
- Datenanforderungen
- SQL
- rel. Speicherstrukturen

Dies ist notwendig, damit der Entwickler die Applikation in Bezug auf Performanz optimieren kann.

Im nächsten Abschnitt wird beschrieben, wie man ein Hibernate-Projekt durchführt. Dabei wird auf die einzelnen Schritte, die dabei zu machen sind, genauer eingegangen.

4. Ein Hibernate-Projekt durchführen

Im Folgenden werden die Schritte erklärt, die notwendig sind, um ein Hibernate-Projekt durchzuführen. Als Entwicklungsumgebung wird hier „Eclipse 3.3.2 Europa“ (im Folgenden einfach nur „Eclipse“ genannt) verwendet. Als Datenbanksystem kommt PostgreSQL in der Version 8.3 zum Einsatz. Das Entwicklungsszenario ist hier *Top down* mit dem in Abschnitt 4.1 beschriebenen Domain-Modell.

4.1. Das Domain-Modell

Bevor wir das Projekt beginnen, betrachten wir uns das zugrunde liegende Domain-Modell einer Dokumentenverwaltung, in Abbildung 4.1 dargestellt als UML-Klassen-Modell.

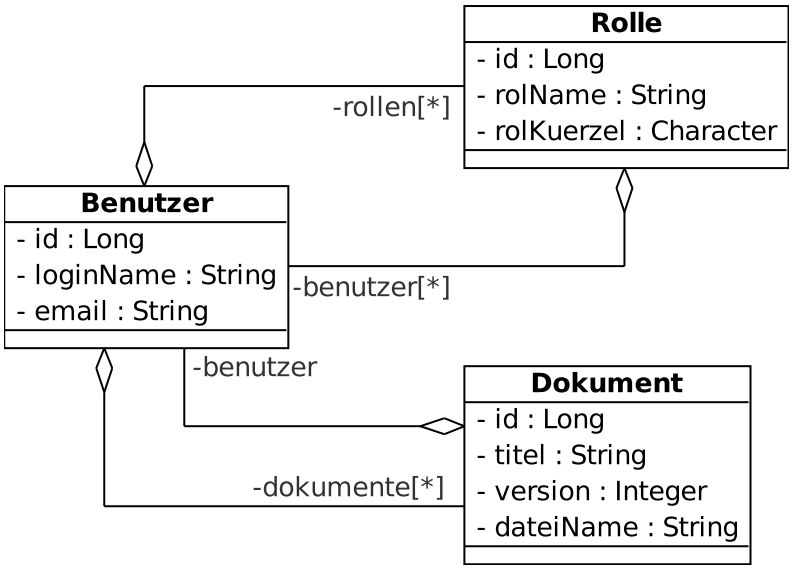

Abbildung 4.1.: Klassendiagramm Dokumentenverwaltung

Wir sehen hier die Klassen `Benutzer`, `Rolle` und `Dokument`. Jede Klasse hat Attribute mit entsprechenden Getter- und Setter-Methoden und einen Default-

Konstruktor im Stil von Java Beans. Der Default-Konstruktor ist nötig, da Hibernate die Klassen per Reflektion instanziiert.

Folgende Beziehungen gelten, welche auch für das Datenbankschema wichtig sind:

- Ein Benutzer kann mehrere Rollen haben
- Eine Rolle können mehrere Benutzer haben (bzw. mehrere Benutzer können die gleiche Rolle haben)
- Ein Benutzer kann mehrere Dokumente haben
- Ein Dokument gehört genau einem Benutzer

Ansonsten sollte das UML-Diagramm selbsterklärend sein. Der vollständige Quellcode der zugehörigen Java-Klassen ist in Anhang A zu finden.

4.2. Arbeitsumgebung einrichten

In diesem Abschnitt wird die Einrichtung der Arbeitsumgebung beschrieben, damit die angegeben Quellcodes auch genau so lauffähig sind.

4.2.1. Datenbank einrichten

Wir laden auf der offiziellen Homepage PostgreSQL in der Version 8.3 herunter und installieren es. Wir richten, wenn nötig, den Beutzer „postgres“ mit de Passwort „postgres“ ein. Dann starten wir das Tool „pgAdmin III“ und verbinden uns zur Datenbank auf `localhost` mit dem eben angegebenen Benutzernamen und dem Passwort. Wir legen die neue Datenbank „dokuverw“ (Dokumentenverwaltung) an und beenden das Tool. Die nun gemachten Einstellungen sind ausreichend. Die Tabellen werden wir später mit dem Hibernate-Tool `hbm2ddl` erzeugen lassen.

4.2.2. Eclipse einrichten

In Eclipse legen wir ein neues Java-Projekt an. Als Name nehmen wir z. B. „Doku-Verwaltung“. Um Hibernate im Projekt verwenden zu können, müssen noch zusätzliche Bibliotheken eingebunden werden. Diese kann man im Download-Bereich von `http://www.hibernate.org` herunterladen. Wir verwenden nur das Paket „Hibernate Core“ in der Version 3.2.6, da dies zum Zeitpunkt der Erstellung dieser Seminararbeit die aktuellste Version war. Nach dem Herunterladen, entpacken wir die Dateien, erstellen im Eclipse-Projekt einen neuen Ordner names `lib` (im Wurzelverzeichnis des Projekts) und fügen folgende JAR-Dateien dort ein:

- `antlr-2.7.6.jar`

- asm.jar
- asm-attrs.jars
- c3p0-0.9.1.jar
- cglib-2.1.3.jar
- commons-collections-2.1.1.jar
- commons-logging-1.0.4.jar
- dom4j-1.6.1.jar
- hibernate3.jar
- jta.jar
- log4j-1.2.11.jar

Außerdem muss noch der JDBC-Treiber für PostgreSQL (postgresql-8.3-603.jdbc4.jar) ins lib-Verzeichnis gepackt werden (http://jdbc.postgresql.org/download.html). Danach müssen all diese JAR-Dateien dem *Java Build Path* hinzugefügt werden.

4.3. Das Projekt beginnen

Nachdem nun die Vorbereitungen getroffen wurden und die Arbeitsumgebung eingerichtet ist, kann mit dem eigentlichen Programmieren begonnen werden.

Wir legen ein neues Package namens de.fhgiessen.dbs an. Dort legen wir die Klassen Benutzer, Rolle und Dokument entsprechend den Listings 4.1, 4.2 und 4.3 an. Die vollständigen Quelltexte der Klassen befinden sich in Anhang A. Außerdem legen wir noch eine Klasse MainKlasse an, die die main-Methode enthält.

Listing 4.1: Java-Code – Klasse Benutzer

```
1  public class Benutzer {
2      private Long id;
3      private String loginName;
4      private String email;
5      private Set rollen = new HashSet();
6      private Set dokumente = new HashSet();
7
8      public Benutzer() {
9      }
10
11     /*
12      * Getter- und Setter und ggf. weitere Konstruktoren
```

```
13      */
14
15 }
```

Listing 4.2: Java-Code – Klasse `Rolle`

```
1  public class Rolle {
2      private Long id;
3      private String rolName;
4      private Character rolKuerzel;
5      private Set benutzer = new HashSet();
6
7      public Rolle() {
8      }
9
10     /*
11      * Getter- und Setter und ggf. weitere Konstruktoren
12      */
13
14 }
```

Listing 4.3: Java-Code – Klasse `Dokument`

```
1  public class Dokument {
2      private Long id;
3      private String titel;
4      private Integer version;
5      private String dateiName;
6      private Benutzer benutzer;
7
8      public Dokument() {
9      }
10
11     /*
12      * Getter- und Setter und ggf. weitere Konstruktoren
13      */
14
15 }
```

Wir sehen in jeder Klasse das Attribut `id`, welches den Primärschlüssel in der Datenbank wiederspiegelt. Außerdem sehen wir Attribute mit Standard-Typen aus `lava.lang` und Attribute vom Typ `Set`, welche die Relationen aus Abbildung 4.1 wiederspiegeln. Für all diese Klassen legen wir in Abschnitt 4.3.2 Mapping-Dateien an, damit Hibernate weiß, wie die Objekte der Klassen in die relationale Datenbank überführt werden können und sollen. Doch zunächst machen wir im folgenden Abschnitt die Grundkonfiguration von Hibernate.

4.3.1. Grundkonfiguration von Hibernate

Man kann Hibernate über eine `properties`-Datei konfigurieren, doch wir werden, wie die meisten Hibernate-Entwickler, die Datei `hibernate.cfg.xml` dazu verwenden. Diese Datei muss Wurzelverzeichnis von `src` des Projektes liegen (sie darf nicht in einem Unter-Package liegen). In Listing 4.4 sehen wir den Inhalt der Datei.

Listing 4.4: Die Datei `hibernate.cfg.xml`

```
<!DOCTYPE hibernate-configuration SYSTEM "http://hibernate.
    sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.connection.driver_class">
            org.postgresql.Driver
        </property>
        <property name="hibernate.connection.url">
            jdbc:postgresql://localhost/dokuverw
        </property>
        <property name="hibernate.connection.username">
            postgres
        </property>
        <property name="hibernate.connection.password">
            postgres
        </property>
        <property name="hibernate.dialect">
            org.hibernate.dialect.PostgreSQLDialect
        </property>

        <!-- Tabellen automatisch erzeugen lassen -->
        <property name="hibernate.hbm2ddl.auto">update</property>

        <!-- Verwenden des C3P0 Connection Pool Providers -->
        <property name="hibernate.c3p0.min_size">5</property>
        <property name="hibernate.c3p0.max_size">20</property>
        <property name="hibernate.c3p0.timeout">300</property>
        <property name="hibernate.c3p0.max_statements">50</property
            >
        <property name="hibernate.c3p0.idle_test_period">3000</
            property>

        <!-- Schoenes, sauberes SQL bei stdout ausgeben -->
        <property name="show_sql">true</property>
        <property name="format_sql">true</property>

        <!-- Liste der XML-Mapping-Dateien -->
```

```
35          <mapping resource="de/fhgiessen/dbs/Benutzer.hbm.xml" />
36          <mapping resource="de/fhgiessen/dbs/Rolle.hbm.xml" />
37          <mapping resource="de/fhgiessen/dbs/Dokument.hbm.xml" />
38      </session-factory>
39  </hibernate-configuration>
```

Die gesetzten Einstellungen sind weitestgehend selbsterklärend, dennoch will ich sie hier kurz beschreiben. In den Zeilen 4 bis 18 teilen wir Hibernate mit, mit welchem Datenbanksystem wir es zu tun haben und wie eine Verbindung zur Datenbank aufgebaut werden kann. An dieser Stelle ist es wichtig zu erwähnen, dass Hibernate mit vielen Datenbanksystemen im entsprechenden SQL-Dialekt sprechen kann. Der von Hibernate intern generierte SQL-Code wird somit für verschiedene Datenbanksysteme entsprechend unterschiedlich aussehen. Damit ist ein Wechsel des Datenbanksystems sehr leicht möglich. In einer selbstgeschriebenen Persistenz-Schicht bedeutet der Wechsel des Datenbanksystems eine Menge Arbeit. Schauen wir nun weiter in der `hibernate.cfg.xml`. In Zeile 21 sagen wir Hibernate, dass es das Datenbankschema mit dem Tool `hbm2ddl` anhand der Mapping-Meta-Daten, welche in Abschnitt 4.3.2 besprochen werden, generieren bzw. ändern soll. Die Zeilen 24 bis 28 widmen sich den Einstellungen der hier eingesetzten Verbindungspooling-Software C3P0, welche in Abschnitt 3.3 bereits erwähnt wurde. Zeile 31 und 32 weisen Hibernate an, die erzeugten SQL-Statements schön formatiert auf der Konsole auszugeben. Dafür wird `log4j` verwendet, welches über die Datei `log4j.properties` im `src`-Verzeichnis konfiguriert wird. In der `properties`-Datei steht u. a. drin, dass auf `stdout` geloggt wird, wie in Listing 4.5 zu sehen ist. Im letzten Abschnitt in der Datei `hibernate.cfg.xml` (die Zeilen 35 bis 37) wird auf alle Mapping-Dateien verwiesen, die Hibernate beachten soll. Wie und wo man die Mapping-Dateien anlegt, wird im nächsten Abschnitt besprochen.

Listing 4.5: Die Datei `log4j.properties`

```
1  # Output an die Konsole weitergeben
2  log4j.appender.stdout=org.apache.log4j.ConsoleAppender
3  log4j.appender.stdout.Target=System.out
4  log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
5  log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %c
       {1}:%L - %m%n
6
7  # Root-Logger Option
8  log4j.rootLogger=INFO, stdout
9
10 # Hibernate Logging-Optionen (INFO zeigt nur die Startup-
       Nachrichten)
11 log4j.logger.org.hibernate=INFO
12
13 # JDBC Laufzeitparameter ausgeben
14 log4j.logger.org.hibernate.type=INFO
```

4.3.2. Mapping-Dateien erstellen

Zu jeder Klasse, die durch Hibernate persistent gemacht werden soll, muss man eine entsprechende Mapping-Datei im XML-Format anlegen. Dadurch weiß Hibernate, wie die entsprechende Klasse in der Datenbank repräsentiert werden soll. Eine solche Mapping-Datei liegt gewöhlich (aber nicht zwingend) im gleichen Verzeichnis wie die `*.java`-Datei der Java-Klasse. Eine Mapping-Datei benennt man, wie in der Hibernate-Community üblich, im Format `<java-klassenname>.hbm.xml`. Für die `Benutzer`-Klasse heißt das also `Benutzer.hbm.xml`. Den Inhalt dieser betrachten wir uns in Listing 4.6.

Listing 4.6: Die Datei `Benutzer.hbm.xml`

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE hibernate-mapping PUBLIC
3      "-//Hibernate/Hibernate Mapping DTD//EN"
4      "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
5  <hibernate-mapping package="de.fhgiessen.dbs">
6      <class name="Benutzer" table="benutzer">
7          <id name="id" column="benutzer_id">
8              <generator class="increment"/>
9          </id>
10
11         <property name="loginName" column="loginname"/>
12         <property name="email" column="email"/>
13
14         <set name="rollen" table="benutzer_rolle">
15             <key column="benutzer_id"/>
16             <many-to-many class="Rolle" column="rolle_id"
17                     not-found="ignore"/>
18         </set>
19
20         <set
21             name="dokumente">
22             <key column="benutzer_id"/>
23             <one-to-many class="Dokument"/>
24         </set>
25     </class>
26 </hibernate-mapping>
```

Der Inhalt dieser Mapping-Datei dürfte weitestgehend selbsterklärend sein. In den Zeilen 5 und 6 teilen wir Hibernate mit, welche Klasse auf welche Tabelle gemappt werden soll und in welchem Package sie sich befindet. In Zeile 7 bis 9 wird der Primärschlüssel festgelegt, wobei das Attribut `name` des `<id>`-Tags sich auf das gleichnamige Attribut in der Java-Klasse bezieht und das Attribut `column` sich auf die Tabellenspalte. Nun folgen in den Zeilen 11 und 12 weitere (gewöhnliche)

Attribute. In den Zeilen 14 bis 18 wird eine *many-to-many*-Assoziation (auch *n:m*-Assoziation) zur Klasse Rolle definiert. Zur Erinnerung: in der relationalen Datenbankwelt braucht man für *many-to-many*-Verknüpfungen eine Zwischentabelle, die Paare der Primärschlüssel beider Tabellen speichert. Für diese Zwischentabelle muss in Hibernate kein extra Mapping erstellt werden. Man teilt Hibernate lediglich mit, über welche Zwischentabelle diese *many-to-many*-Verknüpfung zustande kommt, in unserem Fall `benutzer_rolle`. In den Zeilen 20 bis 24 wird eine *one-to-many*-Assoziation erstellt: Ein Benutzer kann viele Dokumente haben. Wir schauen uns im Folgenden noch die zwei Mapping-Dateien der Klassen `Rolle` (Listing 4.7) und `Dokument` (Listing 4.8) an.

Listing 4.7: Die Datei `Rolle.hbm.xml`

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE hibernate-mapping PUBLIC
3      "-//Hibernate/Hibernate Mapping DTD//EN"
4      "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
5  <hibernate-mapping package="de.fhgiessen.dbs">
6      <class
7          name="Rolle"
8          table="rolle">
9          <id name="id" column="rolle_id">
10             <generator class="increment"/>
11         </id>
12
13         <property name="rolName" column="rolname"/>
14         <property name="rolKuerzel" column="rolkuerzel"/>
15
16         <set name="benutzer" table="benutzer_rolle">
17             <key column="rolle_id"/>
18             <many-to-many class="Benutzer" column="benutzer_id"
19                 not-found="ignore"/>
20         </set>
21     </class>
22 </hibernate-mapping>
```

Listing 4.8: Die Datei `Dokument.hbm.xml`

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE hibernate-mapping PUBLIC
3      "-//Hibernate/Hibernate Mapping DTD//EN"
4      "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
5  <hibernate-mapping package="de.fhgiessen.dbs">
6      <class
7          name="Dokument" table="dokument">
8          <id name="id" column="dokument_id">
9              <generator class="increment"/>
10         </id>
```

```
11
12          <property name="titel" column="titel"/>
13          <property name="version" column="version"/>
14          <property name="dateiName" column="dateiname"/>
15
16          <many-to-one name="benutzer" column="benutzer_id"
17              insert="false" update="false"/>
18      </class>
19  </hibernate-mapping>
```

In der Mapping-Datei zur Klasse `Rolle` sehen wir in den Zeilen 16 bis 19 wieder das *many-to-many*-Mapping, diesmal aber in die andere Richtung. In der Datei `Dokument.hbm.xml` sehen wir eine *many-to-one*-Assoziation: Viele Dokumente können einem Benutzer gehören. Anhand diesen Mapping-Meta-Daten kann Hibernate auch das Datenbankschema erzeugen, wenn wir in Abschnitt 4.3.4 das Hauptprogramm starten. Doch hier in Listing 4.9 schon einmal vorweg der SQL-Code, den Hibernate für das Erstellen der Tabellen in der Datenbank generiert.

Listing 4.9: SQL-Code – Tabellen für Dokumentenverwaltung

```
1  create table benutzer (
2      benutzer_id bigint not null primary key,
3      loginname varchar(255),
4      email varchar(255)
5  );
6
7  create table rolle (
8      rolle_id bigint not null primary key,
9      rolname varchar(255),
10     rolkuerzel character(1)
11 );
12
13 create table benutzer_rolle (
14     benutzer_id bigint not null references benutzer (benutzer_id),
15     rolle_id bigint not null references rolle (rolle_id),
16     primary key (rolle_id, benutzer_id)
17 );
18
19 create table dokument (
20     dokument_id bigint not null primary key,
21     titel varchar(255),
22     version integer,
23     dateiname varchar(255),
24     benutzer_id bigint references benutzer (benutzer_id)
25 );
```

Nachdem nun die Technik mit den Mapping-Dateien klar ist, wollen wir uns langsam Richtung `MainKlasse` bewegen, in der wir Objekte erstellen und in der Datenbank speichern. Doch vorher besprechen wir noch die Klasse `HibernateUtil`.

4.3.3. Die Klasse „HibernateUtil“

Die Klasse `HibernateUtil` ist eine Hilfsklasse und ist in der Hibernate-Community wohlbekannt. Der Sinn dieser Klasse ist es ein zentrales `SessionFactory`-Objekt zu erstellen, welches von der ganzen Applikation genutzt werden kann. Dies vermeidet das ständige erzeugen und vernichten. Die Klasse ähnelt einer Singleton-Instanz, ist aber durch nur statische Attribute und Methoden realisiert (siehe Listing 4.10). Der Sinn von `SessionFactory` ist, `Session`-Objekte zu erzeugen. Außerdem ist `SessionFactory` thread-sicher. Wir legen für die Klasse ein neues Package namens `persistence` an und legen die Klasse dort hinein.

Listing 4.10: Java-Code – Klasse `HibernateUtil`

```
1  public class HibernateUtil {
2      private static SessionFactory sessionFactory;
3
4      static {
5          try {
6              sessionFactory = new Configuration().configure()
7                      .buildSessionFactory();
8          } catch (Throwable ex) {
9              throw new ExceptionInInitializerError(ex);
10         }
11     }
12
13     public static SessionFactory getSessionFactory() {
14         // Alternativ kann man hier auch JNDI nutzen
15         return sessionFactory;
16     }
17
18     public static void shutdown() {
19         // Caches und Verbindungspools schliessen
20         getSessionFactory().close();
21     }
22
23 }
```

4.3.4. Das Hauptprogramm

Nachdem nun die Arbeitsumgebung eingerichtet ist und alle Klassen und Mappings erstellt wurden, ist es an der Zeit, das Ganze zu testen. Wir erstellen dazu die Klasse `MainKlasse`, die die `main`-Methode enthält. Der komplette Quellcode der Klasse ist in Listing 4.11 zu sehen.

Listing 4.11: Java-Code – Klasse `MainKlasse`

```
1  package de.fhgiessen.dbs;
2
3  import java.util.List;
4  import java.util.Set;
5  import org.hibernate.Session;
6  import org.hibernate.Transaction;
7  import persistence.HibernateUtil;
8
9  public class MainKlasse {
10     public static void main(String[] args) {
11         // 1. Unit of Work beginnen
12         Session session = HibernateUtil.getSessionFactory().
               openSession();
13         Transaction tx = session.beginTransaction();
14
15         // Rollen anlegen und in Persistenz-Kontext aufnehmen
16         Rolle rol1 = new Rolle("Benutzer", 'b');
17         Rolle rol2 = new Rolle("Administrator", 'a');
18         session.save(rol1);
19         session.save(rol2);
20
21         // Dokumente anlegen und in Persistenz-Kontext aufnehmen
22         Dokument dok1 = new Dokument("Reise", new Integer(1), "
               Reise.xml");
23         Dokument dok2 = new Dokument("Test", new Integer(6), "
               test_doc.doc");
24         Dokument dok3 = new Dokument("Abrechnung", new Integer(2),
               "abr.pdf");
25         session.save(dok1);
26         session.save(dok2);
27         session.save(dok3);
28
29         // Benutzer anlegen mit Rolle(n) und Dokument(en)
30         // und in Persistenz-Kontext aufnehmen
31         Benutzer ben1 = new Benutzer("peter", "peter@web.de");
32         session.save(ben1);
33         ben1.getRollen().add(rol1);
34         ben1.getRollen().add(rol2);
```

```
35          ben1.getDokumente().add(dok1);
36
37          Benutzer ben2 = new Benutzer("hilde", "hilde@gmx.at");
38          session.save(ben2);
39          ben2.getRollen().add(rol2);
40          ben2.getDokumente().add(dok2);
41          ben2.getDokumente().add(dok3);
42
43          // Rolle aendern
44          rol1.setRolName("User");
45
46          // Transaktion bestaetigen und Session schliessen
47          tx.commit();
48          session.close();
49
50
51
52          // 2. Unit of Work beginnen
53          session = HibernateUtil.getSessionFactory().openSession();
54          tx = session.beginTransaction();
55
56          // Benutzer ausgeben
57          System.out.println("--- Benutzer -------------------------"
                );
58          List benutzer = session.createQuery("from Benutzer").list()
                ;
59          for (Benutzer b : (List<Benutzer>) benutzer) {
60              // Benutzername, E-Mail-Adresse und ID ausgeben
61              System.out.println("  * " + b.getLoginName() +
62                      " (" + b.getEmail() + ") [" + b.getId() + "]");
63
64              // Rolle des Benutzers ausgeben
65              System.out.println("     * Rollen:");
66              for (Rolle r : (Set<Rolle>)b.getRollen()) {
67                  System.out.println("        * " + r.getRolName() +
68                          " (" + r.getRolKuerzel() + ")");
69              }
70
71              // Dokumente des Benutzers ausgeben
72              System.out.println("     * Dokumente:");
73              for (Dokument d : (Set<Dokument>)b.getDokumente()) {
74                  System.out.println("        * " + d.getTitel() +
75                          " (" + d.getDateiName() + ")");
76              }
77          }
78
```

```
79          // Dokumente ausgeben
80          System.out.println();
81          System.out.println("--- Dokumente ------------------------"
                );
82          List dokumente = session.createQuery("from Dokument").list
                ();
83          for (Dokument d : (List<Dokument>)dokumente) {
84              // Dokumenttitel, -dateiname und ID ausgeben
85              System.out.println("  * " + d.getTitel() +
86                      " (" + d.getDateiName() + ") [" + d.getId() + "
                            ] {" +
87                      d.getBenutzer().getLoginName() + "}");
88          }
89
90          // Rollen ausgeben
91          System.out.println();
92          System.out.println("--- Rollen --------------------------"
                );
93          List rollen = session.createQuery("from Rolle").list();
94          for (Rolle r : (List<Rolle>)rollen) {
95              // Rollenname, -kuerzel und ID ausgeben
96              System.out.println("  * " + r.getRolName() + " ("
97                      + r.getRolKuerzel() + ") [" + r.getId() + "]");
98
99              // Benutzer ausgeben, die diese Rolle haben
100             System.out.println("     * Benutzer:");
101             for (Benutzer b : (Set<Benutzer>)r.getBenutzer()) {
102                 System.out.println("        * " + b.getLoginName())
                        ;
103             }
104         }
105
106         // Transaktion bestaetigen und Session schliessen
107         tx.commit();
108         session.close();
109
110         // Programm beenden
111         HibernateUtil.shutdown();
112     }
113 }
```

Der Quellcode ist in zwei Teile aufgeteilt, welche auch zwei verschiedene Transaktionen wiederspiegeln. Im ersten Teil (1. Unit of Work) werden Objekte der Klassen `Benutzer`, `Rolle` und `Dokument` erzeugt und miteinander in Verbindung gebracht. Danach sind die Objekte in der Datenbank präsent. Im zweiten Teil (2. Unit of Work) werden die Objekte aus der Datenbank gelesen und am Bildschirm

ausgegeben.

Erster Unit of Work

In Zeile 12 wird von der `SessionFactory` aus `HibernateUtil` eine `Session` erzeugt. In Zeile 13 wird eine Transaktion begonnen. In den Zeilen 16 bis 19 werden zwei `Rolle`-Objekte angelegt und in den Persistenz-Kontext aufgenommen, gefolgt von drei `Dokument`-Objekten in den Zeilen 22 bis 27. In den Zeilen 31 bis 35 wird ein `Benutzer`-Objekt angelegt und diesem zwei Rollen und ein Dokument hinzugefügt. In den Zeilen 37 bis 41 wird ein weiterer Benutzer angelegt, welchem eine Rolle und zwei Dokumente zugeweisen werden. Hibernate hat alle diese Objekte im Persistenz-Kontext und überwacht sie auf Änderungen. Die Objekte werden nach Hibernates Ermessen in die Datenbank gespeichert bzw. „geupdatet“. In Zeile 44 wird ein `Rolle`-Objekt geändert. Hibernate bekommt dies mit und aktualisiert den Datensatz. In den Zeilen 47 und 48 wird die Transaktion durch `commit()` (erfolgreich) beendet und die Session geschlossen. Wenn man während der Transaktion merkt, dass ein Fehler aufgetreten ist (z. B. Exception), kann man die Transaktion auch durch `rollback()` beenden. Das Datenbanksystem stellt dann den alten Zustand, wie er vor der Transaktion war, wieder her.

Zweiter Unit of Work

In den Zeilen 53 und 54 starten wir eine neue Session und eine neue Transaktion. In den Zeilen 57 bis 77 geben wir die Daten aller Benutzer aus. Dazu holen wir die Daten mit Zeile 58 aus der Datenbank. Wir verwenden hier die Query-API von Hibernate mit HQL. Wir geben für jeden Benutzer seine Rollen und seine Dokumente aus. Nach der Ausgabe aller Benutzer, geben wir noch alle Dokumente und alle Rollen am Bildschirm aus. Die Bildschirm-Ausgabe ist in Listing 4.12 zu sehen. Der restliche Quellcode sollte selbsterklärend sein.

Listing 4.12: Ausgabe der Klasse `MainKlasse`

```
1  --- Benutzer --------------------------------
2    * peter (peter@web.de) [29]
3       * Rollen:
4          * Administrator (a)
5          * User (b)
6       * Dokumente:
7          * Reise (Reise.xml)
8    * hilde (hilde@gmx.at) [30]
9       * Rollen:
10         * Administrator (a)
11      * Dokumente:
12         * Abrechnung (abr.pdf)
```

```
13          * Test (test_doc.doc)
14
15 --- Dokumente ------------------------------
16   * Reise (Reise.xml) [1] {peter}
17   * Abrechnung (abr.pdf) [3] {hilde}
18   * Test (test_doc.doc) [2] {hilde}
19
20 --- Rollen ---------------------------------
21   * Administrator (a) [2]
22      * Benutzer:
23         * hilde
24         * peter
25   * User (b) [1]
26      * Benutzer:
27         * peter
```

Wir haben nun gesehen, welche Schritte nötig sind, um ein Projekt mit Hibernate durchzuführen. Das in diesem Kapitel Demonstrierte ist noch lange nicht alles, was Hibernate kann. Deshalb folgt im nächsten Kapitel ein Überblick über weitere Features von Hibernate.

5. Weitere Features von Hibernate

Im Folgenden wird ein kurzer Überblick über weitere Features von Hibernate gegeben.

Neben den in dieser Seminararbeit verwendeten *many-to-many*-, *many-to-one*- und *one-to-many*-Assoziationen gibt es z. B. noch die *one-to-one*-Assoziation. Hibernate kann aber, wie erwartet, nicht nur Assoziationen mappen, sondern auch Vererbung und das sogar auf unterschiedliche Arten und Weisen. Außerdem unterstützt Hibernate Mapping-Annotationen direkt im Quellcode der Persistenz-Klassen. Dies erspart die XML-Mapping-Dateien, bietet aber nicht alle Möglichkeiten, die in XML-Mapping-Dateien verfügbar sind. Die Schnittstelle für die JPA ist bei Hibernate der `EntityManager`. Hibernate bietet außerdem noch eine sehr gute Möglichkeit Daten aus der Datenbank zu holen über die Criteria-API (Klasse `Criteria`). Man kann damit Restriktionen setzen wie z. B. die (maximale) Anzahl der Ergebnisse, die aus der Datenbank geholt werden und weitere Suchkriterien. Mit `Criteria` kann man Abfragen auf Objekt-Ebene formulieren und braucht kein SQL oder HQL.

Hibernate befindet sich in ständiger Weiterentwicklung, sodass permanent Neuerungen hinzukommen. Aber auch das in dieser Seminararbeit Besprochene rund um Hibernate dürfte die Mächtigkeit dieses Tools demonstrieren.

6. Zusammenfassung

In dieser Seminararbeit wurden im Wesentlichen der Umgang mit dem Tool „Hibernate“ und die Grundlagen des objekt-relationalen Mappings (ORM) beschrieben. Nach der Einführung in die Thematik in Kapitel 1, wurde in Kapitel 2 näher auf die theoretischen Grundlagen beim objekt-relationalen Mapping eingegangen. Dabei wurden verschiedene Persistenz-Mechanismen und generelle Probleme des ORM diskutiert. Es folgten in Kapitel 3 die grundlegenden Dinge von Hibernate, bevor in Kapitel 4 anhand eines Beispiel-Projekts der Umgang mit Hibernate demonstriert wurde. Kapitel 5 gab am Ende nochmal einen kurzen Ausblick auf weitere Hibernate-Features.

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Bürgeln, 3. Juni 2008 _____

Unterschrift

Abbildungsverzeichnis

Listings

Literaturverzeichnis

[BK07] Bauer, Christian ; King, Gavin: *Java-Persistence mit Hibernate*. Hanser Fachbuch, 2007. – ISBN 3–44640–9416

[hib07] *Hibernate API Documentation (3.2.2.ga)*. `http://www.hibernate.org/hib_docs/v3/api/`. Version: 2007. – [Online; Stand 24. Januar 2007]

[Ren07] Renz, Prof. Dr. B.: *Vorlesung Datenbanksysteme*. Fachhochschule Gießen-Friedberg, 2007. – [Stand Sommersemester 2007]

[Wik07] Wikipedia: *Surrogatschlüssel – Wikipedia, Die freie Enzyklopädie*. `http://de.wikipedia.org/w/index.php?title=Surrogatschl%C3%BCssel&oldid=40227688`. Version: 2007. – [Online; Stand 26. Mai 2008]

[Wik08] Wikipedia: *Persistenz – Wikipedia, Die freie Enzyklopädie*. `http://de.wikipedia.org/w/index.php?title=Persistenz&oldid=45455623`. Version: 2008. – [Online; Stand 22. Mai 2008]

A. Quellcode Dokumentenverwaltung

Listing A.1: Java-Code – Klasse `Benutzer` (vollständig)

```
1  package de.fhgiessen.dbs;
2
3  import java.util.HashSet;
4  import java.util.Set;
5
6  public class Benutzer {
7      private Long id;
8      private String loginName;
9      private String email;
10     private Set rollen = new HashSet();
11     private Set dokumente = new HashSet();
12
13     public Benutzer() {
14     }
15
16     public Benutzer(String loginName, String email) {
17         super();
18         this.loginName = loginName;
19         this.email = email;
20     }
21
22     public Long getId() {
23         return id;
24     }
25
26     private void setId(Long id) {
27         this.id = id;
28     }
29
30     public String getLoginName() {
31         return loginName;
32     }
33
34     public void setLoginName(String loginName) {
35         this.loginName = loginName;
```

```
36      }
37
38      public String getEmail() {
39          return email;
40      }
41
42      public void setEmail(String email) {
43          this.email = email;
44      }
45
46      public Set getRollen() {
47          return rollen;
48      }
49
50      public void setRollen(Set rollen) {
51          this.rollen = rollen;
52      }
53
54      public Set getDokumente() {
55          return dokumente;
56      }
57
58      public void setDokumente(Set dokumente) {
59          this.dokumente = dokumente;
60      }
61
62  }
```

Listing A.2: Java-Code – Klasse `Rolle` (vollständig)

```
1  package de.fhgiessen.dbs;
2
3  import java.util.HashSet;
4  import java.util.Set;
5
6  public class Rolle {
7      private Long id;
8      private String rolName;
9      private Character rolKuerzel;
10     private Set benutzer = new HashSet();
11
12     public Rolle() {
13     }
14
15     public Rolle(String rolName, Character rolKuerzel) {
16         super();
17         this.rolName = rolName;
```

```
18          this.rolKuerzel = rolKuerzel;
19      }
20
21      public Long getId() {
22          return id;
23      }
24
25      private void setId(Long id) {
26          this.id = id;
27      }
28
29      public String getRolName() {
30          return rolName;
31      }
32
33      public void setRolName(String rolName) {
34          this.rolName = rolName;
35      }
36
37      public Character getRolKuerzel() {
38          return rolKuerzel;
39      }
40
41      public void setRolKuerzel(Character rolKuerzel) {
42          this.rolKuerzel = rolKuerzel;
43      }
44
45      public Set getBenutzer() {
46          return benutzer;
47      }
48
49      public void setBenutzer(Set benutzer) {
50          this.benutzer = benutzer;
51      }
52
53  }
```

Listing A.3: Java-Code – Klasse `Dokument` (vollständig)

```
1  package de.fhgiessen.dbs;
2
3  public class Dokument {
4      private Long id;
5      private String titel;
6      private Integer version;
7      private String dateiName;
8      private Benutzer benutzer;
```

```
9
10      public Dokument() {
11      }
12
13      public Dokument(String titel, Integer version, String dateiName
            ) {
14          super();
15          this.titel = titel;
16          this.version = version;
17          this.dateiName = dateiName;
18      }
19
20      public Long getId() {
21          return id;
22      }
23
24      private void setId(Long id) {
25          this.id = id;
26      }
27
28      public String getTitel() {
29          return titel;
30      }
31
32      public void setTitel(String titel) {
33          this.titel = titel;
34      }
35
36      public Integer getVersion() {
37          return version;
38      }
39
40      public void setVersion(Integer version) {
41          this.version = version;
42      }
43
44      public String getDateiName() {
45          return dateiName;
46      }
47
48      public void setDateiName(String dateiName) {
49          this.dateiName = dateiName;
50      }
51
52      public Benutzer getBenutzer() {
53          return benutzer;
```

```
54      }
55
56      public void setBenutzer(Benutzer benutzer) {
57          this.benutzer = benutzer;
58      }
59
60  }
```