HS München Fakultät 07

Studiengang Bachelor Wirtschaftsinformatik FWP

Aktuelle Technologien zur Entwicklung verteilter Java-Anwendungen

Die Integration von Datenbanken mittels JPA 2

Maximilian Spelsberg, 05534010

02.05.2014

Erstprüfer/-in / Betreuer/-in: Herr Theis Michael

Inhaltsverzeichnis

1. Einführung

Das Einbinden von Datenbanken in die verschiedensten Anwendungen ist längst keine Besonderheit mehr, sondern als selbstverständlich anzusehen. Fast jede Software nutzt heute eine oder mehrere Datenbanken im Hintergrund, die zum Speichern, Verwalten und Auswerten von Datensätzen dienen.

Es existieren unterschiedliche Managementansätze für Datenbanken, die grundsätzlich in relationale (RDBMS), objektorientiere (OODBMS) und objektrelationale (ORDBMS) Datenbankmanagementsysteme unterschieden werden können. Jedes Modell bringt situationsbezogen Vor- und Nachteile mit sich und sollte deshalb gründlich durchleuchtet werden, bevor eine Wahl getroffen wird. Bisher war ein direkter Vergleich allerdings kaum nötig, da das relationale Schema am weitesten verbreitet ist und die Wahl deshalb, trotz manch einhergehender Nachteile, fast ausschließlich auf dieses Managementsystem fällt. Das relationale Datenschema ist bis dato auch nur als einziges ausreichend erprobt und wird von großen Unternehmen wie Oracle als Lösung angeboten.

Gleichzeitig haben sich verschiedene Ansätze in der modernen Softwareentwicklung hervorgehoben, die das Verwalten und Verarbeiten von Daten im System gewissen Regeln unterwerfen. Das bekannteste Konzept, das heute eine Grundlage für weitere Methodenansätze liefert, ist die objektorientierte Programmierung.

Damit haben sich auf der einen Seite relationale Datenmodelle für Datenbanken und auf der anderen Seite objektorientierte Datensätze im Programmcode zum Standard entwickelt, obwohl diese komplett unterschiedliche Ansätze der Datenverwaltung vertreten. Dies hat zur Folge, dass in vielen Projekten diese völlig unterschiedlichen Verarbeitungsmethoden zusammengeführt werden müssen, um eine gute Kommunikation unter den Gesichtspunkten der Verarbeitungsgeschwindigkeit und Fehlerresistenz zu gewährleisten.

1.1 Schichtenarchitektur

Ein heute schon standardisiertes Vorgehen in der Softwarearchitektur, ist das Aufteilen einer Applikation in mehrere Schichten. Die „Drei-Schichten-Architektur“ oder die auch als „Drei-Schichten-Modell“ bekannte Darstellung bildet dabei die einfachste Methode der Schichtentrennung, wie auch in Abbildung 1 zu sehen ist. Das Thema JPA beziehungsweise das Kommunizieren mit einer Datenbank findet alleine auf der untersten Schicht, der „Persistence Layer“-Schicht, des Modells statt.

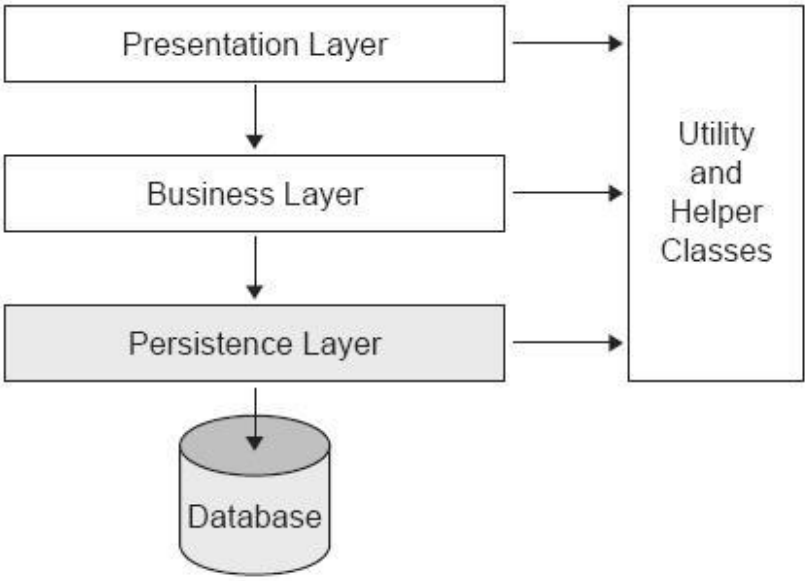

Abbildung 1: Die „Drei-Schichten-Architektur“, die den „Persistence Layer“ als Basis einer Applikation zeigt. Quelle: Bauer C. & King G., „Java Persistence with Hibernate“, Manning, 2007, Seite 21

1.2 Ziel dieser Arbeit

Ziel dieser Arbeit soll es sein, einen einfachen Weg für den Aufbau der Kommunikation zwischen objektorientierter Anwendung und relationalen Datenbanken in Java aufzuzeigen. Dafür wird das als „JPA“ bezeichnete Java ORM beleuchtet und in dem beiliegenden Programmcode auch praktisch angewandt. Dem Aufbau folgend, wird nach dem Definieren der grundliegenden Begriffe, Schritt für Schritt eine Vorgehensweise aufgezeigt, durch die das Anwenden von JPA und deren Funktionen leicht nachgestellt werden kann. Dabei soll neben Codebeispielen direkt zum Text, auch ein beigefügter Programmcode dem Verständnis dienen. Sowohl eingefügter, als auch beiliegender Code ist in Java geschrieben und es wird eine gewisse Kenntnis dieser Sprache vorausgesetzt. Der eingefügte Code ist stets zeilen- oder abschnittsweise kommentiert und wird oft zusätzlich durch nachfolgende Erklärungen ausführlicher behandelt.

1.3 Ziel der beigefügten Anwendung

Neben der schriftlichen Ausarbeitung, soll ein in Java programmierter Code ein Beispiel liefern, wie JPA verwendet werden kann. Zu den weiteren Themen wird dieser Code öfter herangezogen, um detaillierte und praxisbezogene Beispiele geben zu können. Nachfolgend wird ein kurzes Szenario aufgestellt, das das Programm abbilden soll.

Ein auf Kapitalanlage spezialisiertes Unternehmen möchte durch ein im Backend verwendetes Java Programm alle Datensätze in einer gemeinsamen Datenbank speichern und über eine darüber verwendeten Datendrehscheibe alle eingehenden und ausgehenden Datensätze verwalten. Als einer der ersten Schritte wird eine Anwendungsschnittstelle zu einer relationalen Datenbank implementiert, um eine erste Kommunikation zwischen Java und in diesem Fall MySQL zu gewährleisten. In dieser

Beispielanwendung wird ein einfaches ER-Modell verwendet, das im späteren Verlauf natürlich noch beliebig hoch skaliert werden kann.

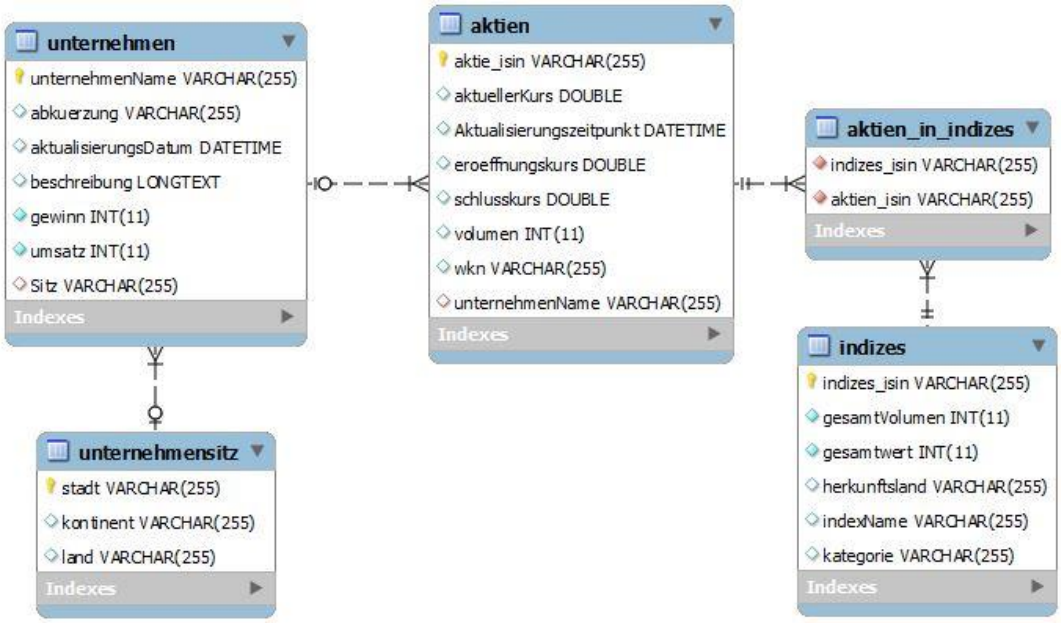

Abbildung 2: Entity-Relationship-Modell der beiliegenden Anwendung.

Der Programmcode soll in einer einfachen Form eine Persistenz Schicht („Persistence Layer“) abbilden. Erhaltene Datensätze sollen auf Basis von JPA in das ER-Modell einer relationalen MySQL Datenbank geschrieben werden. Vorerst sollte allerdings geklärt werden, was es mit „Object/Relational Persistence“ und ORM auf sich hat und was für einen Ansatz JDBC verfolgt beziehungsweise wie JDBC angewendet werden kann.

2. Die klassische Vorgehensweise mit JDBC

JDBC oder auch als „Java Database Connectivity“ bekannt, ist eine API die bereits in Java 1.1 Einzug gefunden hat. [1] Diese API liegt auf dem SQL-Level und erlaubt es, mittels in Java Code geschriebenen SQL-Befehle, mit einer unabhängigen Datenbank zu kommunizieren. [2] Solange die Datenbank relational ist und die jeweiligen SQL-Befehle unterstützt, ist dies eine sehr fehlerresistente und schnelle Methode der Datenbankkommunikation. Folgendes Beispiel zeigt, wie JDBC grundlegend funktioniert:

```
// Aufbauen einer Verbindung
Connection connection = DriverManager.getConnection(DB_URL,USER,PASSWORD);

//Definieren eines Datenbankzugriffs mittels SQL
Statement statement = connection.createStatement();
String sql = "SELECT id, aktie, preis, datum FROM AktienTabelle";
ResultSet resultset = statement.executeQuery(sql);

//Schließen der Verbindung
resultset.close();
statement.close();
connection.close();
```

[1] Vgl. http://www.hs-owl.de/fb5/labor/it/de/sd/uebung/jdbc.pdf, Seite 1, Zugriff 28.04.2013

[2] Vgl. George Reese, Database Programming with JDBC and Java, O’Reilly & Associates Inc., 2 Edition, August 2000, Seite 10

Verbindungen müssen manuell geöffnet und geschlossen werden und die SQL-Befehle werden starr und unflexibel an den Code gebunden. Dabei muss alles bis auf die sogenannte CRUD-Ebene herunter manuell implementiert werden, die die absoluten Grundbefehle Create, Read, Update und Delete definiert. Sobald in größeren Projekten die Design-Entscheidung auf JDBC fällt, ist es daher durchaus möglich, dass ein Drittel der kompletten Programmierarbeit auf das manuelle Überbrücken der ungleichen Paradigmen entfällt. [3] Ein großes Problem ist bei dieser Strategie die Skalier- und Anpassbarkeit.

3. „Object/Relational Persistence“

Da relationale Datenbankmanagementsysteme weder direkt für Java, noch für bestimmte Anwendungen ausgelegt sind, spricht man von dem Prinzip der unabhängigen Daten oder auch „indiependent data“. [4] Dies ist ein wichtiges Prinzip, da Datenbanken anwendungsübergreifend dienen sollen. Dass ein nicht übereinstimmen der Paradigmen von objektorientierten und relationalen Datensätze existiert, wurde ja bereits erläutert und um dieses Dilemma zu umgehen, hat sich vor allem ein Vorgehen durchgesetzt: Das Objekt-Relationale Mapping (oder ORM). Zuerst sollte jedoch zumindest die Begriffsdefinition zu Persistenz geklärt sein.

3.1 Was ist Persistenz?

Wenn in dieser Arbeit von Persistenz (engl. „Persistence“) beziehungsweise persistenten Daten die Rede ist, versteht man das Speichern der Daten in relationale Datenbanken. [5] Im Allgemeinen kann man persistente Daten auch als nicht flüchtig abgespeicherte Datensätzen in permanenten Speichersystemen definieren, die bidirektional – also sowohl das Schreiben, als auch das Lesen - unterstützen. Demnach können Datenbanken auch als persistente Speichermedien bezeichnet werden.

3.2 ORM – Objekt-Relationales Mapping

Nach den Überlegungen, kaum verbreitete objektorientierte Datenbanken an die Anwendung zu binden oder alles bis auf die CRUD-Befehle heruntergebrochen manuell zu implementieren, kommt nun eine dritte und die bisher am weitesten verbreitete Methode des Verwalten von persistenten Daten. Die Rede ist von einem automatisierten Mapping, das manuelles Einpflegen von SQL-Code in den Programmcode, bis zu einem gewissen Grad, überflüssig macht. Durch das Objekt-Relationale Mapping oder kurz ORM wird eine automatische und transparente Persistenz von Objekten, zum Beispiel aus der Java-Anwendung und den Tabellen einer relationalen Datenbank, durch das Heranziehen von Metadaten, erzielt. [6]

ORM bringt zwar wieder etwas Komplexität in die Anwendung und erfordert einiges an Knowhow, allerdings bringt es neben der bereits erwähnten Anpassungsflexibilität und

[3] Vgl. Bauer C. & King G., „Java Persistence with Hibernate“, Manning, 2007, Seite 19

[4] Vgl. Bauer C. & King G., „Java Persistence with Hibernate“, Manning, 2007, Seite 5

[5] Vgl. Bauer C. & King G., „Java Persistence with Hibernate“, Manning, 2007, Seite 5

[6] Vgl. Bauer C. & King G., „Java Persistence with Hibernate“, Manning, 2007, Seite 25

Skalierbarkeit noch weitere Vorteile. Es entsteht eine höhere Produktivität, da der bzw. die Entwickler sich mehr auf die Business-Logik konzentrieren können, als sich mit diesen grundlegenden Arbeiten abmühen zu müssen. Gleichzeitig entsteht eine bessere Lesbarkeit des Codes, da weniger Lines of Code (LoC) verwendet werden. [7]

4. Java Persistence API

Die Java Persistenz API ist eine alleinstehende und standardisierte Persistenz-Technologie, die unabhängig vom EJB-Container sowohl innerhalb, als auch außerhalb ausführbar ist. [8] Zwar ist JPA ursprünglich im Rahmen von EJB 3 entstanden und wurde 2006 mit der EJB 3.0 Spezifikation in der Version 1.0 als standardisierte Java Spezifikation veröffentlicht, ist jedoch ebenfalls auch separat für das Standard-Java, als auch für die Java Enterprise Edition anwendbar. Die Version 1.0 war ein Meilenstein, der bis heute die grundlegenden Funktionen von JPA liefert. Dabei funktioniert JPA seither nach dem POJO-Prinzip und macht das implementieren von Interfaces überflüssig. [9] Um den zu komplexen Ansätzen der Vorgänger von EJB 3 entgegenzuwirken, wurden nach dem „Back-to-Basics“-Prinzip einfache Java-Objekte oder auch „Plain Old Java Objects“ genannt beziehungsweise JavaBeans, eingebunden. JavaBeans werden oft als Synonym zu POJOs behandelt und setzen ein paar grundlegende Konventionen, wie zum Beispiel das Verwenden von Getter/Setter, Default Konstruktoren und die Serialisierbarkeit voraus. Diese simplen Objekte, die zu speichernde Informationen bereitstellen, werden auch als Entitäten bezeichnet, sobald sie gewisse Charakteristiken erfüllen:

- Persistability – Legt grundlegend fest, dass ein Objekt immer persistent gemacht werden kann.
- Identity – Setzt voraus, dass das Objekt eine eindeutige Identität aufweist, die auch als Primärschlüssel in der Datenbank verwendet werden kann.
- Transactionality – Definiert, dass eine ständige Bereitschaft für eine Transaktion, die aus den CRUD-Befehlsätzen bestehen kann, erfüllt ist.
- Granularity – Beschreibt den Aufbau der Variablen innerhalb eines Objekts. Enitäten enthalten einfache, nicht durch z. B. Arrays verpackte Datensätze, die auch keine semantischen Ziele verfolgen.

[10]

Seit 2013 liegt die API der Java Persistence in der Version 2.1 vor, welche wieder neue Features mit sich bringt, auf die wir an späterer Stelle, nach dem klären der grundliegenden Funktionen, eingehen werden.

Für das Verwenden der API steht allerdings noch eine weitere Entscheidungsmöglichkeit für ein Projekt aus, nämlich welches Framework verwendet werden soll, um auf die ORM-Schnittstelle zuzugreifen.

---

[7] Vgl. Bauer C. & King G., „Java Persistence with Hibernate“, Manning, 2007, Seite 29f

[8] Vgl. Backschat Martin, Rücker Bernd, Enterprise JavaBeans 3.0, Elsevier GmbH, 2007, Seite 113

[9] Vgl. Backschat Martin, Rücker Bernd, Enterprise JavaBeans 3.0, Elsevier GmbH, 2007, Seite 113

[10] Vgl. Keith M. & Schincariol M., Pro JPA 2, Second Edition, Seite 15f

4.1 Persistenz Frameworks

Ein Framework ist unabdingbar, um JPA verwenden zu können. Hierbei stehen einem mehrere Frameworks zur Verfügung, wie die nachfolgende Abbildung 3 zeigt.

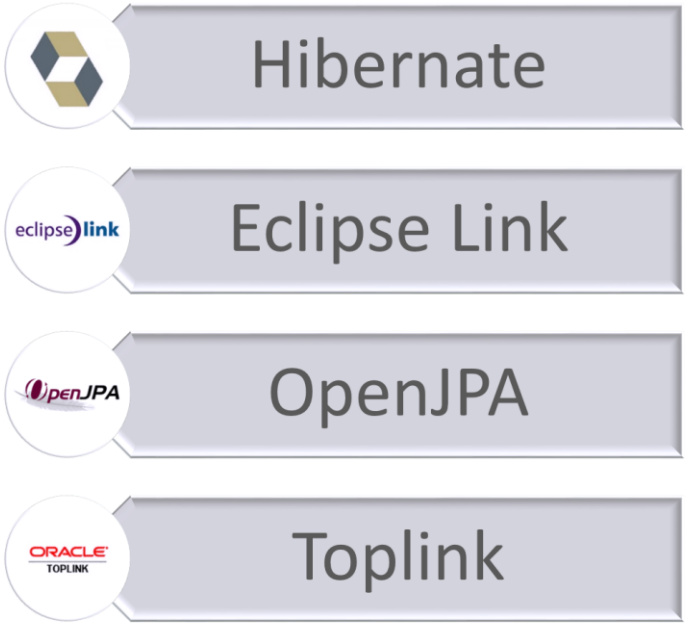

Abbildung 3: Auflistung der gängigsten Frameworks

Die zwei beliebtesten Produkte sind das kommerzielle TopLink von Oracle, das bereits zu den Anfangszeiten von Java angeboten wurde und das als Hibernate bezeichnete Open-Source Projekt. Einige der besten Funktionen aus diesen Frameworks sind dabei in spätere JPA-Spezifikationen geflossen. [11] Für den anhängenden Code wurde Hibernate herangezogen. Allerdings ist dies dem Code nicht anzumerken, da bis auf die mitgelieferten Bibliotheken in den Abhängigkeiten des „Project Object Model“ von Maven, Hibernate nicht im Code auftauchen und nur reines JPA verwendet wurde.

5. Das Mapping von Objekten und relationalen Datenbanken

Im Folgenden gehen wir auf die Vorgehensweisen beim ORM ein und behandeln im Aufbau einer persistenten Datenspeicherung die verschiedenen Funktionsmöglichkeiten, die vor allem durch sogenannte Metadaten vorgenommen werden können. Für das Erstellen einer persistenten Schicht in Java, muss, wie bei Java Projekten grundsätzlich immer, ein Projekt angelegt sein. Dieses muss jedoch bestimmte Voraussetzungen erfüllen, wie das Implementieren einer persistence.xml die in Punkt 5.3 detaillierter beschrieben wird. JPA, Hibernate und die anderen Frameworks bieten dazu auch automatisierende Erzeugungsfunktionen für die Programmierwerkzeuge wie Eclipse oder Netbeans.

5.1 Anlegen von Java-Entitäten und Datenbankschema

Es gibt unterschiedlichen Ausgangsmöglichkeiten, wobei wir auf die folgenden typischen Fälle eingehen werden:

[11] Vgl. Keith M. & Schincariol M., Pro JPA 2, Second Edition, Seite 8

Das als „Top down“ bezeichnete Szenario beschreibt ein bereits angelegt oder geplantes Domainmodel, das in eine bisher nicht existierende Datenbank gespeichert werden soll. Dagegen wird unter dem „Bottom up“-Szenario verstanden, dass ein Datenbankschema bereits existiert und die dazugehörenden Enitäten noch zu erstellen sind. [12] Also ist entweder bereits ein Datenbankschema vorhanden, das gefüllt werden muss oder es ist fachlich definiert, dass Objekte in eine oder mehrere relationale Datenbanken gespeichert werden und das Datenbankschema aus den Objekten im Code generiert werden soll. Für beide Fälle liegen Automatisierungsfunktionen vor, die eine Menge Arbeit abnehmen können.

5.1.1 Generieren von Java-Entitäten

Das Vorgehen, mit dem Entitäten aus einem Datenbankschema generiert werden, wird auch als „Reverse Engineering“ bezeichnet. Da es keine in JPA spezifizierte Funktion ist, sondern über ein extra Framework beziehungsweise Tool beschafft werden muss, wird hier das Thema nur der Vollständigkeit halber erwähnt. Durch ein Tool von Hibernate, das als HibernateToolTask bezeichnet wird, lassen sich beispielsweise XML-Dateien und Java-Code aus einem Datenbankschema generieren. [13] Gleiches leisten mittlerweile auch die IDEs (Integrierte Entwicklungsumgebungen) wie Eclipse und NetBeans. [14]

5.1.2 Anlegen eines Datenbankschemas aus Java-Entitäten

Für das Mapping benötigte Metadaten können entweder direkt im Code der POJO-Objekte durch Annotationen oder ausgelagert durch die bei simultaner Verwendung immer vorrangig behandelter XML-Dateien definiert werden. [15] Annotationen haben erst seit Java 5 Einzug in die Spezifikation gefunden, weshalb das Verwalten von Metadaten in XML-Files ursprünglich dem üblichen Vorgang entsprach und von den Frameworks unterstützt wurde bzw. noch immer wird. Nachfolgend ein abgeleitetes Beispiel für eine in Hibernate verwaltete XML-Datei von der im Beispielprogramm beiliegenden Entität „Aktie.java“:

```
<hibernate-mapping><class name="Aktie" table="Aktien">
      <meta attribute="class-description">
         Diese Klasse enthält alle Informationen zu Aktien.
         Diese werden in der gleichnamigen Tabelle gespeichert!
         @author Maximilian Spelsberg
      </meta>
      <id name="isin" type="string" column="ISIN">
         <generator class="native"/>
      </id>
      <property name="unternehmenName" column="Firma" type="string"/>
</class></hibernate-mapping>
```

Grundsätzlich sagt dieser Ausschnitt aus, dass Hibernate die Klasse „Aktie.java“ injizieren soll und bei der Verarbeitung einige Eigenschaften beachten muss. Es wird der Tabellenname „Aktien“ definiert, eine Klassenbeschreibung, die dem klassischen Javadoc entspricht, ein Primärschlüssel mit dem Namen „isin“ und eine erste Variable mit der Bezeichnung „unternehmenName“, vom Typ String und

---

[12] Vgl. Bauer C. & King G., „Java Persistence with Hibernate“, Manning, 2007, Seite 40

[13] Vgl. Bauer C. & King G., „Java Persistence with Hibernate“, Manning, 2007, Seite 88

[14] Vgl. Ullenboom C, „Java 7 – Mehr als eine Insel“, http://openbook.galileocomputing.de/java7/1507_16_013.html, Zugriff: 30.04.2014

[15] Vgl. Backschat Martin, Rücker Bernd, Enterprise JavaBeans 3.0, Elsevier GmbH, 2007, Seite 113

dem Spaltennamen „Firma“ festgelegt. Dieses Vorgehen, das früher manuell gepflegt werden musste, kann heute ebenfalls automatisierenden Frameworks wie AndroMDA, Middlegen, XDoclet oder ähnlichen XML-Generatoren überlassen werden beziehungsweise liefern die Persistenzframeworks wie Hibernate einen eigenen „basic wizard“ gleich mit, der direkt auf das Framework angepasste XML-Dateien mit Namenskonventionen generiert.

Heute wird häufig die Vorgehensweise der Metaisierung direkt im Sourcecode gewählt, da dies dem Entwickler gewisse Vorteile verschafft. Die Zuordung der Metadaten sind viel ersichtlicher, da alle Informationen zentral und in den passenden Codezeilen implementiert werden. Außerdem lassen sich Annotationen von Codegenerierungstools auslesen und ebenso während der Laufzeit per Reflection abfragen, wenn man das möchte. [16] Parallel zu dem bereits gezeigten XML-Code, ist folgend eine Entität mit Annotationen aufgeführt, die identisch mit der XML-Variante ist.

```
Import javax.persistence.*;

/**
 * Diese Klasse enthält alle Informationen zu Aktien.
 * Diese werden in der gleichnamigen Tabelle gespeichert!
 * @author Maximilian Spelsberg
 */
@Entity
@Table(name="Aktien")
public class Aktie implements Serializable {

  @Id
  @Column(name="ISIN)
  private String isin;

  @Column(name="Firma")
  private String unternehmenName;
}
```

Für die Vereinfachung wurden Getter, Setter und Default-Konstruktoren außen vor gelassen, aber sowohl in der XML, als auch in den Annotationen ist das Definieren von Tabellennamen (@Table) und den Spaltennamen (@Column) nicht unbedingt nötig. Die Frameworks gehen normalerweise immer davon aus, dass bei Fehlen dieser Angaben, der Variablen- und Klassenname verwendet werden soll. Demnach reicht in einer minimalistischen Form die Kennzeichnung, dass es sich um eine Entität handeln soll (@Entity) und eine eindeutige Kennung (@Id). Die folgenden Metadaten behandeln wir nur in Annotationsform, um eine gewisse Übersichtlichkeit herzustellen und ausschließlich aus der Java Persistence API javax.persistence.*, da diese die Grundlage bilden und zusätzliche Funktionen der Bibliotheken angewendeter Frameworks eher Spezialfälle und erweiterte Funktionalitäten bieten. Alle beiliegenden Beispielentitäten sind ebenfalls nur in Annotationsform aus JPA verfasst.

5.2 Entitäten speichern, laden, updaten und löschen

Durch das Erstellen der ersten Entitäten, wurden Objekte auf der Programmseite bereitgestellt, die allerdings erst durch die passenden Klassenaufrufe und der JPA-Logik zu den gewünschten Aktionen führen.

[16] Vgl. Kehle M., Hien R, Röder D., JPA mit Hibernate – Java Persistence API in der Praxis, entwickler.press, 2010, Seite 21

5.2.1 Entitäten Manager

Vor dem Durchführen eines der CRUD-Befehlssätze, muss immer eine sogenannte „Entity Manager Factory“ gestartet werden, aus der pro Transaktion oder fachlichen Transaktionsketten ein Persistenzmanager oder Entitäten Manager (englisch „persistent manager“ oder „Entity Manager“) gebildet wird. Ohne diesen Managern ist jede Entität nur ein einfaches nicht-persistentes Javaobjekt, denn dieser ist, nach dem erstellen, für das Delegieren der Arbeit zum Schaffen einer persistenten Struktur notwendig. [17] Die Entity Manager Factory stellt wie eine Fabrik beliebig viele und vollkommen identische Manager her, was sie zu einer Art Template macht. Die Vorgaben bzw. die Konfiguration der Fabrik wird durch die in der persistence.xml festgehaltene Persistence-Unit vorgenommen.

```
//Deklarieren und initialisieren der EntityManagerFactory und eines EntityManagers
EntityManagerFactory fabrik =
 Persistence.createEntityManagerFactory(„PersisteceUnit_Name“);
EntityManager manager = fabrik.createEntityManager();

//Der Anfang einer Transaktion beginnt immer wie folgt:
manager.getTransaction().begin();

//Entität wird instanziiert & transaktionsspezifische Operationen durchgeführt
Aktie aktie = new Aktie();
Aktie.setValue(value);

//Die Transaktion wird abgeschlossen, der Manager, der seine Aufgabe erfüllt hat,
//zum Schluss entlassen und die Fabrik geschlossen
manager.persist(aktie);
manager.getTransaction().commit();
manager.close();
fabrik.close();
```

Diese Struktur ähnelt zwar noch immer dem vorher gezeigten JDBC-Code, in dem vor- sowie nachbereitende Arbeit für eine Transaktion implementiert werden muss, allerdings ist dieser Code nun variabel und besser skalierbar. Für das automatische verwalten von Persistenzmanagern, um das ständige starten und beenden nicht mehr von Hand implementieren zu müssen, können „Dependency-Injection-Frameworks“ wie CDI, EJB, Spring oder ähnliche, Abhilfe leisten.

Der im Codefragment dargestellte Vorgang, kann auch als create bzw. speichern eines Datensatzes bezeichnet werden. Dabei wird eine neue Entität „Aktie“ instanziiert und passende Values über Setter hinzugefügt. Die Entität „Aktie“ wird durch die Methode persist() in einen durch den Entity Manager verwalteten, persistenten Zustand gebracht. Durch getTransaction().commit() wird die persistente Entität in die Datenbank geschrieben und die Transaktion beendet. Abschließend wird der Entity Manager durch die Close-Methode beendet.

5.2.2 Persistenzkontext

Ist eine Entität einem Manager zugeordnet, werden sämtliche Änderungen an der Entität und die Synchronisierung mit der Datenbank durch den Entity Manager überwacht. [18] Der Bereich des vorher beschriebene Zustands durch die Methode

[17] Vgl. Keith M. & Schincariol M., Pro JPA 2, Second Edition, Seite 19

[18] Vgl. IBM Infocenter: http://pic.dhe.ibm.com/infocenter/radhelp/v7r5/index.jsp?topic=%2Fcom.ibm.jpa.doc%2Ftopics%2Fc_entity_manager.html, Zugriff: 01.05.2014

persist(), in dem Entitäten von einem Entity Manager verwaltet und persistent gehalten werden, wird auch als Persistenzkontext (englisch „Persistence Context“) bezeichnet. Jeder Zustandsmanager besitzt so einen Persistenzkontext, sowie jede Entität, die einen persistenten Zustand besitzt und von einem Zustandsmanager verwaltet wird, in diesem Kontext hinterlegt ist. [19] In Punkt 5.2.3 gehen wir detaillierter darauf ein, wann bzw. wie eine Entität in den Zustand gebracht werden kann, damit diese von dem Entity Manager verwaltet wird.

5.2.3 Lebenszyklus der Entitäten – Zustände und Manipulationen

Die bereits erwähnte persist()-Methode gehört zu einem Methodensatz des Entity Managers, welche die Entitäten manipuliert und ihnen einen persistenten Lebenszyklus verschafft. [20] Dieser Lebenszyklus lässt sich wie folgt darstellen:

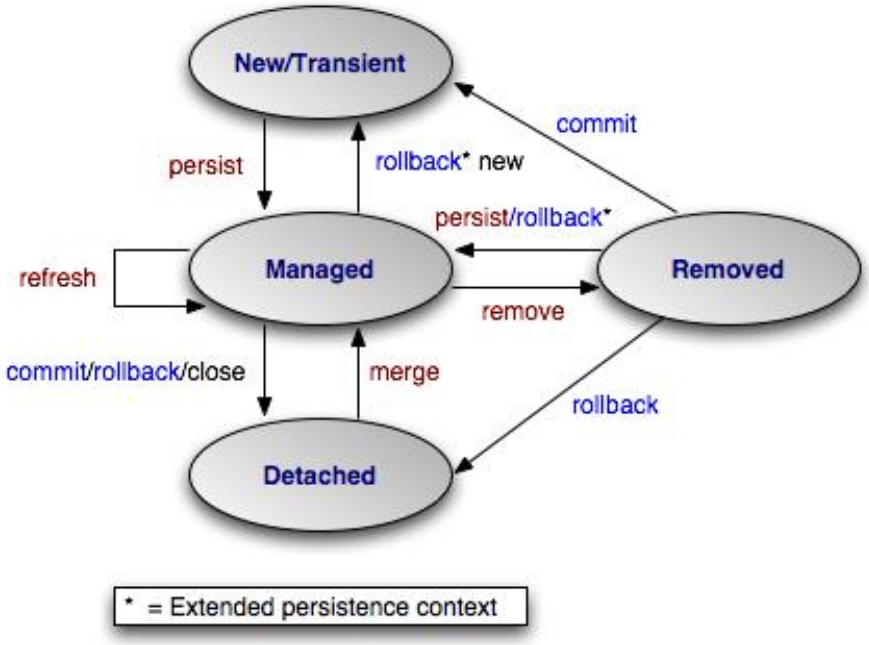

Abbildung 4: Lebenszyklus Management einer Entität, Quelle:

http://www.ibmpressbooks.com/articles/article.asp?p=1192350&seqNum=3, Zugriff 30.04.2014

Durch diese Darstellung des in Abbildung 4 gezeigten Lebenszyklus, werden der Zustand eines POJOs, sowie die logischen Möglichkeiten eines Entity Managers um Zustände von Objekte zu verändern dargestellt. Die vier Objektzustände können wie folgt beschrieben werden:

- New/Transient: Das Objekt ist bereits instanziiert, aber noch nicht in eine Datenbank gespeichert.
- Managed: Das POJO wird nun vom Entitäten Manager verwaltet, ist persistent zu der Datenbank und jede Änderung z. B. über Getter/Setter impliziert eine Änderung in der Datenbank.
- Detached: Der Zustandsmanager ist geschlossen und das Objekt noch vorhanden, aber Änderungen haben keine Auswirkung auf die Datenbank.

[19] Vgl. Bauer C. & King G., „Java Persistence with Hibernate“, Manning, 2007, Seite 384

[20] Vgl. Keith M. & Schincariol M., Pro JPA 2, Second Edition, Seite 117

- Removed: Die Datenbank ist von der Objektinstanz bereinigt und das POJO ist bis zu einer weiteren Aktion ausgeblendet für den Entity Manager.

21

Die Entitäten manipulierenden beziehungsweise zustandsverändernden Operationen können unter JPA in folgende, gängige Methoden unterschieden werden:

5.2.3.1 Persist(), Contains() und Merge()

Der bereits unter 5.2.1 gezeigte Code, verdeutlicht wie einfach ein neues POJO durch die Methode Persist() in den Managed-Zustand gebracht werden kann. Bei dem Auftreten eines Problems, wird eine Exception (Persistence Exception) geworfen und der Zustand wieder zurückgesetzt beziehungsweise nicht verändert. [22] Dieses Vorgehen ähnelt dem ACID-Prinzip der Datenbank, dass Operationen nur ganz oder gar nicht ausgeführt werden. Um festzustellen, dass ein Objekt vom Persistenzmanager verwaltet wird, kann die Methode Contains() angewandt werden. [23] Diese checkt, ob eine Entität verwaltet wird, allerdings ist dies nur selten notwendig.

Nun kann es vorkommen, dass sich das POJO im detached-Zustand befindet, bereits eine Instanz in der Datenbank vorhanden ist und dieses nun doch wieder für Updates persistent gemacht werden muss. Die Merge()-Methode funktioniert dabei als Updatefunktion und führt, wie der Name bereits sagt, das Objekt mit dem bereits vorhandenen Eintrag, wieder zusammen. Merge() funktioniert vom Verhalten so ähnlich wie die Methode Persist(), jedoch mit dem Unterschied, dass keine neue Instanz erzeugt wird, sondern eine Kopie, die von dem Zustandsmanager verwalten wird. [24]

```
//Der Entity Manager („manager“) gleicht das Objekt „aktie“ ab und macht es wieder
//persistent. Im Anschluss erfolgt die Änderung, was auch zur Änderung in der
//Datenbank führt.
public void updateShare(Aktie aktie) {
  manager.merge(aktie);
  aktie.setUnternehmenName(neuerName);
}
```

Diese für die Entität „Aktie“ gekapselte Updatefunktion „updateShare“, geht wie beschrieben vor, dass erst eine Persistenz des parametrisierten Objekts erfolgt, da sich seit dem letzten Schreiben in die Datenbank die Daten der Entität oder der Datenbankinstanz verändert haben könnten und im Anschluss die zu updatenden Werte verändert werden können.

5.2.3.2 Refresh(), Find() und Remove():

Refresh ist, wie der Abbildung 4 zu entnehmen ist, für das Aktualisieren des gleichen Managed-Objektes gedacht. Hierbei wird der aktuelle Stand einer abgefragten Instanz aus der Datenbank entnommen und das Objekt aktualisiert. [25]

[21] Vgl. http://www.ibmpressbooks.com/articles/article.asp?p=1192350&seqNum=3, Zugriff: 30.04.2014

[22] Vgl. Keith M. & Schincariol M., Pro JPA 2, Second Edition, Seite 21

[23] Vgl. Keith M. & Schincariol M., Pro JPA 2, Second Edition, Seite 138

[24] Vgl. Keith M. & Schincariol M., Pro JPA 2, Second Edition, Seite 148

[25] Vgl. Keith M. & Schincariol M., Pro JPA 2, Second Edition, Seite 357

```
//Der Entity Manager („manager“) aktualisiert das Objekt „aktie“ mit dem aktuellen
//Stand der Datenbank
private Aktie refreshShare(Aktie aktie) {
  manager.refresh(aktie);
  return aktie;
}
```

Eine solche „refreshShare“ Methode ist unabdingbar, wenn eine Entität im Persistenzkontext unverändert geblieben ist, aber in der Datenbank eventuell eine Änderung vorgenommen wurde. [26] Sollte nach einem Datenbankeintrag gesucht werden, der keiner persistenten Entität entspricht und nur ein einfacher Lesezugriff erfolgen, dann steht für diesen Fall die Funktion find() zur Verfügung.

```
//Findet eine Datenbankinstanz anhand der mitgelieferten isin und gibt das passende
//Objekt zurück
private Aktie findShare(String isin) {
  return manager.find(Aktie.class, isin);
}

//Findet alle Datenbankinstanzen, die in der Query abgefragt werden (In diesem Fall
//alle der Tabelle Aktie) und gibt diese in einer Liste (Collection) zurück
Private Collection<Aktie> findAllShares() {
  Query query = manager.createQuery("SELECT e FROM Aktie e", Aktie.class);
  return (Collection<Aktie>) query.getResultList();
}
```

Während findShare ein ganz bestimmtes Objekt von der Entität „Aktie“ sucht, ist die zweite Methode „findAllShares“ durch einen implementierten SELECT so ausgelegt, dass sie alle Ergebnisse der Tabelle Aktie als Collection zurück liefert. Da es keine explizite Funktion des Zustandmanagers gibt, in der alle Datensätze einer Tabelle wieder gegeben werden, muss der Umweg über die Methode createQuery() gegangen werden. Durch diese lässt sich mittels Persistence Query Language (JPQL) eine Abfrage bilden. Neben dieser Methode, gibt es noch die createNamedQuery() und die createNativeQuery(), welche eine Instanz einer vordefinierten Abfrage erstellt beziehungsweise im letzteren Fall eine Abfrage mittels nativen SQL erstellen lässt. [27]

Die find()-Methode wird neben dem weiterverarbeiten von Datenbankeinträgen, auch häufig für das Löschen eines Eintrags aus der Datenbank verwendet.

```
//Löscht eine Datenbankinstanz anhand der vorher gefundenen Datenbankinstanz, wenn
//diese gefunden wurde
private void removeShare(String isin) {
  Aktie aktie = findShare(isin);
  if (aktie != null){
    manager.remove(aktie);
  }
}
```

[26] Vgl. Kehle M., Hien R, Röder D., JPA mit Hibernate – Java Persistence API in der Praxis, entwickler.press, 2010, Seite 111

[27] Vgl. IBM Infocenter: http://pic.dhe.ibm.com/infocenter/radhelp/v7r5/index.jsp?topic=%2Fcom.ibm.jpa.doc%2Ftopics%2Fc_entity_manager.html, Zugriff 01.05.2014

Nach dem Laden über die find()-Methode, wird geprüft, ob die Instanz überhaupt vorhanden ist. Das Objekt erhält damit den Zustand „Removed“ und kann nur durch einen „rollback“, siehe Punkt 5.2.3.4, wiederhergestellt werden.

5.2.3.3 Flush() und Commit()

Die Funktionen flush() und commit() des Zustandmanagers führen eine Synchronisation mit der Datenbank aus. Es gibt nur zwei Situationen, in denen diese Funktionen eingesetzt werden. Zum einen, wenn eine Transaktion beendet und die Funktion commit() aufgerufen wird, werden alle vorgenommenen Änderungen dauerhaft in die Datenbank gespeichert und ist ab diesem Moment für jeden sichtbar. Zum anderen kann die Situation eintreten, dass man während einer Transaktion bereits Daten in die Datenbank übertragen möchte und so eine manuelle Zwischenspeicherung erzwingt. [28] Flush() ist die grundlegende Funktion, die bei jeder Datenbankaktualisierung und auch über den „commit“-Befehl ausgeführt wird.

```
//Das Synchronisieren des Entitätenmanagers am Ende einer Transaktion
manager.getTransaction().commit();

//Der Zustandsmanager soll seinen aktuellen Stand in die DB übertragen
manager.flush();
```

5.2.3.4 Close() und Rollback()

Jeder Zustand des in Abbildung 4 dargestellten Lebenszyklus kann rückgängig gemacht werden, solange die Transaktion noch nicht durch ein commit beendet wurde. Alle vorgenommenen Änderungen innerhalb der Transaktion werden zurückgesetzt und entsprechen genau dem Stand vor der Transaktion. [29] Bei einer fehlgeschlagenen Transaktion wird durch das ACID-Prinzip automatisch ein Rollback vorgenommen.

```
//Der Zustandsmanager führt für die Transaktion einen Rollback durch
manager.getTransaction().rollback();

//Der Zustandsmanager und die EntityFactory werden geschlossen
manager.close();
fabrik.close();
```

Die Methode close() schließt den Zustandsmanager und damit wird auch der Persistenzkontext, sowie der Lebenszyklus der Objekte beendet. Ebenfalls kann die EntityFactory geschlossen werden, dass das Ende aller erzeugten Zustandsmanager zur Folge hat.

5.2.3.5 Weitere Manipulationsmöglichkeiten durch den Zustandsmanager

Zu den gängigsten und grundlegenden Funktionen eines Enitäten-Managers kommen zusätzliche Operationsmöglichkeiten, die in spezielleren Anwendungsfällen zum Einsatz kommen können.

In JPA ist das Sperren einer bestimmten Objektinstanz über verschiedene Ansätze möglich. Es gibt für den Zustandsmanager zumindest die Möglichkeit einer manuellen Sperrung durch die Methode Lock(). Seit JPA 2.0 können ebenfalls bei den Methoden

[28] Vgl. Keith M. & Schincariol M., Pro JPA 2, Second Edition, Seite 144

[29] Vgl. Kehle M., Hien R, Röder D., JPA mit Hibernate – Java Persistence API in der Praxis, entwickler.press, 2010, Seite 142

refresh() und find(), Informationen für eine Sperrung übergeben werden. [30] Eine weitere Optionsmöglichkeit seit JPA 2.0 ist die clear()-Methode, die ähnlich wie ein „rollback“ funktioniert. Der komplette Persistenzkontext wird mit dieser Operation geleert und alle „managed“-Objekte werden in den detached-Zustand gesetzt. [31] Detach() heißt die ebenfalls hinzugekommene Methode, die im Gegensatz zu clear() auch einzelne Objektinstanzen aus dem Persistenzkontext in den Detached-Zustand bringen kann.

5.3 Konfigurationsdatei eines JPA-Projekts

Für ein lauffähiges JPA-Projekt wird immer eine Konfigurationsdatei benötigt. Diese wird von der EntityManagerFactory eingelesen und daraus kann eine Verbindung zu der Datenbank und weitere Einstellungen mittels Meta-Informationen vorgenommen werden. Bereits bei dem Anlegen eines JPA-Projekts, wird durch einer IDE wie Eclipse eine persistence.xml in einem Ordner „META-INF“ automatisch angelegt.

Die persistence.xml des beiliegenden Programms:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
             http://java.sun.com/xml/ns/persistence/persistence_2_1.xsd">
  <persistence-unit name="aktiendb1" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <class>de.fwp.entities</class>

    //weiterführende Konfigurationsinformationen von JPA und Hibernate
    <properties>
      <property name="hibernate.show_sql" value="true"/>
      <property name="hibernate.format_sql" value="true"/>
      <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect"/>
      <!-- <property name="hibernate.hbm2ddl.auto" value="validate"/> -->
      <property name="javax.persistence.schema-generation.database.action"
                value="drop-and-create"/>

      //Die Datenbankkonfiguration
      <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
      <property name="javax.persistence.jdbc.url"
                value="jdbc:mysql://127.0.0.1:3306/dbName"/>
      <property name="javax.persistence.jdbc.user" value="root"/>
      <property name="javax.persistence.jdbc.password" value=""/>
    </properties>
  </persistence-unit>
</persistence>
```

Während die Kopfzeilen spezifizieren, dass es sich um die Version 2.1 von JPA und es sich um die persistence.xml handelt, wird direkt danach die persistence-unit namentlich festgelegt, auf die sich die EntityManagerFactory bezieht. Außerdem wird festgelegt, welcher Pfad zu den implementierten Entitäten führt. Nachfolgend können Framework- und JPA-spezifische Informationen hinterlegt werden. In diesem Fall ist Hibernate angewiesen, alle auszuführenden SQL-Statements in einem bestimmten Datenbankdialekt zu formulieren und diese formatiert auszugeben. (Im anhängenden Code wurde ebenfalls auf Basis des von MySQL existierenden „MySQLDialect“ getestet.) Eine bereits interessante Neuerung in JPA 2.1 ist ein eigenes Vorgehen bei dem Generieren von Datenschemen. Dies macht die

[30] Vgl. Keith M. & Schincariol M., Pro JPA 2, Second Edition, Seite 111f

[31] Vgl. Keith M. & Schincariol M., Pro JPA 2, Second Edition, Seite 143, 146

Konfiguration um einen Schritt unabhängiger von den Frameworks wie Hibernate. Durch die Option „drop-and-create“ wird festgelegt, dass bei jedem Start das aktuelle Datenschema gelöscht und aus den bestehenden Entitäten neu erstellt wird. Anschließend muss noch eine Datenbankkonfiguration, bestehend aus Treiber, Url und Login-Daten, hinterlegt werden.

Mit dem aktuellen Stand sollte nun eine einfache Implementierung ohne dem Anstreben eines komplexen Datenbankschemas möglich sein. In folgenden Punkten gehen wir auf weiterführende Themen ein, um auch anspruchsvollere Situationen umsetzen zu können.

5.4 Das Mapping von Tabellenbeziehungen und Vererben von Entitäten

Heutzutage ist das Speichern aller zusammenhängenden Datensätze in eine einzige Tabelle aus mehreren Gesichtspunkten keine Lösung. Es wird daher bei relationalen Ansätzen immer mit Tabellensplittungen und Beziehungen zwischen den Tabellen gearbeitet. Bisher wurden Entitäten, der Vereinfachung halber, ausschließlich ohne Beziehungen behandelt, obwohl diese sehr häufig auftreten. Grundsätzlich lassen sich die am häufigsten auftretenden Beziehungstypen in folgende Kardinalitäten aufteilen:

- Eins-zu-eins-Beziehung (1:1)
- Eins-zu-viele-Beziehung (1:n)
- Viele-zu-eins-Beziehung (n:1)
- Viele-zu-viele-Beziehung (n:m)

Zusätzlich ist es in der Objektorientierung auch üblich, dass ein Javaobjekt von einem anderen erbt und von mehreren Entitäten, Informationen in einer Superklasse hinterlegt sind. Diese Beziehungen unter Entitäten erfordern weitere Konfigurationen.

Im Folgenden werden die Implementierungsmöglichkeiten von Tabellenbeziehungen mittels JPA aufgezeigt und wie mit Vererbungen zwischen Entitäten umgegangen werden kann.

5.4.1 Das Abbilden von Tabellenbeziehungen

Die einfachste ist die 1:1-Beziehung, bei der eine Entität auch nur mit einer anderen in Verbindung steht, so dass der Primärschlüssel der einen Entität einmalig in der anderen Tabelle vorkommt.

```
@Entity
public class Unternehmen implements Serializable {

  @Id
  private String unternehmenName;

  @OneToOne
  @JoinColumn(name="Sitz")
  private UnternhemenSitz stadt;
}
```

```
@Entity
public class Firmensitz implements Serializable {

  @Id
  private String stadt;
  private String land;
}
```

In diesem Beispiel wird in nur eine Richtung die ID („stadt“) der Tabelle Firmensitz, in der Tabelle Unternehmen aufgeführt, so dass eine unidirektionale 1:1 Beziehung entsteht. Unidirektional ist eine Beziehung, solange sie wie in diesem Beispiel nur in eine Richtung verläuft. Über die Tabelle Firmensitz alleine, lässt sich demnach nicht abfragen, welche Unternehmen in welcher Stadt ihren Sitz haben. Die zweite Variante, die eine solche Abfrage ermöglicht, ist die bidirektionale Verbindung und unterscheidet sich in der Implementierung dadurch, dass in beiden Klassen die Annotation @OneToOne steht. [32]

Bei einer 1:n-Beziehung muss auf die unterschiedlichen Seiten der Beziehung geachtet werden. Während die eine Entität mehrere Instanzen einer anderen Entität verbindet, hat in umgekehrter Richtung jede Entitäten-Instanz nur maximal ein Gegenstück. Wenn beide Richtungen implementiert wurden, wie nachfolgend zu sehen ist, wird die Verbindung wieder als bidirektional bezeichnet.

```
@Entity
public class Aktie implements Serializable {

  @ManyToOne
  @JoinColumn(name = "unternehmenName")
  private Unternehmen unternehmen;
}
```

```
@Entity
public class Unternehmen implements Serializable {

  @OneToMany(mappedBy = "unternehmenName")
  private Collection<Aktie> aktieCollection;
}
```

Durch die Annotation @OneToMany und dem Zusatz „mappedBy“, wird eine Beziehung zu der Entität „Aktie“ aufgebaut, die besagt, dass ein Unternehmen verschiedene Aktien besitzen kann. In die andere Richtung stellt die Annotation @ManyToOne eine Beziehung her, durch die jede Entität „Aktie“ nur eine Entität „Unternehmen“ zugewiesen bekommt.

Die letzte und komplexeste Beziehung ist eine Many-to-Many (@ManyToMany) oder auch n:m Beziehung. Sie kann ebenfalls unidirektional, sowie bidirektional sein und erfordert eine Hilfstabelle um die Entitäten verbinden zu können. [33]

[32] Vgl. Müller & Wehr, Java Persistence Api 2, Hanser, Seite 90f

[33] Vgl. Müller & Wehr, Java Persistence Api 2, Hanser, Seite 107

```
@Entity
public class Aktie implements Serializable {

  @Id
  private String aktien_isin;

  @ManyToMany(mappedBy="aktien")
  private List<Indizes> index;
}
```

```
@Entity
public class Indizes implements Serializable {

  @Id
  private String indizes_isin;

  @ManyToMany
  @JoinTable( name="aktien_in_indizes",
              joinColumns={@JoinColumn(name="indizes_isin")},
              inverseJoinColumns={@JoinColumn(name="aktien_isin")}
            )
  private List<Aktie> aktien;
}
```

In diesem Beispiel wird jeweils die „ISIN“ aus den Entitäten Aktie und Indizes in einer Hilfstabelle „indizes_isin“ verlinkt. Dies geschieht vor allem durch die Annotation @JoinTable.

5.4.2 Lademethoden

Vor allem bei Beziehungstypen kommt schnell die Frage auf, ob man Datensätze während eines Ladevorgangs immer komplett in den Speicher laden möchte. Situationsbezogen muss entschieden werden, was besser für die Performance wäre. Mit der Eigenschaft „fetch“ und den beiden Verhalten „EAGER“ und „LAZY“ ist leicht eine Datenzugriffsstrategie aufzubauen. [34]

```
  @ManyToMany(fetch = FetchType.LAZY)
  private List<Indizes> index;
```

Lazy Loading, das im Gegensatz zu EAGER Loading Datensätze nicht sofort mitlädt, sondern erst wenn sie benötigt werden, ist nur ungenau spezifiert worden, da die Expertengruppe sich nicht einig geworden ist [35] und zusätzlich kann es zu hässlichen Exceptions während dem Anwendungsbetrieb kommen, wenn zum Beispiel eine Entität aus dem Persistenzkontext heraus genommen wird, aber Datensätze noch nicht vollständig geladen wurden.

5.4.3 Transitive Persistenz

Wenn wir nun wieder auf den Lebenszyklus zurückkommen und uns die möglichen Zustandsveränderungen einzelner Entitäten noch einmal vor Augen halten, können

[34] Vgl. Kehle M., Hien R, Röder D., JPA mit Hibernate – Java Persistence API in der Praxis, entwickler.press, 2010, Seite 187

[35] Vgl. Kehle M., Hien R, Röder D., JPA mit Hibernate – Java Persistence API in der Praxis, entwickler.press, 2010, Seite 187

einige Fragen aufkommen. Zum Beispiel wie mit dem Umstand in JPA umgegangen werden kann, wenn eine in Beziehung stehende Entität persistent wird, während die andere es nicht ist oder gelöscht wurde. Die Lösung heißt Transitive Persistenz und erlaubt bei entsprechender Implementierung die Weitergabe von Operationen an die in Beziehung stehenden Entitäten. [36] Über CascadeTypes kann ein Verhalten für das Weitergeben von Operationen festgelegt werden.

```
@ManyToMany(cascade = CascadeType.ALL)
```

Mit CascadeType.ALL werden alle Operationen („persist“, „merge“, „remove“, „refresh“) durchgereicht. Es können aber auch einzelne Operationen zum durchreichen festgelegt beziehungsweise aufaddiert werden, wie nachfolgendes Beispiel zeigt:

```
@ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
```

5.4.4 Vererbungen

Der Beispielcode des beiliegenden Programms enthält keine Vererbungen, aber die Möglichkeit besteht schon von Beginn an durch Frameworks und das spätere Einfließen dieser Funktion in die JPA-Spezifikation. Dabei wird zwischen den folgenden drei Vererbungsstrategien unterschieden.

- SINGLE_TABLE
- JOINED
- TABLE_PER_CLASS

Während SINGLE_TABLE eine Tabelle für die komplette Vererbungshierarchie bereitstellt, wird durch TABLE_PER_CLASS nur für jede konkrete und durch JOINED für jede konkrete und abstrakte Klasse eine Tabelle erzeugt. [37]

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE
public abstract class AbstractWertpapiere implements Serializable {...}

@Entity
@DiscriminatorValue(value = "Anleihen")
public class Anleihen extends AbstractWertpapiere {...)
```

Die im Beispiel gezeigten Annotationen @Inheritance definiert, wie die Vererbungshierarchie abgebildet werden soll. [38] Durch diese und der Annotation @DiscriminatorValue wird eine große, zusammengeführte Tabelle erzeugt. Durch Vererbung kann sogar der Primärschlüssel, der eigentlich als fester Bestandteil einer Entität gilt, in eine Superklasse ausgelagert werden. [39]

---

[36] Vgl. Kehle M., Hien R, Röder D., JPA mit Hibernate – Java Persistence API in der Praxis, entwickler.press, 2010, Seite 88

[37] Vgl. Kehle M., Hien R, Röder D., JPA mit Hibernate – Java Persistence API in der Praxis, entwickler.press, 2010, Seite 92

[38] Vgl. Keith M. & Schincariol M., Pro JPA 2, Second Edition, Seite 297

[39] Vgl. Kehle M., Hien R, Röder D., JPA mit Hibernate – Java Persistence API in der Praxis, entwickler.press, 2010, Seite 65

6. JPA 2.1

In der neusten Spezifikation von JPA sind wieder neue Funktionen eingeflossen, die meistens schon in gleicher oder ähnlicher Form von Frameworks bereitgestellt wurden. Eine Funktion, die in die persistence.xml einfließen kann, wurde bereits unter dem Punkt 5.3 erwähnt. Dank JPA 2.1 lässt sich das automatische Löschen und Generieren beziehungsweise Updaten von Datenbankschemen ähnlich leicht und unabhängig von Frameworks implementieren. Weitere Neuerungen sind unter anderem ein Konverter. Durch einen Konverter können Daten aus fachlicher Sicht, in beide Richtungen beeinflusst werden. [40] In Richtung Datenbank lassen sich dadurch zum Beispiel boolean-Werte von „0“ und „1“ in „t“ und „f“, oder ähnlichem umwandeln. Aus der Datenbank heraus, können Daten ebenfalls manipuliert werden, so dass zum Beispiel optimierende Maßnahmen, wie das Umwandeln gekürzter, abgespeicherter Zeichenketten bei jeder Abfrage auf Applikationsseite zu einer Performanceoptimierung führen kann. Für die Realisierung muss die Annotation @Converter an die Konverter-Methode gehängt und zusätzlich ein AttributConverter implementiert werden. Weitere Möglichkeiten, wie unter anderem die Annotation @Convert(autoApply=true), lässt ihn zu einem Default-Konverter werden, der automatisch bestimmte Typen umwandelt, wenn diese in die Datenbank wandern oder aus dieser kommen. [41]

7. Zusammenfassung und Fazit

JPA kommt in der aktuellen Spezifikation mit so vielen Optionen und Konfigurationsmöglichkeiten, dass nur die Grundlagen und ein Bruchteil davon in dieser Arbeit behandelt werden konnten. Mittlerweile ist JPA auch so umfangreich, dass auf Funktionen von Frameworks großteils verzichtet werden kann. Für die Anwendung ist, wie wir gesehen haben, einiges an Vorwissen notwendig, um überhaupt damit arbeiten zu können. Bei steigender Komplexität wie beispielsweise einer verteilten Anwendung wird ein gewisses Know-how abverlangt. Es gelingt anfangs meist trotz vieler Beispiele und Tutorials nicht, ohne Fehlermeldungen ein etwas komplexeres Datenbankschema mit Beziehungen aufzubauen. Da versteht sich von Alleine, dass bei manchen traditionellen JDBC-Entwicklern, JPA als keine passende Alternative gilt. Allerdings wenn man erst einmal weg von JDBC und dem implementieren nativer SQL-Passagen kommt und die Grundlagen von der Java Persistence API verstanden hat, ist es ein sehr hilfreiches Werkzeug. Bei einem Werkzeug bleibt es aber dann auch, denn es ist kein Allheilmittel. Vielmehr muss vor einem Projekt die Anforderung abgeschätzt und anhand dieser die richtige Wahl getroffen werden. Beispielsweise macht JPA keinen Sinn, wenn ein einfacher und statischer Datenbanktransfer umgesetzt werden muss, da nicht nur die Komplexität für diese Aufgabe zu groß sein würde, sondern auch bei sehr großen Datenmengen hohe Kosten im Transfer entstehen könnten. Sollte dagegen eine beliebig skalier- und anpassbare Persistenzschicht erreicht werden wollen, kann JDBC den Tod für das Projekt bedeuten.

Gegenwärtig ist der Umweg über Technologien wie JDBC oder JPA noch kaum vermeidbar und am häufigsten anzutreffen. Allerdings lässt sich auch ein Trend

---

[40] Vgl. http://www.gedoplan.de/sites/default/files/pdfs/kundenzeitschrift/online_2013_herbst.pdf, Seite 4, Zugriff: 01.05.2013

[41] Vgl. http://www.gedoplan.de/sites/default/files/pdfs/kundenzeitschrift/online_2013_herbst.pdf, Seite 4, Zugriff: 01.05.2013

feststellen, der versucht ist, objektorientierte Datenbanken in bestimmten Anwendungsfällen zum Einsatz zu bringen. Da dies allerdings neben dem Abwiegen von Vor- und Nachteilen, auch die der Akzeptanz der Unternehmen voraussetzt, bleibt abzuwarten ob diese Art der Datenbankmanagementsysteme eine gewisse Relevanz erreichen und sich damit eine aus Entwicklersicht eventuell einfachere Implementierungsmöglichkeit als über die Java Persistence API etablieren wird.

8. Hinweis zu der beiliegenden Anwendung

Die Anwendung wurde unter Java 7 als Maven-Projekt in Eclipse Kepler erstellt und setzt auf eine MySQL Datenbank auf. Für eine lokale Datenbankumgebung kamen XAMPP und MySQL Workbench sowie der DBVisualizer zum Einsatz. Externe Libraries wurden neben dem Framework Hibernate nur ein mysql-connector, das Logging-Tool slf4j und das Testing-Tool jUnit importiert. Für das testweise Erstellen von Datenbankschemen und –einträgen steht ein Unittest bereit, der aus einer konsolidierten Beispieltabelle Datensätze holt und diese über eine Datensenke in das neue Schema überführt.

9. Literaturverzeichnis

[1] Backschat Martin, Rücker Bernd, Enterprise JavaBeans 3.0, Elsevier GmbH, 2007, Seite 113

[2] Bauer C. & King G., „Java Persistence with Hibernate“, Manning, 2007, Seite 5-384

[3] http://www.gedoplan.de/sites/default/files/pdfs/kundenzeitschrift/online_2013_herbst.pdf, Seite 4, Zugriff: 01.05.2013

[4] George Reese, Database Programming with JDBC and Java, O’Reilly & Associates Inc., 2 Edition, August 2000, Seite 10

[5] http://www.hs-owl.de/fb5/labor/it/de/sd/uebung/jdbc.pdf, Seite 1, Zugriff 28.04.2013

[6] IBM Infocenter: http://pic.dhe.ibm.com/infocenter/radhelp/v7r5/index.jsp?topic=%2Fcom.ibm.jpa.doc%2Ftopics%2Fc_entity_manager.html, Zugriff 01.05.2014

[7] Kehle M., Hien R, Röder D., JPA mit Hibernate – Java Persistence API in der Praxis, entwickler.press, 2010, Seite 21-187

[8] Keith M. & Schincariol M., Pro JPA 2, Second Edition, Seite 8-357

[9] Müller & Wehr, Java Persistence Api 2, Hanser, Seite 90-107

[10] Ullenboom C, „Java 7 – Mehr als eine Insel“, http://openbook.galileocomputing.de/java7/1507_16_013.html, Zugriff: 30.04.2014

10. Abbildungsverzeichnis

Versicherung

„Ich versichere, dass ich die vorstehende Arbeit selbständig angefertigt und mich fremder Hilfe nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß veröffentlichtem oder nicht veröffentlichtem Schrifttum entnommen sind, habe ich als solche kenntlich gemacht.“

_____