1 Einleitung

In jedem Softwareprojekt muss man sich über die Speicherung der Daten Gedanken machen. Für Konfigurationen und wenige Daten reicht es meistens aus, diese in einfachen Property oder XML-Dateien abzulegen. Jedoch kommt man bei größeren Datenmengen meistens nicht an der Verwendung einer Datenbank vorbei. Mit Java gibt es verschiedene Ansätze Daten in einer Datenbank abzulegen. Dabei ist die Verwendung von JDBC die Basis von vielen Möglichkeiten, da damit SQL Statements direkt an die Datenbank geschickt werden können.

Um aber nicht in jedem Projekt das Rad bzw. die Persistenz der Daten neu erfinden zu müssen, gibt es objektrelationale Mapper. Mit der Java Persistence API (JPA) wurde eine einheitliche Schnittstelle für die Persistenz in Java spezifiziert, die mittlerweile von zahlreichen Frameworks, beispielsweise Hibernate, unterstützt wird. In den nachfolgenden Kapiteln soll sowohl ein allgemeiner Überblick über objektrelationale Mapper (auch als "O/R-Mapper" bezeichnet) gegeben als auch im Spezielle JPA und Hibernate vorgestellt werden. [2]

2 Unvereinbarkeit der Paradigmen bzw. Impedance Mismatch

In der Softwareentwicklung ist mit "Impedance Mismatch" der Unterschied in der Struktur zwischen normalisierten relationalen Datenbanken und objektorientierten Klassenhierarchien gemeint. Diese Unterschiede werden in den folgenden Unterkapiteln genauer beschrieben. [3]

2.1 Das Problem der Granularität

Ein objektorientiertes Modell ist typischerweise sehr feingranular, so kann es beispielsweise eine Entity Person geben, die eine Entity Adresse als Attribut hat. [2]

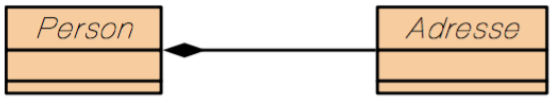

Abbildung 1: Komposition zwischen Person und Adresse [2]

Objekte können jegliche Granularität haben, Tabellen hingegen sind bezüglich der Granularität beschränkt. In einer relationalen Datenbank sind die Daten der Person inklusive den Adressdaten normalerweise in einer Tabelle abgelegt [2, 3]

ID	Vorname	Nachname	...	Adresse_PLZ	Adresse_Stadt	...
1	Max	Mustermann	...	01234	Musterstadt	...
2	...	...	...	...	...	...

Abbildung 2: Abbildung der Komposition in der Datenbank [2]

Mit welchen Mitteln man ein feingranulares Objektmodell in zweidimensionalen Tabellen abbilden kann, wird in Kapitel "0

Komponenten" gezeigt.

2.2 Das Problem der Subtypen

Vererbung ist in Programmiersprachen wie Java selbstverständlich. Relationale Datenbanken kennen aber keine Vererbung. In Kapitel "7 Inheritance Mapping" wird gezeigt, welche Strategien es zur Abbildung von Vererbungshierarchien gibt und welche Vor- und Nachteile die jeweilige mit sich bringt.

2.3 Das Problem der Identität

In Java sind zwei Objekte identisch, wenn beide dasselbe Objekt sind. Wenn zwei Objekte identische Werte enthalten, dann sind die Objekte gleich, aber nicht unbedingt identisch. Objektidentität wird in Java mit dem == Operator überprüft, Objektgleichheit mit "equals()". [3]

Objektidentität in Java:

```
objektA == objektB;
```

Objektgleichheit in Java:

```
objektA.equals(objektB);
```

In relationalen Datenbanken wird ein Eintrag in einer Tabelle über die Daten, die er enthält, identifiziert; damit können gleiche Datensätze gefunden werden. Allerdings kann man nicht sicherstellen, dass diese identisch sind. Um nun für ein Objekt den entsprechenden identischen Eintrag in der Datenbank zu finden, muss ein eindeutiger Primärschlüssel eingeführt werden. In den Objekten wird dieser Primärschlüssel ebenso eingefügt und somit kann über diesen die Identität zwischen Objekt und Eintrag in der Datenbank gewährleistet werden. [2]

Auf diese Thematik wird in Kapitel "0

Datenbankidentität, Objektidentität und -gleichheit" genauer eingegangen.

2.4 Mit Beziehungen (Assoziationen) verbundene Probleme

Beziehungen gibt es auch in relationalen Datenbanken. Mit einem Fremdschlüssel in der einen Tabelle wird ein Primärschlüssel in einer anderen Tabelle referenziert und somit die Tabellen in Beziehung gebracht. In der objektorientierten Welt gibt es mehrere Arten von Beziehungen:

- 1-zu-1-,
- 1-zu-viele-,
- viele-zu-1- und
- viele-zu-viele-Beziehungen.

Diese können letztendlich alle mit Fremdschlüsseln abgebildet werden. Etwas komplizierter ist die 1-zu-viele-Beziehung, da dort ein Primärschlüssel einen Fremdschlüssel referenziert und bei der viele-zu-viele-Beziehung muss eine Beziehungstabelle (Join-Tabelle) eingeführt werden. Die Beziehungstabelle enthält zwei Fremdschlüssel, die jeweils auf eine Seite der Beziehung zeigen. In der viele-zu-viele-Beziehung (Abbildung siehe unten) kann ein Student mehrere Dozenten (oder Professoren) haben und ein Dozent kann ebenso mehrere Studenten haben. Beziehungen werden in Kapitel "0

Assoziationen (Beziehungen)" behandelt. [2, 3]

2.5 Das Problem der Graphennavigation (Datennavigation)

Über Objekte mit Java zu navigieren ist sehr leicht. Wenn beispielsweise auf alle Vorlesungen eines Dozenten zugriffen werden soll, so wird einfach

```
dozent.getVorlesungen();
```

aufgerufen. Zur Datenbank können bis dahin bereits zwei Zugriffe erfolgt sein. Einer für die Abfrage des Dozenten und ein weiterer für die Vorlesungen des Dozenten. Als Alternative kann ein SQL-Join verwendet werden:

```
select * from DOZENT left outer join VORLESUNG where ...
```

Alternativ dazu könnte die Datenbankabfrage auch erst zu einem späteren Zeitpunkt erfolgen. Dieses Thema wird in Kapitel "5.4 Transitive Persistenz bzw. "cascade"" genauer beschrieben. [2]

2.6 Das Problem der unterschiedlichen Typen

Oft unterscheiden sich die verfügbaren Datentypen einer Programmiersprache von denen in einer relationalen Datenbank. Aus diesem Grund muss bei manchen Datentypen eine Umwandlung zwischen dem Datentyp im Code und dem Datentyp in der Datenbank durchgeführt werden.

O/R-Mapper können die meisten Datentypen automatisch zu einem in der Datenbank passenden Datentyp in beide Richtungen umwandeln. Sollte dies nicht möglich sein, kommt es zu einer Exception. In diesem Fall kann meist auch ein manuelles Mapping konfiguriert werden.

3 Primärschlüssel

3.1 Anforderungen und Definition

Für den Primärschlüssel sollen folgende Anforderungen gelten:

- Er darf nicht null sein
- Er muss über alle Einträge in einer Tabelle eindeutig sein
- Er soll nie verändert werden

In den meisten Fällen ist es erforderlich, einen künstlichen Primärschlüssel zu den Klassen hinzuzufügen. Meist erfolgt dies über eine ID als Zahlenwert, die die O/R-Mapper automatisch hinzufügen. Zum Generieren der ID stehen oftmals verschiedene Generatoren zur Verfügung, die auch von der verwendeten Datenbank abhängig sein können. [2]

Es ist aber auch möglich, einen natürlichen (und zusammengesetzten) Primärschlüssel zu definieren. In der Praxis zeigt sich aber oft, dass diese über einen längeren Zeitraum betrachtet vor allem den letztgenannten Punkt nicht erfüllen können. [2]

3.2 Datenbankidentität, Objektidentität und -gleichheit

Objekte, die persistent in der Datenbank gespeichert sind, haben eine Datenbankidentität. Das heißt, dass zwei Objekte gleich sind, wenn sie in derselben Tabelle gespeichert sind und denselben Primärschlüssel haben. [3]

In der Welt von Java gibt es die Objektidentität und die Objektgleichheit, die mit "==" und "equals()" überprüfbar sind.

- Objektidentität: Zwei Objekte sind gleich, wenn sie dieselbe Adresse im Speicher der JVM (Java Virtual Machine) haben
- Objektgleichheit: Zwei Objekte sind gleich, wenn sie denselben Inhalt haben und somit die Methode "equals()" true liefert. Falls "equals()" nicht überschrieben wurde, wird die Objektidentität zurückgegeben.

Zur Definition der Objektgleichheit sollte "equals()" und "hashcode()" immer überschrieben werden. Für die Objektgleichheit sollten ein oder mehrere Attribute definiert werden, die ein individuelles Objekt machen. Bei einer Klasse "User" könnte dies z.B. die E-Mail-Adresse sein. [2]

4 Komponenten

Komponenten haben keine Datenbankidentität, das heißt sie verfügen über keinen Primärschlüssel. Stattdessen gehören sie zu einer Entity, und ihr Zustand wird innerhalb der Tabelle der dazugehörigen Entity gesichert.

Für die weitere Erklärung ist folgendes Beispiel gegeben:

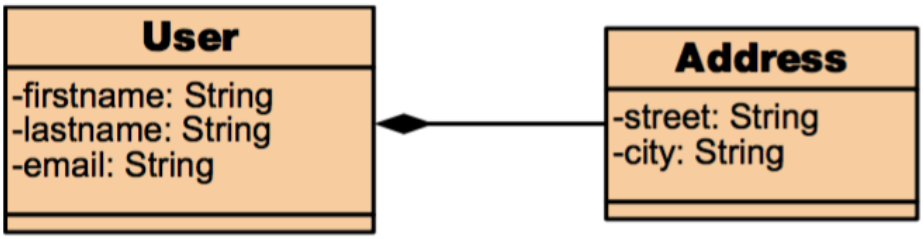

Abbildung 3: Komposition zwischen User und Address [2]

Die Klasse User enthält noch keine Informationen über die Anschrift eines Benutzers. Dazu soll es die Klasse Address geben, die als Komponente an den User angebunden werden soll, wie es in Abbildung oben zu sehen ist. Normalerweise wird eine Tabelle auf eine Klasse abgebildet. In der objektorientieren Programmierung ist es jedoch oft wünschenswert, eine Tabelle auf mehrere Klassen abzubilden, um damit ein feingranulares Klassenmodell entwerfen zu können. Komponenten ermöglichen die Abbildung mehrerer Klassen auf eine Tabelle. [2, 3]

Ein Datensatz in der Tabelle könnte beispielsweise folgendermaßen aussehen:

Id	Firstname	Lastname	Email	Address_Street	Address_City
1	Moritz	Muster	booksonline@...	Musterstr. 11	Musterhausen

Abbildung 4: Abbildung der Komposition in der Datenbank [2]

5 Assoziationen (Beziehungen)

5.1 Kardinalität

Die Kardinalität beschreibt den Grad einer Beziehung zwischen zwei Entities. Es wird zwischen folgenden Kardinalitäten unterschieden:

- Eins-zu-Eins (1:1): Eine Entity steht mit einer anderen Entity in Beziehung. Für die andere Richtung gilt dasselbe.
- Eins-zu-Viele/Viele-zu-Eins (1:n/n:1): Eine Entity steht mit mehreren Entities eines Typs in Beziehung. In die andere Richtung steht eine Entity mit einer Entity des anderen Typs in Beziehung.
- Viele-zu-Viele (n:n): Wie 1:n, nur dass nun auch in die andere Richtung eine Entity mit mehreren Entities eines Typs in Beziehung steht.

5.2 unidirektional vs. bidirektional

Bei allen Assoziationen wird noch zwischen unidirektionalen und bidirektionalen Beziehungen unterschieden. Unidirektional bedeutet, dass bspw. von einer Klasse Person zur Adresse navigiert werden kann, nicht aber von Adresse zur Person.

Bei einer bidirektionalen Beziehung kann auch von der Klasse Adresse zur Person navigiert werden. Dazu wird eine zweite Referenz zurück benötigt, die allerdings nicht in der Datenbank persistiert wird, da nur eine Seite der Assoziation als Besitzer dieser Assoziation gesehen werden kann. Die andere Seite bzw. die Rückreferent muss daher im Code entsprechend gekennzeichnet werden. Außerdem muss diese Rückreferenz oftmals manuell gesetzt werden. [2]

5.3 Transitive Persistenz bzw. "cascade"

Mit transitiver Persistenz wird bei O/R-Mappern bestimmt, ob bestimmte Operationen auf ein Objekt an Beziehungsobjekte weitergegeben werden sollen. Dies betrifft vor allem das Erstellen, Updates und das Löschen von Entities. Beispielsweise ist durch ein Durchreichen eines Abspeicherns (Updates) nur mehr ein Befehl im Code notwendig. Die betroffenen Beziehungsobjekte müssen im Code entsprechend gekennzeichnet werden. [2, 3]

5.4 Fetch-Type

Mittels des Fetch-Types lässt sich bestimmen, wann beim Laden eines Objekts die dazugehörigen Beziehungsobjekte geladen werden. Dabei unterscheidet man grundsätzlich zwischen zwei Strategien: "Lazy" und "Eager". "Eager" bedeutet, dass beim Laden eines Objekts sofort die Objekte der Assoziationen mitgeladen werden. Im Falle von "Lazy" werden diese Objekte erst beim Aufruf dieser nachgeladen. "Eager" sollte daher eher bei simplen Assoziationen mit wenigen betroffenen Objekten eingesetzt werden, da sonst die Performance leidet. "Lazy" sollte eher bei komplexen Assoziationen eingesetzt werden, da es sonst zu unnötigen Verzögerungen durch das Nachladen kommt. [2]

5.5 Typen

5.5.1 1-zu-1-Beziehungen

1-zu-1-Beziehungen lassen sich auf zwei Arten umsetzen, entweder per Fremdschlüssel oder per Primärschlüssel. Im Falle einer bidirektionalen Assoziation muss eine dieser beiden Seiten als Besitzer der Assoziation gekennzeichnet werden, wobei die Seite frei gewählt werden kann.

5.5.1.1 Mapping mittels Fremdschlüssel

Beim Mapping mittels Fremdschlüssel wird mittels Fremdschlüssel auf den Primary Key der anderen Entity referenziert. Dieses Fremdschlüssel-Attribut muss allerdings eindeutig (Constraint UNIQUE) sein, da sonst öfters auf die andere Entity referenziert werden könnte. [2]

5.5.1.2 Mapping mittels Primärschlüssel

Eine "echte" 1-zu-1-Beziehung wird über dieselben Primärschlüssel erreicht. Dabei wird dem O/R-Mapper zusätzlich mitgeteilt, dass beide Entities über denselben Primärschlüssel verfügen sollen. Somit referenziert der eine Primary Key zusätzlich als Foreign Key auf den Primary Key der anderen Entity. [2]

5.5.2 1-zu-n (und n-zu-1) Beziehungen

Eine 1-zu-n Beziehung wird mittels Foreign Key auf der n-Seite zu der 1-Seite in der Datenbank realisiert. Dies bedeutet, dass die n-Seite Besitzer der Assoziation ist, somit also ein bestimmtes Attribut besitzt.

Im Falle einer bidirektionalen Beziehung besitzt die 1-Seite eine Collection. Diese wird allerdings so nicht in der Datenbank abgebildet und erfordert eine spezielle Kennzeichnung im Code. Weiters kann es erforderlich sein, dass die Rückreferenz manuell gesetzt werden muss. Dieses Setzen der Rückreferenz kann mittels folgendem Beispiel gezeigt werden: Es existiert eine 1-zu-n-Beziehung zwischen Publisher und Book. Wird nun ein neues Book zu einem Publisher hinzugefügt, wird die Rückreferenz folgendermaßen gesetzt:

```
public void addBook(Book book) {
  this.books.add(book);
  book.setPublisher(this);
}
```

Außerdem sollte man bei 1:n oder n:1 Beziehungen auf den Fetch-Type achten. Hier kann es oftmals sinnvoll sein, diesen als "Lazy" zu definieren, um ein vorzeitiges unnötiges Laden zahlreicher Objekte zu verhindern. [2]

5.5.3 N-zu-m-Beziehungen

N-zu-m-Beziehungen werden immer mit einer zusätzlichen Verbindungstabelle (Assoziationstabelle) gebildet. Diese Verbindungstabelle wird vom O/R-Mapper erstellt. Die Verbindungstabelle verfügt über Primary Keys, welche gleichzeitig Foreign Keys auf beide Enden der Assoziation darstellen. [2, 3]

Im Beispiel einer N-zu-M-Beziehung zwischen Book und BookCategroy würde die Assoziation folgendermaßen aussehen:

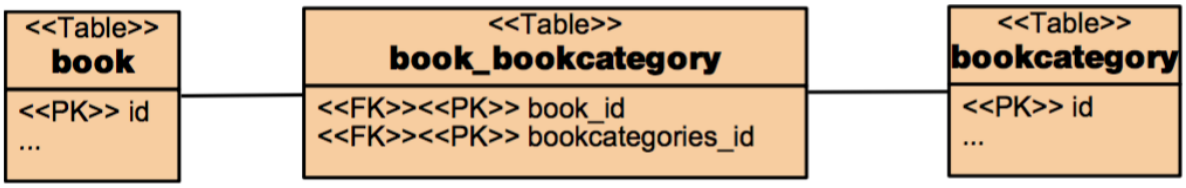

Abbildung 5: Abbildung einer N-zu-m-Beziehung in der Datenbank [2]

Bei einer bidirektionalen Beziehung muss allerdings weiterhin der Besitzer der Assoziation angegeben werden, wobei dieser im Gegensatz zur 1-zu-n-Beziehung frei gewählt werden kann. Das Setzen der Rückreferenz kann ähnlich wie im vorherigen Kapitel beschrieben erfolgen: [2]

```
public void addBookCategory(BookCategory bookCategory) {
  this.bookCategories.add(bookCategory);
  bookCategory.getBooks().add(this);
}
```

6 Collections

In den folgenden Abschnitten sollen die Möglichkeiten und Spezialitäten im Umgang mit Collections näher beleuchtet werden. Ein Teil des Themas wurde bereits im vorherigen Abschnitt betrachtet, da die *-n-Beziehungen als Collection abgebildet werden. An dieser Stelle wird allerdings vor allem auf Collections mit primitiven Datentypen eingegangen. Viele O/R-Mapper bieten die Möglichkeit, diese ohne Anlegen einer Klasse/Entity zu persistieren.

6.1 Persistente Collections

Um Collections zu persistieren, müssen diese den Typ des Interfaces besitzen (in Java z.B. nur "List" oder "Map") und keinen konkreten Typen, da die O/R-Mapper intern andere Implementierung mit Zusatzfunktionalität benötigen. Die Collection muss im Code entsprechend gekennzeichnet werden. [2, 3]

Beispiel: Eine Collection mit Kapitelüberschriften (Sectionheader) wird für jedes Buch in einem Set abgespeichert. Der O/R-Mapper legt die Daten in zwei Tabellen folgendermaßen ab:

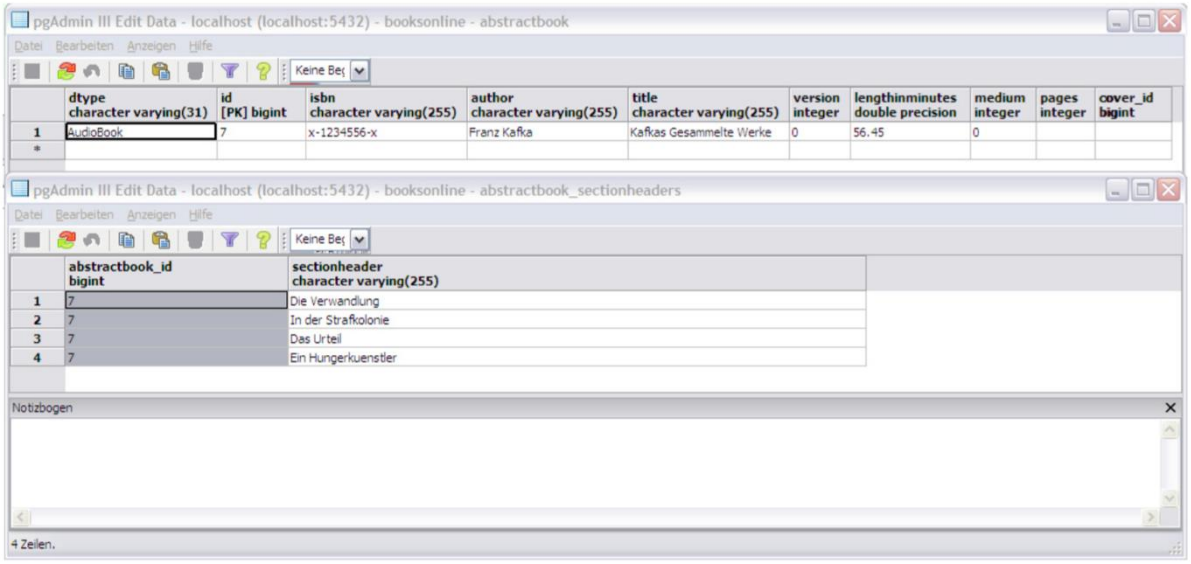

Abbildung 6: Abbildung einer Collection in der Datenbank [2]

6.2 Collections mit Index oder Schlüssel

Möchte man eine Collection oder Assoziation indizieren (Verwendung von "List") oder auf die Elemente mit einem Schlüssel zugreifen (Verwendung von "Map") müssen einige Besonderheiten beachtet werden. Hierbei geht es vor allem um die Fragen, wie die Indizes der einzelnen Listenelemente (Beibehaltung der Reihenfolge) und wie die Keys der Maps gespeichert werden. [2, 3]

6.2.1 Maps

Die bereits erwähnte 1-zu-n-Relation zwischen Publisher und Book besteht als Rückreferenz aus einer Collection. Statt einer Collection soll nun eine Map verwendet werden, wobei der Key die ID der Value (Book) darstellen soll. In diesem Fall ist keine zusätzliche Tabelle erforderlich, da die ID bereits in Book steht. Es könnte auch ein anderes Attribut als Key verwendet werden, dieses muss allerdings UNIQUE sein! [2, 3]

6.2.2 Listen

Im Falle von Listen muss bedacht werden, dass der O/R-Mapper eventuell nicht automatisch für die Beibehaltung der Reihenfolge der Listenelemente sorgt. Dies kann oftmals durch eine entsprechende Kennzeichnung im Code erreicht werden. [2]

Die Realisierung kann erfolgen, indem eine zusätzliche Spalte in die Tabelle, die für die Liste genutzt wird, eingefügt wird. In dieser Spalte wird die Reihenfolge als jeweiliger Zahlenwert abgespeichert. Betrachtet man den Screenshot im vorherigen Kapitel ("Persistente Collections"), so würde dies bedeutet, dass in der Tabelle "abstractbook_sectionheaders" diese Spalte eingefügt werden würde. [3]

6.3 Sortierte Collections

Viele O/R-Mapper unterstützen außerdem, Collections beim Abrufen automatisch zu sortieren. So könnte beispielsweise eine Liste von Namen (Strings) alphabetisch direkt beim Laden sortiert und so in der Collection bereitgestellt werden. [2]

7 Inheritance Mapping

Die einzelnen Methoden, eine Klassenhierarchie in der Datenbank abzubilden, werden an folgendem Beispiel erklärt:

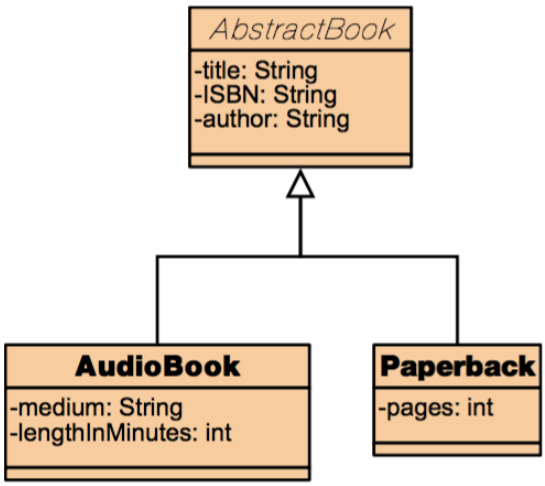

Abbildung 7: Klassenhierarchie verschiedener Bücher [2]

7.1 Eine Tabelle pro Hierarchie

Die einfachste Methode ist, alle Klassen der Hierarchie durch eine einzige Tabelle abzubilden. Damit der O/R-Mapper unterscheiden kann, auf welches Objekt ein Eintrag in der Tabelle gemappt werden muss, benötigt dieser ein zusätzliches Attribut (Discriminatorvalue). Für jede Klasse wird eine solche Discriminatorvalue bestimmt. [2, 3]

Im Buchbeispiel wird eine Tabelle AbstractBook angelegt und der Typ durch die Spalte "dtype" bestimmt:

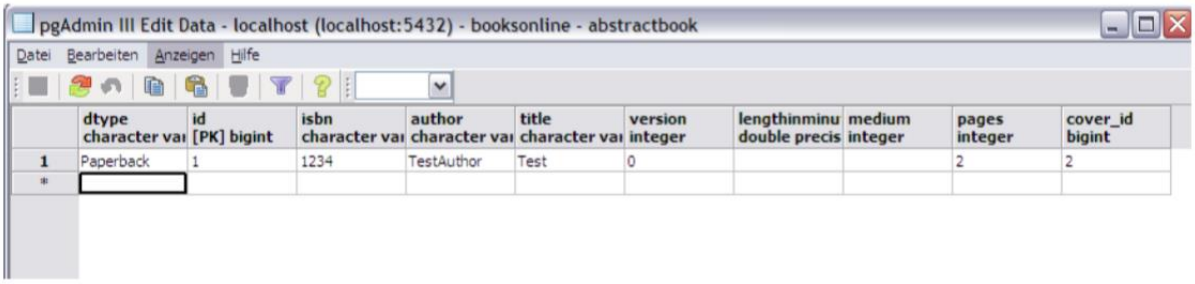

Abbildung 8: Abbildung der Hierarchie in einer einzigen Tabelle [2]

Ein Nachteil dieser Strategie sind die vielen leeren Attribute, die entstehen, da die verschiedenen Klassen über unterschiedliche Attribute verfügen. Das heißt, größere Vererbungshierarchien können schnell zu einer sehr großen Tabelle mit vielen unnötigen Spalten führen. Daraus folgt ein weiterer Nachteil, der die Datenintegrität der Datenbank betrifft. Alle Felder der abgeleiteten Klassen müssen nullable sein. [2]

Großer Vorteil dieser Vererbungsstrategie ist die Performance. Für alle Abfragen wird nur ein SELECT benötigt und kein zusätzlicher JOIN mit einer anderen Tabelle. Dazu zählen auch polymorphe Abfragen: Eine polymorphe Abfrage wäre in diesem Beispiel eine Abfrage nach allen Büchern (AudioBook & Paperback). Natürlich werden sich sehr große Tabellen auch auf die Performance auswirken. [3]

Zusammenfassend folgt daraus, dass sich diese Strategie für Vererbungshierarchien mit Subklassen, die nur wenige (unterschiedliche) Attribute haben und polymorphe Abfragen benötigen, sehr gut geeignet ist. Falls aber polymorphe Abfragen nicht benötigt werden, kann die Strategie "Eine Tabelle pro konkreter Klasse" die bessere Wahl sein.

7.2 Eine Tabelle pro konkreter Klasse

Anstatt alle Klassen einer Vererbungshierarchie in eine Tabelle zu stecken, wird bei dieser Vererbungsstrategie für jede konkrete Klasse eine Tabelle angelegt. In diesem Beispiel wird somit für "Paperback" und für "AudioBook" eine Tabelle erstellt. Da hier die Tabellen eindeutig den Entities zugeordnet werden können, wird kein Unterscheidungsfeld (Discriminator) benötigt. [2]

Die Tabellen für die konkreten Klassen enthalten neben ihren eigenen Attributen auch alle Attribute der geerbten Klasse. Das hat den Nachteil, dass, wenn ein Attribut der Superklasse geändert wird, alle Tabellen der Subklassen geändert werden müssen. Ein weiterer Nachteil dieser Strategie ist, dass polymorphe Abfragen nicht optimal unterstützt werden können, da hierfür mehrere SELECTs benötigt werden: [2, 3]

```
select id, name, medium, lengthInMinutes from AudioBook where ...
select id, name, pages from Paperback where ...
```

Falls der O/R-Mapper UNION unterstützt, kann sich die Abfrage wieder auf einen Datenbankzugriff reduzieren.

```
select * from (  select id, name, ..., 'A' as type from AudioBook
union
select id, pages, 'P' as type from Paperback ) where ...
```

Absolut unbrauchbar wird diese Strategie, wenn polymorphe Assoziationen ins Spiel kommen. Von der Entity Publisher soll mit getAbstratctBooks() auf alle Bücher zugegriffen werden, das heißt, der Fremdschlüssel zu Publisher müsste eigentlich in AbstractBook sein. AbstractBook ist aber eine abstrakte Klasse und wird daher nicht als Tabelle in der Datenbank gemappt. In diesem Fall müssten alle Subklassen, hier AudioBook und Paperback, einen Fremdschlüssel auf Publisher haben. Es wird empfohlen, hierfür eine der anderen beiden Vererbungsstrategien zu wählen. [2, 3]

Vorteil dieser Strategie ist, dass Abfragen auf konkrete Klassen sehr einfach und performant sind:

```
select id, name, ..., pages from Paperback
```

Wie bereits im vorherigen Kapitel erwähnt, ist dieser Vererbungsstrategie für Vererbungshierarchien ohne polymorphe Abfragen und Beziehungen gut geeignet.

7.3 Eine Tabelle pro Klasse

Bei dieser Vererbungsstrategie wird für jede abstrakte und konkrete Klasse eine Tabelle angelegt. Dies bedeutet, dass für "AbstractBook", "Paperback" und "AudioBook" eine Tabelle erstellt wird. Jede Tabelle hat einen Primärschlüssel id, der zugleich auch Fremdschlüssel zur Superklasse ist. [3]

JOINED hat den Vorteil, dass die Datenbankintegrität nicht verletzt wird, da beispielsweise Attribute in den Subklassen nicht nullable sein müssen. Wesentlich komplexer werden allerdings die Abfragen: Für eine polymorphe Abfrage über alle AbstractBooks muss ein Select-Statement mit outer joins generiert werden, und für eine Abfrage auf eine konkrete Klasse benötigt ein Select-Statement immer INNER JOINs: [2]

```
select b.id, b.name, p.pages from Paperback p inner join Book b on
b.id = p.id  where ...
```

7.4 Gegenüberstellung

Tabellarische Gegenüberstellung der einzelnen Arten, eine Klassenhierarchie in der Datenbank abzubilden:

Eine Tabelle pro	*Konkrete Abfrage*	*Polymorphe Abfrage*	*Polymorphe Assoziationen*	*Vielzahl unterschiedlicher Attribute*	*Komplexe Vererbungshierarchie*	*Datenintegrität*
Hierarchie	+	+	+	-	-	-
Konkreter Klasse	+	-	-	~	~	+
Klasse	-	-	+	+	+	+

8 Hibernate

8.1 Allgemeines

Hibernate ist ein open source ORM-Framework, implementiert in der Sprache Java. Es bietet unter Anderem eine vollständige Implementierung der JPA Javas und findet somit Anwendung sowohl in Java SE als auch EE Applikationen. [1][2]

8.2 Entities

Entities entstammen der Java Persistence API (JPA) und sind die Nachfolger der Entity Beans aus Java EE. Sie werden mit @Entity gekennzeichnet und unterstützen sowohl abstrakte und konkrete Klassen, als auch Vererbunge, polymorphe Assoziationen und Abfragen. Die Attribute einer Entity-Klasse werden als solche persistiert und können mit weiteren Annotations mit zusätzlicher Information bestückt werden. Diese Klassen verlangen bei private Attributen getter- und setter-Methoden und einen Defaultkonstruktor. [2]

8.3 JPA Entity Manager und Hibernate Session

Um eine Entity-Klasse zu persistieren ist es notwendig, dass ein Persistencekontext die Entities die persistiert sind oder persistiert werden sollen verwaltet. Dieser Persistencekontext kann 2 verschiedenen Implementierungen untergeordnet sein:

- JPA EntityManager
- Hibernate Session

Diese sind trotz ihres möglicherweise verwirrenden Namens auf der gleichen Ebene und arbeiten die Persistierung der Entities in die Datenbank ab. Diese Klassen werden, dem Designpattern Session-per-Request nach, bei jedem Request erzeugt und kapseln die Transaktionen.
Für beide Implementierungen gibt es 2 Vorgehensweisen:

- Container-managed
- Application-managed

Container-managed bedeutet, dass sich der Container um jegliche Lebenszyklen kümmert. Sowohl Sessions bzw. EntityManager als auch Transaktionen werden vom Container gestartet als auch beendet und die Applikation hat lediglich mittels Annotations Einfluss auf diese Komponenten (bspw. kann man Anfordern dass eine extra Transaktion gestartet wird, da prinzipiell einer vorhandenen gejoined wird, sofern eine verfügbar ist).

Container-managed

Eine Container-managed-Session/-EntityManager wird als solche gekennzeichnet, indem sie mit der Annotation @PersistenceContext beschrieben wird. Will man nun eine sog. PersistenceUnit (Daten für die DB-Verbindung) wählen die im File „persistence.xml“ vorhanden sind wählen, natürlich nur wenn mehr als 1 vorhanden ist, kann man diese als Parameter angeben:

```
<persistence-unit name="ExercisesPU" transaction-type="JTA">
    <provider>com.objectdb.jpa.Provider</provider>
    <properties>
      <property name="javax.persistence.jdbc.url" value="$objectdb/db/exercises.odb"/>
      <property name="javax.persistence.jdbc.user" value="admin"/>
      <property name="javax.persistence.jdbc.password" value="admin"/>
    </properties>
  </persistence-unit>
</persistence>
```

So sieht bspw. ein persistence.xml-File aus.

```
@PersistenceContext(unitName = "ExercisesPU")
private EntityManager em;
```

Würde nun einen EntityManager injecten lassen, welcher die Daten der persistence-unit „ExercisePU“ verwendet. Selbiges gilt für die Session:

```
@PersistenceContext(unitName = "ExercisesPU")
private Session session;
```

Hibernate folgt standardmäßig dem Pattern Session-per-Request und erstellt bei jedem Request eine Session bzw. einen EntityManager, sofern diese containter-managed sind, also wie hier injected werden.

Mit dem EntityManager kann man nun, sofern container-managed vorgegangen wurde, simple Methoden ausführen:

```
em.persist(SomeEntity);
em.merge(AnotherEntity);
em.remove(ThirdEntity);
```

Neben diesen Methoden zum persistieren, mergen und entfernen einer Entity kann auch mit *find()* gesucht werden und andere Methoden sind in der Doku nachzulesen.

Da Transaktionen prinzipiell in diesem Fall von dem Container gehandled werden, kann es jedoch passieren, dass man für spezielle Aufgaben eine Transaktion benötigt. Diese kann per Annotation angefordert werden:

```
@TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
public void addItem(String item, Integer quantity) {

    em.persist(object);

}
```

Dieses TransactionAttribute verlangt bspw. dass in jedem Fall eine neue Transaktion gestartet wird, irrelevant ob bereits welche vorhanden sind. Zudem kann dieser Transaktion nicht gejoined werden und sie wird committed, sobald die Methode ausgeführt wurde.
Andere TransactionAttributes sind in der Dokumentation nachzulesen. Diese committen jedoch für gewöhnlich, wenn alle Methoden des Requests abgearbeitet sind.

Application-managed
Die zweite Option ist die application-managed Vorgehensweise. Man kann auch hier sowohl Sessions als auch EntityManager application-managed aufrufen, was einem den Vorteil bringt, sowohl Sessions als auch Transaktionen komplett selbst zu implementieren. Hier müssen die Lifecycles beider manuell

behandelt werden. Um eine Session bzw. einen EntityManager zu erstellen benötigt man eine Session-/EntityManager-Factory. Diese kann wie folgt injected werden:

```
@PersistenceUnit(unitName = "ExercisesPu")
EntityManagerFactory emf;
```

Die SessionFactory muss hier jedoch auf eine andere Art erzeugt werden. Sie kann mithilfe eines Configuration-Objekts erzeugt werden, dem zuvor mittels cfg-File von Hibernate die Daten übergeben werden müssen.

8.4 JPQL

Bei der Java Persistence Query Language handelt es sich um eine Abfragesprache, die die veraltete EJB QL ersetzt hat. Hierbei handelt es sich um eine plattformunabhängige und objekt-orientierte Sprache, welche innerhalb der JPA spezifiziert wurde. Diese Sprache wird verwendet, um persistierte Entities einer relationalen DB abzufragen. Der Vorteil dieser Sprache ist die bereits erwähnte plattformunabhägikeit, außerdem sind die Ergebnisse polymorph. Der Nachteil ist jedoch der gleiche, da die Sprache nicht plattformabhängig ist, kann sie nicht alle Features eines gewissen Systems unterstützen. Ob nun die JQPL oder doch SQL verwendet werden sollte ist situationsbedingt, sollte es jedoch möglich sein, ist die JPQL zu bevorzugen. [2][4]

8.5 Schnittstellen

Transaktionen sind Abschnitte, die Befehle bzw. Ausführungen enthalten, die nach Abhandlung dieser entweder zurückgenommen (Rollback) oder abgeschickt und damit tatsächlich ausgeführt werden können (commit).
Die JPA unterstützt 2 Arten von Transaktionen, zum Einen die der JTA und zum Anderen über EntityTransaction-Interfaces.
Bei der JTA wird mittels des Entitymanagers an einer bereits bestehenden Transaktion teilgenommen. Diese Transaktionen werden unabhängig von EntityManager erzeugt bzw. geschlossen.
Die andere Option, Resource-Local-Transactions, ermöglicht einem die direkte Steuerung. Man erhält ein Transaction Objekt mithilfe der Methode *getTransaction()*, sollte der Entity-Manager auf den diese Methode aufgerufen wird für die Verwendung von JTA konfiguriert sein, wird eine Exception geworfen. [2]
Mit den Befehlen *begin()*, *commit(), rollback(), setRollbackOnly(), getRollbackOnly* und *isActive()*, können die verschiedenen Befehle einer Transaktion durchgeführt werden.

Auch hier muss erneut unterschieden werden mit welcher Art von EntityManager gearbeitet wird. Handelt es sich um ein Container-managed Umfeld,

8.5.1 Locks

Zum Problem der Locks kommt es auch in diesen Anwendungen, insbesondere bei Multiuseranwendungen. Für Locking kann auf verschiedene Arten vorgegangen werden, unter Anderem gibt es das optimistische als auch das pessimistische Locking. [2]

8.5.1.1 Optimistisches Locking

Beim optimistischen Locking wird davon ausgegangen, dass es zu keinerlei Überschneidungen kam und die Änderungen werden appliziert. Sollte es nun doch zu Fehlern gekommen sein, müssen diese mühsam und manuell nachbearbeitet werden. [2]
In der JPA wird optimistisches Locking mittels Versionierung umgesetzt, wobei jede Entity ein mit *@Version* gekennzeichnetes Attribut bekommt, welches einem der folgenden Datentypen entsprechen muss: [2]

- int
- Integer
- short
- Short
- long
- Long
- Timestamp

Sobald dieses Attribut angelegt wird, wird für diese Entity optimistisches Locking aktiviert.
Bei einem Update wird dieses Attribut mit dem persistierten verglichen, stimmen diese nicht überein, ist klar, dass es zu einem Fehler kam.
Prinzipiell wird beim optimistischen Locking nicht bis zum tatsächlichen Aufruf des commits gelocked. Die Daten werden normal bearbeitet, ohne Zeilen oder Entities zu sperren und bevor der tatsächliche Commit auf die DB ausgeführt wird, wird überprüft ob der zuvor ausgelesene Wert noch der gleiche ist (wenn nicht → Fehler) und danach für das Update kurzfristig gelocked.
Sollte bei diesem Update die Versionsnummer nicht mehr mit der damals gelesenen übereinstimmen, wird auch hier ein Fehler geworfen.

8.5.1.2 Pessimistisches Locking

Beim pessimistischen Locking wird davon ausgegangen, dass es häufig zu Überschneidungen kommt und in diesem Fall wird die bearbeitete Zeile gesperrt sobald darauf zugegriffen wird. Diese Art des Locking erhöht natürlich die Sicherheit der Konsistenz, führt jedoch zu Performanceeinbußen. [2]

8.5.1.3 Lock-Modi

Mit der Methode *EntityManager.lock()* wird ein Lock aufgerufen. Für diesen Lock gibt es jedoch verschiedene Typen, sogenannte LockModeTypes. Die folgenden sind jene, die von der JPA zur Verfügung gestellt werden, nicht direkt von Hibernate:

- OPTIMISTIC (vor JPA 2.0 als READ bezeichnet)
  Hier wird optimistisches Locking umgesetzt.
- OPTIMISTIC_FORCE_INCREMENT (vor JPA 2.0 als WRITE bezeichnet)
  Hier wird auch optimistisches Locking umgesetzt, aber das Versionsattribut wird bei jedem Zugriff auf das Objekt erhöht

Seit JPA 2.0 ist nun auch pessimistisches Locking möglich. Man spricht hierbei von pessimistischen Locks, wenn langandauernde Locks sofort durchgeführt werden.

- PESSIMISTIC_READ
  Die Entity und zugehörigen Zeilen werden gelockt, es kann jedoch noch gelesen werden. (Shared Lock)
- PESSIMISTIC_WRITE
  Ein Schreiblock auf Entity und DB-Zeilen wird ausgeführt.
- PESSIMISTIC_FORCE_INCREMENT
  Das Versionsattribut wird unabhängig von Änderungen inkrementiert.

Für Hibernate gibt es andere LockModes. Diese werden bspw. einer Session gesetzt und können mit den Methoden *Session.load(), Session.lock(), Session.get() und Query.setLockMode* gesetzt werden. [2]
Diese Lockfunktionalität kann abhängig von der DB nicht verfügbar sein. Sollte dies der Fall sein, verwendet Hibernate automatisch den, der diesem LockMode am nähesten kommt. Folgende Modi sind verfügbar:

- NONE
  Hier wird kein Lock auf Zeilen der DB gesetzt. Prinzipiell wird nur auf die DB zugegriffen, falls sich die Entity nicht im Cache befindet.
- READ
  Hier wird direkt auf die DB zugegriffen und die Version der Entities wird überprüft.
- UPGRADE
  Identisch mit READ der JPA
- UPGRADE_NOWAIT
  Dieser Typ ist ähnlich dem UPGRADE, Oracle unterstützt jedoch den Befehl SELECT FOR UPDATE NOWAIT. Hier wird lediglich bei einem vorhandenen Lock nicht gewartet, sondern eine Exception geworfen.
- WRITE
  Ein Modus, welcher nicht vom User verwendet werden kann. Dieser wird automatisch von Hiebrnate verwendet, sollte ein Datensatz aktualisiert oder eingefügt werden.

Für Entities die mittels JPA gelocked werden sollen, ist es notwendig dass sich diese bereits im PersistenceKontext befinden. Bei Hibernate ist dies nicht der Fall. [2]

Optimistisches Locking hat bei Hibernate erweiterte Einstellungen. Mittels Annotation können für den Optimistic Lock die folgenden Strategien gesetzt werden: [2]

- NONE
  optimistisches Locking wird deaktiviert
- VERSION
  Defaultverhalten. Überprüfung findet statt, überlappende Änderungen möglich.
- DIRTY
  Nur veränderte Attribute der Entity werden mittels Versionsüberprüfung gechecked. Überlappende Änderungen auch hier möglich.
- ALL
  Alle Attribute der Entity werden auf Veränderung überprüft. Kann dann verwendet werden, wenn kein *@Version* Attribut verwendet werden kann.

Sollen DIRTY oder ALL verwendet werden, ist *dynamic-update=true* zu setzen. Will man ein Attribut aus den Checks entfernen, kann es mit *excluded=true* gekennzeichnet werden. [2]

8.6 Beispiel Klasse zur Persistierung

```
@Entity
@Table(name = "ACCOUNT")
public class Account {
    @Id
    @Size(max = 100)
    String email;

    @Size(max = 100)
    String name;

    @NotEmpty
    String password;

    @JsonCreator
    public Account(@JsonProperty("email") String email,
@JsonProperty("name") String name,@JsonProperty("password") String
password) {
        this.email = email;
```

```
        this.name = name;
        this.password = password;
    }

    protected Account(){}
}
```

Bei diesem Beispiel handelt es sich um eine simple Klasse Account, welche in einer DB mittels Hibernate persistiert werden soll. Bei diesem Beispiel wurden die Annotations der Hibernateimplementierung der JPA verwendet.

@Entity stammt aus der JPA und legt fest, dass diese Klasse persistiert werden soll.
@Table gibt einem Möglichkeiten verschiedene Konfigurationen der Tabelle

Folgende Kapitel, einschließlich diesem werden noch ergänzt

8.7 Assoziationen

8.8 Vererbung

8.9 Collections

9 Quellen

[1] Hibernate ORM, http://hibernate.org/orm/ , zuletzt abgerufen am: 31.05.2016
[2] JPA mit Hibernate, Daniel Röder, publiziert bei: Software & Support Verlag GmbH in 2010
[3] Hibernate Recipes: A Problem-Solution Approach, Srinivas Guruzu & Gary Mak, publiziert bei: Apress Media LLG in 2010
[4] JPQL Language Reference, http://docs.oracle.com/cd/E12839_01/apirefs.1111/e13946/ejb3_langref.html , zuletzt abgerufen am: 31:05.2016
[5] Configuration, https://docs.jboss.org/hibernate/core/3.3/reference/en-US/html/session-configuration.html , zuletzt abgerufen am: 31.05.2016