Dirk Weil

Java EE 7

Enterprise-Anwendungsentwicklung leicht gemacht

entwickler.press

Inhaltsverzeichnis

3 Java Persistence

3.1 Worum geht's?

Vor der Behandlung der Details soll hier kurz abgesteckt werden, in welchem Bereich der Anwendungsentwicklung wir uns in diesem Kapitel befinden. Wie der Name schon sagt, geht es um Persistenz, d. h. das Speichern und Laden von Java-Objekten in bzw. aus einem dauerhaften Speichermedium. Java Persistence schränkt den Blick noch etwas weiter ein, und zwar auf relationale Datenbanken als Speichermedium, was sicher den allergrößten Teil von Unternehmensanwendungen abdeckt.

Die Aufgabenstellung lautet also, Objekte von Java-Klassen wie der in Listing 3.1 auf eine Tabelle einer relationalen Datenbank wie der in Abbildung 3.1 abzubilden, d. h. die grundlegenden Operationen der Datenbank wie Erzeugen und Löschen von Einträgen, Lesen und Verändern von Werten oder Suchanfragen möglichst einfach und effektiv zur Verfügung zu stellen.

```
public class Country
{
  private String isoCode; // Primärschlüssel
  private String name;
  private String phonePrefix;
  private String carCode;

  // Getter und Setter nach Bedarf …
}
```

Listing 3.1: Beispielhafte Java-Klasse für persistente Daten

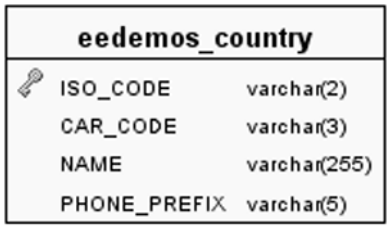

Abbildung 3.1: Beispieltabelle

3.1.1 Lösungsansätze

Um die beschriebene Aufgabe zu lösen, stehen uns unterschiedliche Wege offen. Zunächst enthält Java mit JDBC eine Abstraktion für Datenbankzugriffe als Core Library. Damit kann man sich die gewünschten Operationen z. B. in einer Helferklasse zur Verfügung stellen (Listing 3.2).

```
public class CountryJdbcHelper
{
  public void save(Country country)
    throws ClassNotFoundException, SQLException
  {
    Connection connection = JdbcUtil.getConnection();
    PreparedStatement statement = null;
    try
    {
      statement = connection.prepareStatement(
       "insert into EE_DEMOS_COUNTRY(ISO_CODE,NAME,PHONE_PREFIX,CAR_CODE)
        values (?,?,?,?)");
      statement.setString(1, country.getIsoCode());
      statement.setString(2, country.getName());
      statement.setString(3, country.getPhonePrefix());
      statement.setString(2, country.getCarCode());
      statement.executeUpdate();
      …
```

Listing 3.2: DB-Zugriff per JDBC[1]

Aus Gründen der Übersichtlichkeit ist im Listing der Auf- und Abbau der Verbindung zur Datenbank weggelassen worden. Ebenso ist die Exception-Verarbeitung nur angedeutet. Man sieht daran schon, dass man sich bei der direkten Nutzung von JDBC auf einer recht niedrigen Ebene im Programm bewegt. Das ist zwar nicht wirklich schwierig, aber doch nicht mal eben gemacht.

Im Bereich der serverbasierten Anwendungen, mit dem sich dieses Buch beschäftigt, gab es schon recht früh Ansätze zur Lösung der Thematik auf einer etwas höheren Ebene. Der EJB-Standard enthält bis zur Version 2.1 Entity Beans, die von der direkten Nutzung von JDBC abstrahieren und bei Nutzung der sog. Container Managed Persistence die „niederen Aufgaben“ dem EJB-Container überlassen. Diese an sich gute Idee wurde allerdings unglücklicherweise mit dem Komponentenkonzept von EJBs vermischt, obwohl die damit verbundenen Vorteile wie bspw. Ansprechbarkeit über Remote- und Local-Interfaces, konfigurierbares Environment etc. für den Persistenzaspekt keine Vorteile bringen, ja sogar hinderlich sind. So gerieten die Entity Beans sehr schnell in den Ruf, schwergewichtige und unflexible Monster zu sein, was sogar auf die gesamte EJB-Spezifikation ausstrahlte und viele Projekte komplett von der Nutzung von EJB abgebracht hat.

1 Den in diesem Kapitel gezeigten Beispielcode finden Sie im Begleitprojekt *ee-demos-jpa*

Parallel zu den EJBs – teilweise sogar schon früher – traten die sog. O/R-Mapper auf den Plan. Die Object/Relational-Mapping-Frameworks verfolgen grob den Ansatz, möglichst wenig Restriktionen bezüglich der speicherbaren Objekte aufzubauen, also einfache Java-Klassen als persistente Klassen zu nutzen und diese so mit Metadaten in Form von XML oder später auch Annotationen anzureichern, dass mithilfe von Frameworkmethoden ein Speichern und Laden der Objekte möglich wird.

Das mündete einerseits in einen Java-Standard: Java Data Objects (JDO). Der Persistenzaspekt wurde bei den JDO-implementierenden Produkten häufig durch ein Bytecode Enhancement nach dem Compile der Klassen in den Bytecode hineingewebt, was zum damaligen Zeitpunkt ein neues und ungewöhnliches Vorgehen war. JDO hat interessanterweise nie die Marktbreite erreicht, die man eigentlich hätte erwarten können, vielleicht weil nahezu zeitgleich ein äußerst populäres Open-Source-O/R-Framework namens Hibernate auftauchte, das mit ganz ähnlichen Konzepten glänzte, statt des Byte Code Enhancements zur Build-Zeit allerdings auf dynamische Proxies und Bytecode-Modifikation zur Laufzeit setzte.

Die Ideen aus JDO und Projekten wie Hibernate sind es nun, die sich im Standard Java Persistence wiederfinden. Mehr noch: Produkte wie Hibernate, EclipseLink (Nachfolger des O/R-Urgesteins Toplink), oder DataNucleus (Referenzimplementierung von JDO) sind heute Implementierungen von JPA.

Ein Wort zu den Begrifflichkeiten: Der Standard heißt eigentlich Java Persistence. Im gängigen Sprachgebrauch findet sich aber häufig die gerade erwähnte Abkürzung JPA für Java Persistence API. Im Rahmen dieses Buches werden die Begriffe synonym verwendet.

3.1.2 Anforderungen an O/R-Mapper

Die Erwartungen an ein Persistenzframework lassen sich teilweise schon an der selbstgestrickten JDBC-Lösung ablesen. Darüber hinaus entwickeln sich rasch weitere Wünsche in Richtung Einfachheit und Komfort, sodass im Endeffekt eine Liste wie diese herauskommt – ohne Gewichtung und Anspruch auf Vollständigkeit:

- Verbindungsverwaltung
- Mapping von Klassen und Attributen auf Tabellen und Spalten
- Formulierung und Ausführung von SQL-Befehlen
- Transaktionssteuerung
- Verwaltung von Relationen mit Navigation darüber
- Abbildung von Vererbungsbeziehungen
- Generierung technischer IDs
- Komfortable, objektorientierte Suchmöglichkeiten
- Caching

3.1.3 Entwicklung des Standards

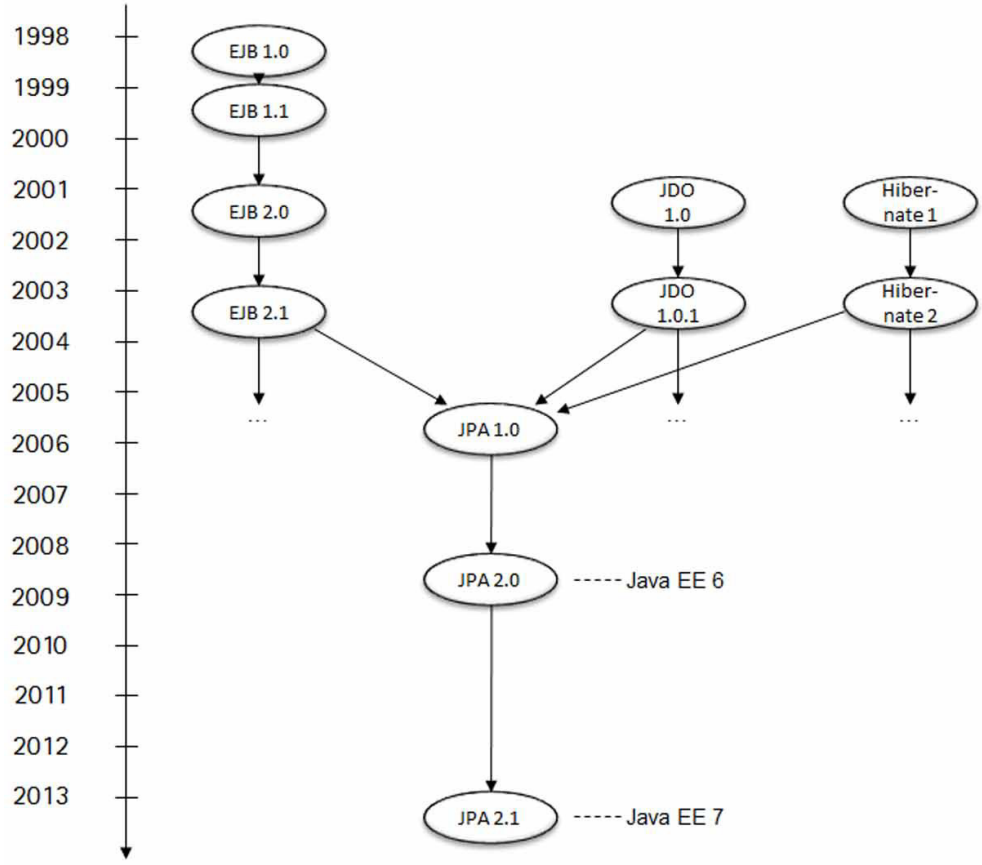

Abbildung 3.2: Historie von Java Persistence

Der Bedarf an Frameworks zur Abbildung von Java-Objekten auf Datenbankeinträge ist immer schon groß gewesen. So hatten sich bis zur Mitte der vergangenen Dekade mehrere Alternativen entwickelt, u. a. die in Abbildung 3.2 gezeigten Spezifikations- bzw. Produktstränge EJB, JDO und Hibernate. War man bis zu dem Zeitpunkt in kontroverse, teilweise schon als religiös zu bezeichnende Diskussionen verstrickt, haben sich die verschiedenen Team dann doch zusammengerauft und den gemeinsamen Standard JPA 1.0 als Teil der Java EE 5 aus der Taufe gehoben. JPA wurde damals gemeinsam mit EJB 3.0 im JSR 220 entwickelt. Seit JPA 2.0 wird die Weiterentwicklung unabhängig von EJB vorgenommen: JSR 317 für die Version 2.0, JSR 338 für 2.1.

3.1.4 Architektur von Anwendungen auf Basis von JPA

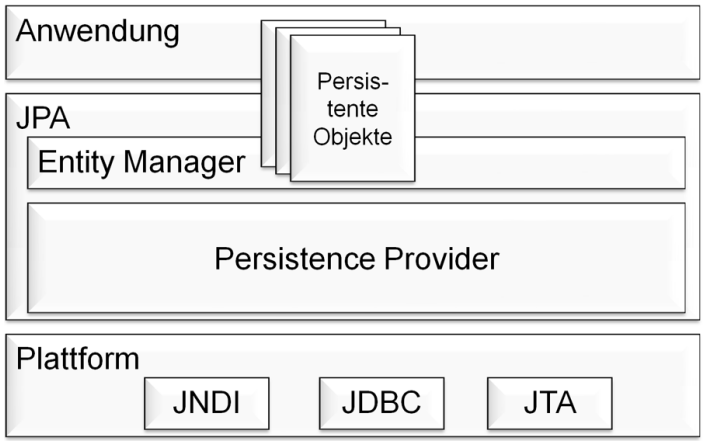

Abbildung 3.3: 10 000-m-Sicht auf JPA-Anwendungen

Nutzt eine Anwendung Java Persistence, so kommuniziert sie mit dem JPA-Framework i. W. mithilfe des dort vorhandenen Entity Managers (Abbildung 3.3). Die zu verarbeitenden Daten werden als persistente Objekte ausgetauscht, die sich – wie später deutlich werden wird – nur wenig von normalen Java-Objekten unterscheiden.

Der Entity Manager ist technisch als Java-Interface ausgebildet. Die konkrete Implementierung ist nicht in JPA selbst enthalten, sondern wird von einem Persistence Provider beigesteuert. Dafür stehen u. a. folgende Produkte zur Verfügung:

- EclipseLink (Referenzimplementierung)[2]
- Hibernate[3]
- OpenJPA[4]

Als Plattform reicht JPA eine Java-SE-Umgebung. Es nutzt dann JDBC und bietet eigene Methoden zur Initialisierung und Transaktionssteuerung an. In einem Applikationsserver nutzt JPA die darin angebotenen Infrastrukturdienste wie Datasources oder globale Transaktionssteuerung. Das Buch fokussiert hauptsächlich auf die serverseitige Nutzung.

Die Java-EE-Applikationsserver haben verpflichtend einen eingebauten JPA Provider. Im Fall von GlassFish wird EclipseLink verwendet, JBoss nutzt Hibernate, Geronimo wird mit OpenJPA ausgeliefert und WebSphere wie auch WebLogic nutzen eine von OpenJPA abgeleitete Implementierung.

2 *http://www.eclipse.org/eclipselink/*
3 *http://www.hibernate.org/*
4 *http://openjpa.apache.org/*

3.2 Die Basics

Vor der Benutzung von JPA sind nur wenige Hürden zu nehmen, da für die meisten Konfigurationsparameter sinnvolle Vorgabewerte existieren. Das in der Java EE häufig anzutreffende Prinzip „Convention over Configuration" findet auch hier Anwendung: Eine explizite Konfiguration ist nur dann nötig, wenn von den Vorgaben abgewichen werden soll. Lassen Sie uns im Folgenden einen Blick auf die Grundlagen der Nutzung von JPA werfen – zunächst anhand einfacher persistenter Objekte. Relationen, Vererbung und andere Spezialitäten heben wir uns für später auf.

3.2.1 Entity-Klassen

Entities, d. h. persistente Objekte im Sinne von JPA, sind – Sie ahnen es schon – POJOs. Entity-Klassen sind also gemäß dieser im Java-EE-Umfeld allgegenwärtigen Bezeichnung nichts Besonderes, haben insbesondere keine vorgeschriebene Basisklasse und müssen kein bestimmtes Interface implementieren. Einzige Voraussetzung ist die Existenz eines parameterlosen Konstruktors. Mithilfe von Annotationen aus dem Paket *javax.persistence* oder alternativ mittels eines XML-Deskriptors werden den Klassen Metainformationen für die Nutzung durch JPA mitgegeben. Vieles davon ist optional, sodass die Klasse *Country* in Listing 3.3 bereits eine ausreichend ausformulierte persistente Klasse darstellt.

```
@Entity
@Access(AccessType.FIELD)
public class Country
{
  @Id
  private String isoCode;
  private String name;
  private String phonePrefix;
  private String carCode;

  // Getter und Setter nach Bedarf …
}
```

Listing 3.3: Entity-Klasse mit Field Access

Im Vergleich zu einer normalen Java-Klasse fallen nur die drei Annotationen auf: *@Entity*[5] markiert die Klasse als Entity, *@Access* wählt das Attributzugriffsverfahren (s. u.), und *@Id* bestimmt das identifizierende Attribut – gleichbedeutend mit dem Primärschlüssel in der Datenbank.

5 Alle hier verwendeten Klassen stammen aus dem Paket *javax.persistence* und seinen Unterpaketen

Soll die Klasse parameterbehaftete Konstruktoren enthalten, muss darauf geachtet werden, dass auch ein parameterloser Konstruktor definiert wird. Es reicht, wenn dieser *protected* ist.

Die Annotation *@Id* am Feld *isoCode* bewirkt, dass beim Schreiben eines Eintrags in die Datenbank der Primärschlüsselwert direkt aus der Instanzvariablen herausgelesen wird. Umgekehrt wird ein aus der Datenbank gelesener Wert direkt dort hineingeschrieben. Mehr noch: Dieses Field Access genannte Verfahren gilt automatisch auch für alle weiteren Felder der Klasse. Die Variablen dürfen durchaus *private* sein.

Alternativ kann der JPA-Zugriff auf Entity-Objekte auch über Getter bzw. Setter geschehen. Dazu müssen statt der Instanzvariablen die Getter annotiert werden (Listing 3.4).

```
@Entity
@Access(AccessType.PROPERTY)
public class Country
{
  private String isoCode;
  // …

  @Id
  public String getIsoCode()
  {
    return this.isoCode;
  }
  protected void setIsoCode(String isoCode)
  {
    this.isoCode = isoCode;
  }
  // …
}
```

Listing 3.4: Entity-Klasse mit Property Access

Auch hier gilt dieser Property Access automatisch für alle Getter/Setter-Paare der Klasse. Die Methoden müssen nicht *public* sein, *protected* reicht.

Das Attributzugriffsverfahren wird mit der Annotation *@Access* explizit gewählt. Dies kann wie gezeigt auf Klassenebene geschehen wie auch bei einzelnen Attributen. Eine Vermischung von Field Access und Property Access ist damit möglich, aber nicht empfehlenswert. Ohne *@Access* ergibt sich das Verfahren implizit durch die Platzierung der Attributannotationen, wobei dann keine Vermischung innerhalb einer Entity erlaubt ist.

Ob Sie Field Access oder Property Access einsetzen, ist Ihnen vollkommen freigestellt. Mir gefällt Ersteres besser, da die Persistenzannotationen – wenn es denn später mehr als nur *@Id* sind – nicht so verstreut werden und Getter und Setter nur bei fachlichem Bedarf vorgehalten werden müssen. Beim Property Access ist andererseits positiv, dass durch den Methodenzugriff eine Entkopplung von der reinen Datenstruktur stattfindet.

3.2.2 Konfiguration der Persistence Unit

Ganz ohne XML geht es nicht: Ein Deskriptor namens *META-INF/persistence.xml* ist notwendig, um die Verbindung zur Datenbank zu spezifizieren. Listing 3.5 zeigt ein Beispiel dafür.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
  http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
  version="2.1">
  <persistence-unit name="ee-demos">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <jta-data-source>jdbc/eeDemos</jta-data-source>
    <properties>
      <property name="eclipselink.ddl-generation"
                value="create-or-extend-tables" />
      <property name="eclipselink.ddl-generation.output-mode"
                value="database" />
    </properties>
  </persistence-unit>
</persistence>
```

Listing 3.5: Persistenzdeskriptor[6]

Der Persistenzdeskriptor deklariert ein oder mehrere Persistence Units, deren Namen – im Beispiel *ee-demos* – frei gewählt werden können. Die Angabe des Persistenzproviders mit dem Element *<provider>* ist optional. Wird sie weggelassen, wird der Standardprovider des Zielservers verwendet, im GlassFish also bspw. EclipseLink. Die einzige verpflichtende Angabe ist die der zu nutzenden Datasource. Hier muss der JNDI-Name einer im Server passend konfigurierten Datasource angegeben werden.

Mithilfe der Properties können spezielle Eigenschaften des verwendeten Providers konfiguriert werden. Die oben angegebene Property *eclipselink.ddl-generation* kann bspw. genutzt werden, um es EclipseLink zu erlauben, Schemaänderungen in der Datenbank vorzunehmen. Der Wert *create-or-extend-tables* wird den Entwicklern unter uns gut gefallen, während die DB-Administratoren ein mulmiges Gefühl beschleicht: Hier folgt das DB-Schema der Klassenstruktur der Entities, beim Start der Anwendung nicht vorhandene Tabellen oder Spalten werden automatisch angelegt. Dies ist ein idealer Modus in den Entwicklungsphasen einer Anwendung. Und bevor die DB-Administratoren sich schaudernd abwenden, stellen wir die Property für ein Release der Anwendung auf *none*. Es stehen noch weitere Modi zur Verfügung und auch andere Provider haben ein solches Feature. Schauen Sie bei Bedarf in die entsprechende Dokumentation.

An dieser Stelle sei darauf hingewiesen, dass hier zunächst nur der Einsatz von JPA innerhalb einer gemanagten Umgebung beschrieben wird, d. h. innerhalb eines Applikations-

6 Sollten Sie noch die Version 2.0 verwenden wollen oder müssen, ersetzen Sie in den XML-Namespace-Angaben *xmlns.jcp.org* durch *java.sun.com* sowie 2_1 und 2.1 durch 2_0 bzw. 2.0

servers, der zumindest CDI beherrscht. Am Ende des Kapitels finden Sie Informationen darüber, wie JPA in SE-Anwendungen genutzt werden kann.

Der Deskriptor hat noch eine weitere Bedeutung: Er bestimmt die Klassen, die JPA berücksichtigt, die also Teil der Persistence Unit sind. Dies sind alle Klassen mit entsprechender Annotation, die in einem Classpath-Verzeichnis oder einem JAR File liegen, das in seinem Verzeichnis *META-INF* den Deskriptor *persistence.xml* enthält. In einer Webanwendung muss der Deskriptor also in *WEB-INF/classes/META-INF* platziert werden, damit die in *WEB-INF/classes* abgelegten Klassen für JPA Berücksichtigung finden. In Abbildung 3.4 ist dies beispielhaft dargestellt.

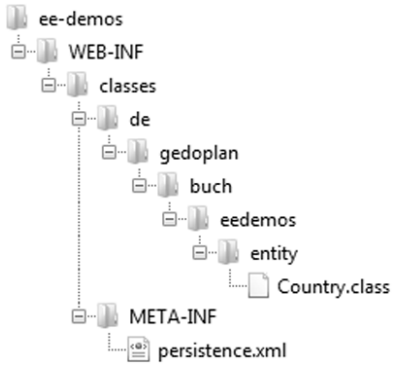

Abbildung 3.4: Persistence Unit in einer Webanwendung

Mithilfe weiterer Elemente lässt sich der Geltungsbereich der *persistence.xml* einschränken oder auch auf weitere JAR Files ausdehnen. Details dazu findet man in Kapitel 8 (Entity Packaging) der Java-Persistence-Spezifikation.

Zusätzlich zum Persistenzdeskriptor kann man im Verzeichnis *META-INF* auch die Mapping-Datei *orm.xml* ablegen, wenn man statt mit Annotationen mit XML arbeiten möchte. Damit könnte man bspw. das oben gezeigte Mapping der Klasse *Country* ohne Annotationen definieren, siehe Listing 3.6.

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
  http://java.sun.com/xml/ns/persistence/orm_2_0.xsd"
  version="2.0">
  <entity class="de.gedoplan.buch.eedemos.entity.Country" access="FIELD">
    <attributes>
      <id name="isoCode"/>
    </attributes>
  </entity>
</entity-mappings>
```

Listing 3.6: Mapping-Datei

Weitere Mapping-Dateien können mithilfe von <*mapping-file*>-Elementen in *persistence.xml* angemeldet werden. Die Nutzung von XML-Mappings anstelle der entsprechenden Annotationen ist dann nützlich, wenn man den Java-Quellcode der betroffenen Klassen nicht mit Annotationen versehen möchte oder kann, bspw. weil man nicht über die Quellen verfügt. Ansonsten ist die Verwendung von Annotationen natürlich einfacher und auch fehlersicherer.

3.2.3 CRUD

Zur Arbeit mit persistenten Objekten stellt JPA das Interface *EntityManager* zur Verfügung. Objekte dieses Typs übernehmen die wesentliche Arbeit beim Laden, Speichern und Suchen von Datenbankeinträgen. Die konkrete Implementierung wird dabei vom benutzten JPA-Provider beigesteuert.

Eine Entity-Manager-Instanz kann man sich auf einfache Weise per Injektion zur Verfügung stellen lassen. Dazu dient die Annotation *@PersistenceContext*, der der Name der Persistence Unit (aus *persistence.xml*) als Parameter übergeben wird. Der Codeausschnitt in Listing 3.7 referenziert die in Listing 3.5 konfigurierte Persistence Unit *ee_demos*.

```
@PersistenceContext(unitName = "ee_demos")
private EntityManager entityManager;
```

Listing 3.7: Bereitstellung einer Entity-Manager-Instanz per Injektion

Der Parameter kann entfallen, wenn nur genau eine Persistence Unit in *persistence.xml* deklariert wurde.

Nachdem nun alles soweit vorbereitet ist, sind die grundlegenden Operationen zum Erzeugen, Lesen, Ändern und Löschen von persistenten Einträgen trivial: *EntityManager* bietet dazu entsprechenden Methoden an – siehe Listing 3.8.

```
void persist(Object entity)
<T> T find(Class<T> entityClass, Object primaryKey)
void refresh(Object entity)
void remove(Object entity)
```

Listing 3.8: „EntityManager"-Basisoperationen

Diese Operationen arbeiten allerdings auf einer anderen Ebene als man das z. B. von der direkten Ausführung von SQL-Befehlen mittels JDBC gewohnt ist: Der Zusammenhang zwischen einer Entity-Manager-Methode und der zugehörigen DB-Operation ist eher mittelbar. So führt der Aufruf von *persist* zwar schlussendlich zu einem neuen Eintrag in der Datenbank, die Ausführung des entsprechenden *INSERT*-Befehls kann aber (und wird in der Regel) verzögert geschehen, im Extremfall am Ende der Transaktion.

Womit wir schon beim Thema Transaktionen wären. JPA integriert sich dazu praktischerweise in das Transaktionssystem des Servers: Ein mit dem oben gezeigten Code injizierter Entity Manager ist automatisch transaktionsgebunden. Das bedeutet, dass er ohne weiteres Zutun eine JTA-Transaktion des Servers übernimmt und sich beim Transaktionsende passend verhält: Ein Commit führt (spätestens) zur Ausführung der notwendigen SQL-Befehle, ein Rollback dagegen zum Verwerfen eventueller Änderungen.

Die schreibenden Operationen wie *persist* und *remove* verlangen bei einem transaktionsgebundenen Entity Manager zwingend eine aktive Transaktion, die bspw. mithilfe der im vorigen Kapitel vorgestellten Annotation *@TransactionRequired* gestartet werden kann. Lesende Operationen wie *find* benötigen nicht unbedingt eine aktive Transaktion.

find sucht einen Eintrag anhand seiner ID. Auch hier ist der Zusammenhang zum analogen SQL-*SELECT* recht lose: Der Entity Manager führt einen sog. First Level Cache, in dem sich alle persistenten Objekte befinden, die der Entity Manager während seiner Lebenszeit „gesehen" hat. Man kann sich diesen Cache als *Map* vorstellen, deren Schlüssel die IDs und deren Werte die Entity-Objekte sind. Befindet sich das gesuchte Objekt bereits im Cache, liefert *find* es direkt zurück, ohne erneut die Datenbank zu befragen. Ein explizites Neuladen eines zuvor gelesenen Objekts kann mit *refresh* ausgelöst werden. Bei einem transaktionsgebundenen Entity Manager wird der Cache beim Transaktionsende geleert.

Die Methode *remove* erwartet als Parameter ein bereits gemanagtes Objekt. Das bedeutet, dass das zu löschende Objekt zuvor bspw. mittels *find* eingelesen worden sein muss. Später werden die sog. Bulk Operations beschrieben, mit denen u. a. mehrere Einträge auf einmal ohne vorheriges Einlesen gelöscht werden können.

Die bisher genannten Entity-Manager-Operationen sind zwar nur Einzeiler, dennoch erweist es sich in der Praxis als positiv, sie nicht an den benötigten Stellen der Geschäftslogik jeweils auszuprogrammieren, sondern sie in einer separaten Klasse zu sammeln, die das logische Konzept einer Datenablage – eines Repositories – modelliert. Für die anfangs eingeführte Beispiel-Entity *Country* zeigt Listing 3.9, wie diese Helferklasse aussehen könnte:

```
public class CountryRepository
{
  @PersistenceContext(name = "ee_demos")
  private EntityManager entityManager;

  @TransactionRequired
  public void insert(Country country)
  {
    this.entityManager.persist(country);
  }

  public Country findById(String id)
  {
    return this.entityManager.find(Country.class, id);
  }
```

```
  @TransactionRequired
  public void delete(String id)
  {
    Country country = findById(id);
    if (country != null)
    {
      this.entityManager.remove(country);
    }
  }
}
```

Listing 3.9: Repository-Klasse

Vermissen Sie etwas? Ja – Create, Read, Delete haben wir gesehen – und was ist mit Update? Dafür gibt es interessanterweise keine explizite Entity-Manager-Methode. Vielmehr werden alle Änderungen an Objekten, die der Entity Manager derzeit managt, automatisch spätestens bei einem Commit der Transaktion in die Datenbank gesichert.

Dieses transparente Update lässt sich sogar noch umfassender ausnutzen, wenn man statt eines transaktionsgebundenen Entity Managers einen verwendet, der seine Entity-Objekte auch nach dem Commit weiter managt. Solchen *Extended* oder *Application Managed* Entity Managern wird später in diesem Kapitel ein extra Abschnitt gewidmet.

Wie zuvor erwähnt, werden Änderungen in der Datenbank in aller Regel verzögert durchgeführt, um so mehrere Operationen zusammenzufassen oder sie im Fall eines Rollbacks komplett unterdrücken zu können. Dieser sog. Flush passiert spätestens am Transaktionsende, kann aber auch mithilfe der gleichnamigen Methode explizit ausgelöst werden. Zudem werden die Änderungen normalerweise vor Ausführung einer Query zurückgeschrieben.

3.2.4 Detached Objects

Frühere Persistenzframeworks hatten oftmals die Eigenschaft, dass die persistenten Objekte nur im Kontext des Frameworks lebensfähig waren. So ließen sich bspw. Entity Beans nur innerhalb des EJB-Containers nutzen. Die übliche Antwort der Entwickler auf solche Unzulänglichkeiten ist häufig die Postulierung von Patterns, so auch hier: Das Entwurfsmuster Data Transfer Object (oder auch Value Object) wurde genutzt, um die persistenten Daten auch außerhalb des Persistenzkontexts bereitstellen zu können. Es bedeutete i. W. die Bereitstellung einer zusätzlichen Klasse mit den gleichen inhaltlichen Eigenschaften wie die der persistenten Klasse, deren Objekte als Transportbehälter genutzt wurden: Im Persistenzkontext gelesene Daten wurden, in entsprechende Data Transfer Objects umkopiert, den Aufrufern zur Verfügung gestellt. Analog mussten die in den Clients veränderten Werte wieder in persistente Objekte kopiert werden, bevor die Ablage in der Datenbank stattfinden konnte.

Dieses zwar nicht komplizierte, aber doch aufwändige Anti-Pattern ist mit JPA obsolet geworden, da JPA-Entities auch ohne Entity Manager lebensfähig sind – es sind ja einfache POJOs. Ihre Anreicherung um Metadaten in Form von Annotationen oder XML ist dabei überhaupt nicht hinderlich. Die JPA-Entities können also direkt im Sinne der alten Data Transfer Objects verwendet werden – eine Kopie ist nicht mehr nötig.

Das Herauslösen persistenter Objekte aus dem Persistenzkontext nennt sich Detachment. Detached Objects unterscheiden sich von Persistent Objects nur dahingehend, dass es für sie keine direkte Verbindung zur Datenbank mehr gibt. Der Entity Manager hat sie sozusagen „vergessen".

Persistente Objekte lassen sich auf verschiedene Weisen implizit oder explizit detachen: Mithilfe der Entity-Manager-Methode *detach* kann ein einzelnes Objekt aus dem Entity Manager herausgelöst werden. *clear* tut dies für alle vom Entity Manager verwalteten Objekte. Dies passiert auch beim Schließen des Entity Managers mit *close* und bei einem Transaktions-Rollback. Ebenfalls detacht sind Kopien persistenter Objekte, seien sie durch Klonen oder per Serialisierung und Deserialisierung erzeugt. Letzteres ist bspw. der Fall, wenn Daten an Remote Clients geliefert werden.

Wichtig ist hier das Verständnis, dass die Detached Objects ganz normale Java-Objekte sind, bei denen es zwar keine besondere Beziehung zur Datenbank mehr gibt, die aber ansonsten beliebig verwendet, verändert oder anderweitig genutzt werden können.

Detached Objects können dem Entity Manager wieder hinzugefügt werden, und zwar mit der Methode *merge*. Dabei ist zu beachten, dass das übergebene Objekt weiterhin detacht bleibt und als Aufrufergebnis ein inhaltlich gleiches Objekt zurückgegeben wird, das persistent ist.

3.2.5 Entity-Lebenszyklus

JPA-Entities können bezüglich des Entity Managers in verschiedenen Zuständen sein:

- **Transient:** Diese Objekte sind neu oder auf andere Weise ohne Zutun des Entity Managers erzeugt worden. Datenbankeinträge dazu existieren noch nicht.
- **Persistent:** Solche Objekte werden vom Entity Manager gemanagt. Die zugehörigen Datenbankeinträge sind vorhanden und folgen den Statusänderungen der Objekte spätestens zum Transaktionsende.
- **Detached:** Dies sind aus dem Persistenzkontext herausgelöste Objekte. Passende Datenbankeinträge sind vorhanden, Objektänderungen werden aber nicht dorthin propagiert.

Durch Benutzung der Entity-Manager-Methoden wechseln Objekte ihren Zustand, wodurch sich der in Abbildung 3.5 dargestellte Lebenszyklus der Entity-Objekte ergibt.

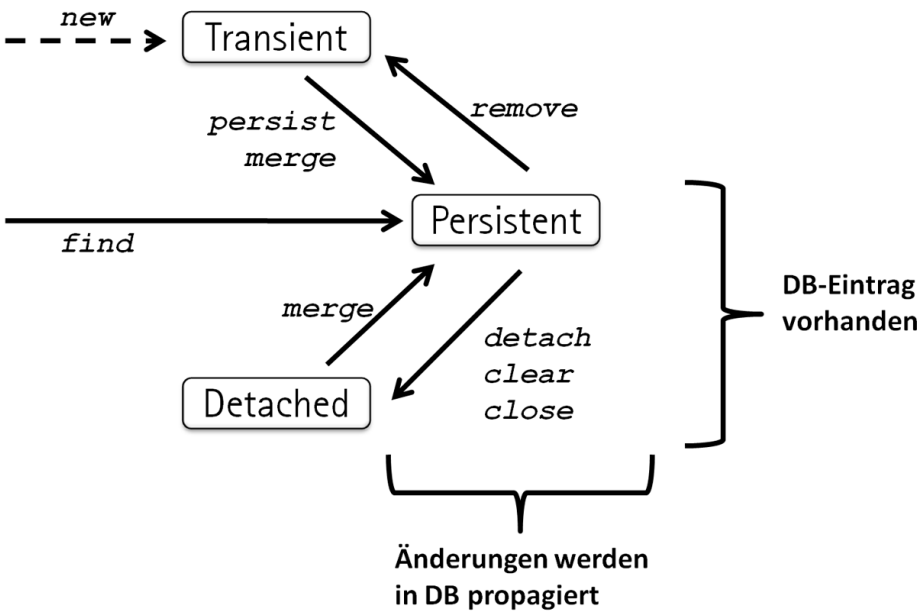

Abbildung 3.5: Entity-Lebenszyklus

Einige Statusübergänge sind aus Übersichtlichkeitsgründen weggelassen worden. So dürfen bspw. *persist* und *merge* auch auf bereits persistente Objekte angewandt werden. Dabei passiert bis auf die später erläuterte Kaskadierung von Operationen nichts.

Eine genaue Beschreibung der Übergänge findet sich in der JPA-Spezifikation im Abschnitt „Entity Instance's Life Cycle". Dort wird zudem noch der Zustand *Removed* beschrieben, der hier – ebenfalls der Einfachheit halber – mit *Transient* vermischt wurde.

3.2.6 Mapping-Annotationen für einfache Objekte

Die bisherigen Beispiele haben weitgehend von den Default-Einstellungen Gebrauch gemacht. Lassen Sie uns nun einen Blick auf weitere Möglichkeiten des Persistenz-Mappings einer Java-Klasse werfen. Hier soll allerdings nicht der entsprechende Abschnitt der JPA-Spezifikation dupliziert werden, daher beschränkt sich das Folgende auf die häufig genutzten Parameter. Weitere Details finden Sie im Abschnitt „Metadata for Object/Relational Mapping" der Spezifikation.

- *@Entity*

Die auf Klassenebene angewendete Annotation *@Entity* akzeptiert als Parameter den sog. Entity-Namen. Er wird später benötigt, um Abfragen mithilfe von JPQL formulieren zu können. Wird er nicht angegeben, gilt der einfache Klassenname als Entity-Name.

- *@Table*

Ebenfalls auf Klassenebene kann die Annotation *@Table* verwendet werden. Mithilfe des Parameters *name* (und bei Bedarf *catalog* und *schema*) kann der Name der Tabelle in der Datenbank bestimmt werden. Die Voreinstellung dafür ist der Entity-Name. Der Parameter *uniqueConstraints* erlaubt – wie der Name schon sagt – die Angabe von Unique Constraints. Er wirkt sich aber nur aus, wenn der Provider die Tabelle anlegt, wenn also bspw. Hibernate wie oben erwähnt mit *hibernate.hbm2ddl.auto=update* konfiguriert ist. Zudem ist hier zu beachten, dass die Contraints auf Basis der Spaltennamen angelegt werden, nicht etwa auf Basis der Java-Attributnamen. Mit JPA 2.1 ist der Parameter *indexes* hinzugekommen, mit dem zusätzliche Performanceindexe angelegt werden können.

Listing 3.10 zeigt, wie die schon mehrfach bemühte Beispielklasse mit einem Entity-Namen versehen wird – der allerdings dem Vorgabewert entspricht – und wie die zugehörige Tabelle explizit benannt und mit je einem Unique Contraint und einem Index belegt wird.

```
@Entity(name="Country")
@Access(AccessType.FIELD)
@Table(name = "EEDEMOS_COUNTRY",
  uniqueConstraints = @UniqueConstraint(columnNames = "NAME"),
  indexes = @Index(columnList = " PHONE_PREFIX "))
public class Country
{
  …
```

Listing 3.10: Anwendung von „@Entity“ und „@Table“

Ist es sinnvoll, Entity-Namen, Tabellennamen (und später auch Spaltennamen) explizit vorzugeben oder sollte man auf den Default Entity Name = einfacher Klassenname = Tabellenname vertrauen? Nun, das kommt drauf an … Zum einen sparen die Vorgabewerte natürlich Tipparbeit und man darf sich durchaus eine gewisse, sinnvolle Grundfaulheit aneignen, um seine Produktivität zu steigern. Auf der anderen Seite kann man mit den Default-Werten recht leicht in Refactoring-Fallen tappen. Stellen Sie sich z. B. vor, dass der Name der Beispielklasse von *Country* nach *Land* geändert werden soll, weil die allgemeinen Entwicklungsrichtlinien seit Neuestem deutsche Namen für Identifier (pardon: Identifizierer) verlangen. So ein Refactoring ist in den aktuellen IDEs ein Klacks, aber Achtung: Bei Verwendung der Defaults müssen die Query-Texte und der Tabellenname in der Datenbank passend verändert werden! Das würde sich mit expliziten Angaben vermeiden lassen.

Die im Weiteren besprochenen Mapping-Annotationen gelten für die Attributebene – also bei Field Access für die Instanzvariablen und bei Property Access für die Getter.

- *@Column*

Hiermit wird analog zu *@Table* die Angabe von Eigenschaften der Tabellenspalte ermöglicht. Neben dem Parameter *name*, der den Spaltennamen annimmt, akzeptiert *@Column* diverse Parameter zur Beeinflussung der Spaltendeklaration, die sich allerdings wieder nur dann auswirken, wenn der Provider die Tabelle in der Datenbank erzeugt. Details finden Sie in der JPA-Spezifikation.

Der Boole'sche Parameter *unique* erlaubt die Anlage von Unique Contraints zusätzlich zu denen auf Tabellenebene.

Die ebenfalls Boole'schen Parameter *insertable* und *updatable* lassen gewisse „Tricksereien" zu: Stehen diese Werte auf *false*, wird das betroffene Attribut nicht in *INSERT*- bzw. *UPDATE*-Statements verwendet. Das lässt sich z. B. nutzen, um mehrere Attribute auf die gleiche Tabellenspalte abzubilden, von denen dann bei Schreibzugriffen nur eines verwendet wird. Anwendungen dazu finden sich bspw. bei den später besprochenen Relationen, um den dabei implizit vorhandenen Foreign Key separat (und nur zum Lesen) zur Verfügung zu stellen.

Schließlich ermöglicht der Parameter *table* die Platzierung der Spalte in eine Secondary Table. Die Beschreibung dazu folgt später.

- *@Enumerated*

Für Aufzählungstypen gibt es zwei Möglichkeiten der Ablage der Werte in der Datenbank. Zum einen kann der numerische Ordnungswert verwendet werden, also *0* für die erste Konstante im Aufzählungstyp, *1* für die zweite usw. Die zweite Möglichkeit legt den Namen der abzuspeichernden Konstanten als Text in der Datenbank ab. Zur Auswahl übergibt man *@Enumerated* eine Konstante aus der Aufzählung *EnumType*: *ORDINAL* bzw. *STRING*. Voreingestellt ist *ORDINAL*.

Das numerische Verfahren birgt eine große Gefahr, und zwar wieder einmal bei einem Refactoring. Nehmen wir an, wir haben einen Aufzählungstyp *AmpelStatus* wie in Listing 3.11. Unser Verkehrsleitsystem wird um 3:00 abgeschaltet, als alle Ampeln gerade auf *ROT* stehen. Das Softwareupdate, das wir dann aktivieren, enthält u. a. die refaktorierte Klasse *AmpelStatus*, in der die Farbkonstanten aus irgendeinem Grund in die Reihenfolge *ROT*, *GELB*, *GRUEN* gebracht worden. Die ehemals roten Ampeln sind nun plötzlich grün (Ordnungswert *2*). Hoffentlich geht das gut …

```
public enum AmpelStatus
{
  GRUEN,
  GELB,
  ROT
}


@Entity
public class Verkehrsleitsystem
{
  …
  private AmpelStatus ampel1;
  private AmpelStatus ampel2;
  …
```

Listing 3.11: Aufzählungstyp und seine Verwendung in einer Entity

Seit JPA 2.1 kann anstelle von *@Enumerated* ein selbstentwickelter Konverter verwendet werden (s. Abschnitt 3.2.7 Custom Converter).

- *@Lob*

Je nach dem unterliegenden Datentyp lassen sich hiermit BLOBs und CLOBs deklarieren. Ist das betroffene Attribut vom Typ *char[]*, *Character[]* oder *String*, wird der Wert in der Datenbank als CLOB behandelt. Andere Typen werden als BLOB verarbeitet.

- *@Transient*

Die Voreinstellung ist so, dass alle Attribute der Entity-Klasse persistent sind. Will man davon abweichen, muss das betroffene Attribut mit *@Transient* markiert werden. Es ist dann nicht Teil der Datenbanktabelle, wird dann also auch von allen entsprechenden Operationen ignoriert.

- *@Temporal*

Beim Mapping eines Attributs vom Typ *java.util.Date* oder *java.util.Calendar* sind mehrere Abbildungen auf die Datenbank möglich: Ablage des kompletten Werts inkl. Datum und Uhrzeit oder Speicherung von Datum bzw. Uhrzeit allein. Das gewünschte Verfahren muss durch Annotation des Attributs mit *@Temporal* gewählt werden. Zur Auswahl akzeptiert *@Temporal* einen Wert aus dem Aufzählungstyp *TemporalType*: *TIMESTAMP*, *DATE* bzw. *TIME*. *TIMESTAMP* ist der Vorgabewert.

Seit JPA 2.1 kann anstelle von *@Temporal* ein selbstentwickelter Konverter verwendet werden (s. Abschnitt 3.2.7 Custom Converter).

Bei der Arbeit mit Zeiten ist zu beachten, dass die Granularität der Werte nicht mit der der Datenbankspalte übereinstimmen muss. Während die Java-Werte bis zur Millisekunde genau sind, speichert die Datenbank ggf. gröber oder feiner ab.

Nachdem wir nun die einfachen Attribute betrachtet haben, können wir uns langsam an etwas komplexere Dinge wagen: Attribute, die aus mehreren Teilelementen zusammengesetzt sind.

- *@Embeddable*

Wir sind es gewohnt, Objekte aus anderen Objekten zu komponieren, um so für eine übersichtliche Struktur zu sorgen und Datentypen ggf. auch wiederverwerten zu können. Listing 3.12 zeigt eine solche Struktur: Der Datentyp *Address*, bestehend aus den üblichen Adressfeldern, wird als Typ eines Attributs in *Employee* verwendet.

Damit das nach Wunsch funktioniert, muss *Address* mit *@Embeddable* annotiert sein. *Address* muss (wie zuvor die Entities) einen parameterlosen Konstruktor vorweisen und kann mit den bereits besprochenen Annotationen weiter parametrisiert werden. Im Ergebnis ergibt sich daraus ein Mapping der Klasse *Employee* auf eine Tabelle, die für das Attribut *address* drei Spalten enthält (Abbildung 3.6).

Die Spaltennamen ergeben sich dabei aus den Vorgaben oder Annotationen der eingebetteten Klasse, im Beispiel sind dies also *zipCode*, *town* und *street*. Das lässt sich an der einbettenden Stelle mithilfe der Annotationen *@AttributeOverrides* und *@AttributeOverride* erreichen, siehe Listing 3.13 – eine zugegebenermaßen grenzwertige Konstruktion im Hinblick auf die Lesbarkeit des Programmcodes …

```
@Embeddable
@Access(AccessType.FIELD)
public class Address
{
  private String zipCode;
  private String town;
  private String street;
  …
}

@Entity
@Access(AccessType.FIELD)
public class Employee
{
  @Id
  private int    id;
  private String  name;

  private Address address;
  …
```

Listing 3.12: Eingebettetes Objekt

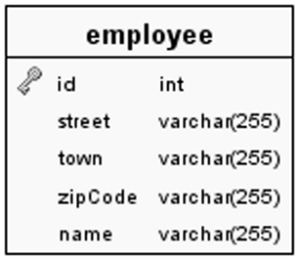

Abbildung 3.6: „Employee"-Tabelle mit eingebetteter „Address"

```
@Entity
@Access(AccessType.FIELD)
public class Employee
{
  …
  @AttributeOverrides({
    @AttributeOverride(name = "zipCode",
                       column = @Column(name = "homeZipCode")),
```

```
    @AttributeOverride(name = "town",
                       column = @Column(name = "homeTown")),
    @AttributeOverride(name = "street",
                       column = @Column(name = "homeStreet")) })
  private Address homeAddress;
  …
```

Listing 3.13: Explizite Angabe von Spaltennamen bei einem eingebetteten Attribut

- *@ElementCollection*

Diese Annotation kann einerseits auf Attribute vom Typ *java.util.Collection* oder einem davon abgeleiteten Interface angewendet werden. Der Elementtyp der *Collection* kann einfacher Datentyp sein oder ein *@Embeddable*. Die Daten der *Collection* werden in einer separaten Tabelle abgelegt.

Ganz ähnlich funktioniert das auch mit Attributen vom Typ *java.util.Map*. Die Basistypen der *Map* können wieder einfache oder *@Embeddables* sein.

Listing 3.14 zeigt je eine Anwendung von *@ElementCollection*. Die resultierende Tabellenstruktur zeigt Abbildung 3.7.

```
@Entity
@Access(AccessType.FIELD)
public class Employee
{
  …
  @ElementCollection
  private List<String> skills;

  @ElementCollection
  private Map<String, String> phones;
  …
```

Listing 3.14: Collection-Attribute

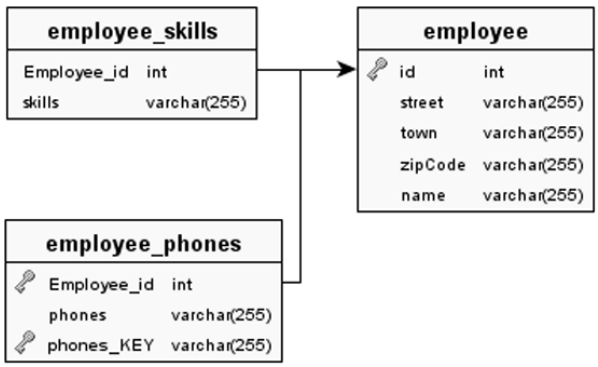

Abbildung 3.7: Ablage von Collection-Attributen in zusätzlichen Tabellen

Der Name der Zusatztabelle ergibt sich standardmäßig als Verkettung des Namens der Haupt-Entity, einem "_" und dem Namen des *Collection*- bzw. *Map*-Attributs. Das kann aber mithilfe der Annotation *@CollectionTable* übersteuert werden. Damit können zudem die Namen der Fremdschlüssel der Zusatztabelle bestimmt werden, die per Default aus dem Zielentitynamen, "_" und dem Namen des referenzierten Attributs zusammengesetzt werden. Auch die restlichen Spaltennamen der Zusatztabelle können explizit angegeben werden. Dazu dienen die Annotationen *@Column* und *@MapKeyColumn* im Fall einfacher Elementtypen sowie *@AttributOverrides* und *@AttributeOverride* im Fall von *@Embeddable*-Elementen. Die Details dazu finden Sie in der JPA-Spezifikation.

Generell ist zu beachten, dass *@ElementCollection* nur auf Attribute der Interfacetypen aus *java.util* angewandt werden darf. Es darf also statt *List* nicht etwa *ArrayList* benutzt werden. Außerdem empfiehlt sich unbedingt die Nutzung der parametrisierten Typen *java.util.Collection<E>*, da dann der oder die Elementtypen bekannt sind. Bei Verwendung der Raw Types sind zusätzliche Annotationen notwendig, was die Lesbarkeit deutlich verschlechtert.

3.2.7 Custom Converter

Bis JPA 2.0 musste man sich i. W. mit dem Mapping zwischen Java-Attributen und Datenbankspalten zufrieden geben, das auf recht niedriger Ebene, nämlich im JDBC-Standard, definiert wird. Mit den in der Version 2.1 eingeführten Konvertern kann man das Mapping nun sehr flexibel beeinflussen.

Dazu wird eine Konverterklasse benötigt, die mit ihren beiden Methoden für die Umwandlung eines Java-Atributs in einen Spaltenwert und umgekehrt sorgt. Sie muss mit *@Converter* annotiert sein und das Interface *AttributeConverter<X,Y>* implementieren. Dabei ist *X* der Attributtyp und *Y* der Typ des Datenbankfelds. Das zu konvertierende Attribut wird mit *@Convert* annotiert und die Converter Class als Parameter angegeben (Listing 3.15).

```
@Converter
public class YesNoConverter
  implements AttributeConverter<Boolean, String>
{
  public String convertToDatabaseColumn(Boolean fieldValue) { … }
  public Boolean convertToEntityAttribute(String columnValue) { … }
}

@Entity
public class Country
{
  …
  @Convert(converter = YesNoConverter.class)
  private boolean expired;
```

Listing 3.15: Konverter

Die Converter Class kann als Standardkonverter registriert werden, indem der Annotation *@Converter* der Parameter *autoApply=true* mitgegeben wird. Ein solcher Konverter ist automatisch für alle Attribute passenden Typs aktiv. Ausgenommen sind nur ID- und Versionsattribute[7] sowie Attribute, die explizit mit *@Convert*, *@Enumerated* oder *@Temporal* annotiert sind (Listing 3.16).

```
@Converter(autoApply = true)
public class ContinentConverter
  implements AttributeConverter<Continent, String>
{
  …
}

@Entity
public class Country
{
  …
  // Nutzt implizit ContinentConverter
  private Continent continent;
```

Listing 3.16: Standardkonverter

Die Annotation *@Convert* akzeptiert den Parameter *disableConversion*. Wird dafür *false* übergeben, findet keine implizite Konvertierung für das damit annotierte Attribut statt.

Neben der Anwendung für einfache Attribute kann @Convert auch für Klassen und Collections genutzt werden. Für Details dazu sei auf die Spezifikation verwiesen („Convert Annotation“).

3.2.8 Generierte IDs

Die bereits mehrfach angeführte Beispielklasse *Country* hat eine fachliche Identität. Das Attribut *isoCode* enthält den international genormten 2-Zeichen-Code, der das entsprechende Land eindeutig identifiziert. Diese Situation finden wir bei Weitem nicht immer vor. Es ist im Gegenteil so, dass aktuelle Softwaredesigns eher technische Identitäten favorisieren. Der Hintergrund ist der, dass man bei fachlichen Identitäten auf längere Sicht nicht sicher sein kann, ob sie wirklich unveränderlich sind. Zudem sind fachliche Attribute oft nur in Kombinationen eindeutig, sodass man sich schnell mit zusammengesetzten IDs konfrontiert sieht.

An eine technische ID haben wir nur den Anspruch, dass sie eindeutig ist. Sie hat keine fachliche Bedeutung, kann also generiert werden. Dies ist mit JPA in einfacher Weise möglich, wenn die Entity ein einzelnes ID-Attribut besitzt. Dieses wird zusätzlich zu *@Id* mit *@GeneratedValue annotiert*. Der JPA-Provider vergibt dann automatisch für jedes neu abgespeicherte Objekt eine neue, eindeutige ID (Listing 3.17).

7 Versionsattribute werden später im Abschnitt 3.6.4 über Locking diskutiert

```
@Entity
@Access(AccessType.FIELD)
public class City
{
  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  private Integer id;
  …
```

Listing 3.17: Generierte ID

Zur Generierung der ID stehen drei verschiedene Generatoren zur Auswahl, die mithilfe des Parameters *strategy* ausgewählt werden. Folgende Werte stehen dafür zur Verfügung:

- *GenerationType.IDENTITY*

Bei dieser Strategie wird die Vergabe einer neuen, eindeutigen ID durch die Datenbank vorgenommen. Voraussetzung ist natürlich, dass die eingesetzte DB dies auch unterstützt. So bieten bspw. Derby und MySQL Autoincrement Columns an; MS SQL Server kennt analog Identity Columns. Bei einem *INSERT* weist die Datenbank dem betroffenen Spaltenwert automatisch die nächste freie Nummer zu.

Ein starker Vorteil dieser Strategie ist, dass auch andere Anwendungen, die in die gleiche DB schreiben, ohne weiteres Zutun das gleiche Verfahren nutzen. Nachteilig ist allerdings, dass der generierte Wert erst zum Zeitpunkt der Ausführung des *INSERT*-Statements ermittelt werden kann und er zudem durch einen weiteren DB-Zugriff wieder ausgelesen werden muss, um das Java-Objekt im Speicher entsprechend zu aktualisieren. Das kann je nach Datenbank recht inperformant sein.

- *GenerationType.SEQUENCE*

Hier wird eine Sequence der Datenbank genutzt und damit wiederum ein Konstrukt, das von der Zieldatenbank angeboten werden muss. Dies ist z. B. für Oracle oder PostgreSQL der Fall. Die Sequence liefert bei jedem Zugriff eine neue Zahl, die vom Provider zur Besetzung der ID des neu erzeugten Entity-Objekts verwendet wird.

Vorteilhaft ist hier die Entkopplung vom späteren Einfügen des Satzes in die Datenbank, der ID-Wert steht also schon zum Zeitpunkt des Aufrufs von *persist* zur Verfügung. Außerdem können Datenbanken i. d. R. mit Sequenzen gut optimiert umgehen, benötigen also bspw. auch bei nebenläufigen Zugriffen nicht unbedingt eine Transaktion.

Der Sequence-Generator lässt sich noch weiter parametrisieren, indem man der Annotation *@GeneratedValue* mit dem Parameter *generator* den Namen eines spezifischen Generators mitgibt und diesen mit der zusätzlichen Annotation *@SequenceGenerator* deklariert. Damit wird es möglich, auf die Benennung der Sequence in der Datenbank und ihren Startwert Einfluss zu nehmen (Listing 3.18).

```
@Id
@GeneratedValue(strategy = GenerationType.SEQUENCE,
```

```
                generator = "cityGenerator")
@SequenceGenerator(name = "cityGenerator",
                   sequenceName = "CITY_GEN", initialValue = 200000)
private Integer id;
```

Listing 3.18: Konfiguration des Sequence-Generators

- *GenerationType.TABLE*

Der Table-Generator lässt sich mit jeder Datenbank nutzen. Der Provider verwendet eine eigene Tabelle zur Nachbildung einer Sequenz. Im Vergleich zum vorigen Verfahren hat dieses hier den (kleinen) Nachteil, dass kein speziell für die Sequenzerzeugung optimiertes Verfahren der Datenbank verwendet wird. Dafür ist dies das einzige portable Verfahren.

Auch der Table-Generator lässt sich mit weiteren Parametern ähnlich zu denen des Sequence-Generators ausstatten, und zwar mithilfe der Annotation *@TableGenerator* (Listing 3.19).

```
@Id
@GeneratedValue(strategy = GenerationType.TABLE,
                generator = "cityGenerator")
@TableGenerator(name = "cityGenerator",
                table = "CITY_GEN",
                pkColumnName = "GENERATOR",
                valueColumnName = "NEXT_ID",
                initialValue = 200000)
private Integer id;
```

Listing 3.19: Konfiguration des Table-Generators

- *GenerationType.AUTO*

Dieser Modus stellt keinen weiteren Generator dar. Vielmehr wird hier dem Provider die Auswahl des vermeintlich günstigsten Generators überlassen. Die meisten Provider lassen die Wahl auf den Table-Generator fallen, was insofern verständlich ist, da dieser Modus mit jeder Datenbank verfügbar ist. Nach meiner Beobachtung ist Hibernate der einzige Provider, der je nach Zieldatenbank unterschiedliche Generatoren nutzt. Der Identity-Generator kommt zum Einsatz, wenn die Datenbank dieses Verfahren unterstützt. Falls das nicht so ist, dafür aber Sequenzen zur Verfügung stehen, wird der Sequence-Generator gewählt. Wenn auch das nicht möglich ist, nutzt Hibernate den Table-Generator.

Der Automatikmodus ist voreingestellt.

Dem Provider kann erlaubt werden, mit einem Generator-Zugriff direkt mehrere neue IDs zu besorgen und nach und nach zu verbrauchen. Diese sog. Allocation Size kann beim Sequence- und beim Table-Generator konfiguriert werden, und zwar mithilfe des Parameters *allocationSize* der Annotationen *@SequenceGenerator* und *@TableGenerator*. Der

Vorgabewert ist lt. Spezifikation *50*, was aber nach meiner Beobachtung nicht von allen Providern eingehalten wird. Eine große Allocation Size wirkt sich günstig auf die Einfügeperformanz aus, wie man der Abbildung 3.8 entnehmen kann. Dort ist die Laufzeit für das Einfügen von 50 000 neuen Einträgen in eine Datenbank für die *allocationSize* 1, 10, 100 und 1 000 qualitativ darstellt. Bei einer Allocation Size größer als 1 bleiben allerdings potenziell generierte Werte ungenutzt.

Für den Identity-Generator gibt es leider keine Möglichkeit, IDs auf Vorrat zu besorgen. Hier müssen die generierten Werte immer sofort zurückgelesen werden.

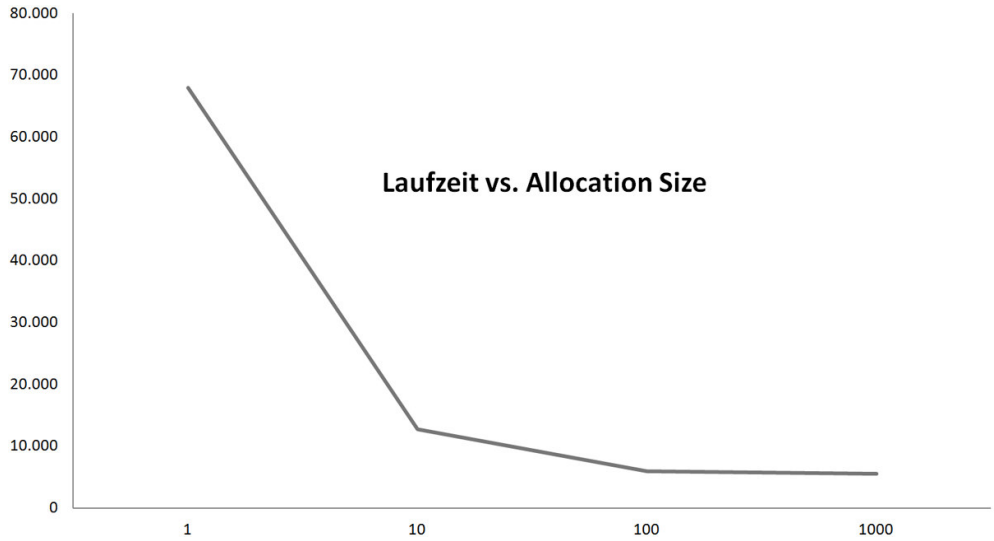

Abbildung 3.8: Einfluss der Allocation Size auf die Laufzeit von Einfügeoperationen

In der aktuellen Version des Standards stehen nur die beschriebenen Generatoren zur Verfügung. Sie benötigen alle ein ganzzahliges ID-Attribut. Die JPA-Spezifikation gibt nicht exakt Auskunft über die verwendbaren Datentypen, enthält aber Beispiele mit *int*, *Integer*, *long* und *Long*.

3.2.9 Objektgleichheit

Die Gleichheit von Java-Objekten wird mithilfe der Methode *equals* definiert. Eng damit verbunden ist *hashCode*: Ist *a.equals(b) true*, muss *a.hashCode()* den gleichen Wert liefern wie *b.hashCode()*.

Es ist eine häufig anzutreffende und gute Programmierkonvention, dass *equals* und *hashCode* in jeder Klasse definiert werden müssen – von einigen technischen Klassen oder Hilfsklassen vielleicht mal abgesehen.

Wie sollten diese Methoden nun für Entity-Klassen programmiert werden? Java Persistence macht da prinzipiell keine Vorgabe, dennoch lassen sich aus dem Ablagekonzept einer Datenbank Rückschlüsse auf die Objektgleichheit auf der Java-Seite ziehen: In der Datenbank definiert der Primärschlüssel eine Art Gleichheit innerhalb der Einträge einer Tabelle. Vor dem Hintergrund ist es sinnvoll, *equals* auf Basis des ID-Attributs zu definieren, also beim Vergleich zweier Entity-Objekte nicht alle Attribute paarweise zu vergleichen, sondern sich auf das ID-Attribut zu beschränken (bzw. die ID-Attribute, wenn es mehrere sein sollten). Für die Country-Klasse könnten *equals* und *hashCode* also wie in Listing 3.20 definiert werden. Der Aufwand zur Erstellung dieser Methoden ist gering. Bei Nutzung einer aktuellen IDE sind dazu nur wenige Mausklicks nötig.

```
@Entity
@Access(AccessType.FIELD)
public class Country
{
  @Id
  private String isoCode;
  …
  public boolean equals(Object obj)
  {
    if (this == obj)
    {
      return true;
    }
    if (obj == null)
    {
      return false;
    }
    if (getClass() != obj.getClass())
    {
      return false;
    }
    final Country other = (Country) obj;
    return this.isoCode.equals(other.isoCode);
  }

  public int hashCode()
  {
    return this.isoCode.hashCode();
  }
  …
```

Listing 3.20: „equals“ und „hashCode“

Dieses Vorgehen lässt sich natürlich auch bei Klassen mit technischen ID-Attributen anwenden, allerdings gibt es da eine Schwierigkeit, wenn die Attributwerte generiert werden: Die ID-Werte sind frühestens nach dem Aufruf von *persist* gesetzt, evtl. auch erst später, wenn der *INSERT* in die Datenbank erfolgt. Ein Vergleich zweier Entity-Objekte

wird also potenziell unterschiedlich ausfallen, wenn er vor oder nach dem Einfügen in die DB erfolgt. Schlimmer noch: Alle noch nicht persistenten Objekte einer Klasse haben ihre ID-Attribute noch nicht gesetzt, d. h. diese sind bspw. alle *null*. Im Sinne einer „normal" programmierten *equals*-Methode wären diese Objekte dann alle gleich – eine Situation, die vermutlich nicht so ganz im Sinne des Erfinders wäre …

Abhilfe lässt sich auf verschiedene Weise schaffen, wobei es allerdings keinen Königsweg gibt. In jedem Fall wird die Information benötigt, ob die ID des betroffenen Objekts bereits gesetzt worden ist. Im Allgemeinen lässt sich das mithilfe der Methode *getIdentifier* aus dem Interface *PersistenceUnitUtil* herausfinden. Sie liefert die ID für das übergebene Entity-Objekt zurück. Ist diese noch nicht gesetzt, ist der Return-Wert *null*. Ein Objekt vom Typ *PersistenceUnitUtil* kann über zwei Ecken aus dem Entity Manager heraus geliefert werden. Man könnte sich also eine Helfermethode wie die in Listing 3.21 bereitstellen, um den ID-Zustand eines Objekts erfragen zu können. Noch einfacher geht es aber, wenn man für die ID-Attribute keine primitiven Datentypen wählt, sondern objektorientierte, statt *int* also bspw. *Integer*. In dem Fall ist ein noch nicht gesetztes Attribut am Wert *null* zu erkennen. Das hat zudem den Vorteil, dass für die Entscheidung kein Entity Manager benötigt wird, auf den man innerhalb einer Entity-Methode auf einfache Weise keinen Zugriff hat.

```
public boolean isIdSet(Object entity)
{
  PersistenceUnitUtil persistenceUnitUtil
    = entityManager.getEntityManagerFactory().getPersistenceUnitUtil();
  Object id = persistenceUnitUtil.getIdentifier(entity);
  return id != null;
}
```

Listing 3.21: Helfermethode zur Ermittlung, ob die ID eines Objekts gesetzt ist

Nach diesen Vorüberlegungen lässt sich *equals* nun so formulieren, dass für noch nicht mit einer ID versehene Objekte *false* geliefert wird. Der Hashcode solcher Objekte kann z. B. mit *0* angenommen werden (Listing 3.22). Es sei an dieser Stelle darauf hingewiesen, dass dieses Verfahren nicht absolut wasserdicht ist, denn *equals* und *hashCode* werden vor und nach dem Persistieren der Objekte unterschiedliche Ergebnisse liefern. Das ist nur dann tolerierbar, wenn gewährleistet ist, dass frisch persistierte Objekte im weiteren Verlauf des Programms nicht mehr unmittelbar benötigt, vor ihrer weiteren Verwendung also neu geladen werden. Das wird bei den allermeisten Geschäftsprozessen so sein.

```
@Entity
@Access(AccessType.FIELD)
public class City
{
  @Id
  @GeneratedValue
  private Integer id;
```

```
…
public boolean equals(Object obj)
{
  if (this == obj)
  {
    return true;
  }
  if (obj == null)
  {
    return false;
  }
  if (getClass() != obj.getClass())
  {
    return false;
  }
  final City other = (City) obj;
  return this.id != null && this.id.equals(other.id);
}

public int hashCode()
{
  return this.id != null ? this.id.hashCode() : 0;
}
…
```

Listing 3.22: „equals“ und „hashCode“ bei generierter ID

Auf der sicheren Seite wäre man, wenn man *equals* und *hashCode* im Falle einer noch nicht besetzten ID durch Auswurf einer passenden Exception (*IllegalArgumentException* o. ä.) abbräche. Leider ist das kein wirklich gangbarer Weg, da angrenzende Frameworks (z. B. Bean Validation – siehe gleichnamiges Kapitel) für einige Operationen darauf angewiesen sind, *equals* und *hashCode* schon für transiente Objekte aufrufen zu können.

In der Literatur und in Forenbeiträgen findet man häufig auch die Meinung, man solle Entity-Objekte zusätzlich zur technischen Identität mit einer fachlichen versehen, die man solange für Vergleiche benutzt, bis die technische ID gesetzt ist. Es ist schwer, sich dem anzuschließen, da damit der Sinn einer technischen Identität weitgehend ad absurdum geführt wird. Zudem wäre der Entwicklungsaufwand unvertretbar höher.

3.2.10 Basisklassen für Entity-übergreifende Aspekte

Die bislang betrachteten Entity-Klassen weisen einige Gemeinsamkeiten auf: Es gibt ein einzelnes ID-Attribut und die Methoden *hashCode*, *equals* sowie ggf. *toString* sind prinzipiell gleich implementiert. Zudem ist Serialisierbarkeit häufig nützlich oder sogar notwendig. Diese eher technischen Aspekte lassen sich sehr gut in eine Basisklasse auslagern (Listing 3.23).

```
public abstract class SingleIdEntity<K> implements Serializable
{
  public abstract K getId();

  public int hashCode()
  {
    K thisId = getId();
    return thisId != null ? thisId.hashCode() : 0;
  }

  public boolean equals(Object obj)
  {
    …
  }
  …
}

@Entity
@Access(AccessType.FIELD)
public class Country extends SingleIdEntity<String>
{
```

Listing 3.23: Auslagerung technischer Aspekte in eine Basisklasse

In gleicher Weise können natürlich auch ID-Generatoren in Basisklassen ausgelagert werden (Listing 3.24). Hier ist die Annotation der Basisklasse mit *@MappedSuperclass* wichtig. Sie sorgt dafür, dass die Attribute der Basisklasse in das Mapping auf die DB-Tabelle einbezogen werden.

```
@MappedSuperclass
@Access(AccessType.FIELD)
public abstract class GeneratedIntegerIdEntity
  extends SingleIdEntity<Integer>
{
  @Id
  @GeneratedValue(strategy = GenerationType.TABLE,
                  generator = "IntegerIdGenerator")
  @TableGenerator(name = "IntegerIdGenerator",
                  allocationSize = 100,
                  table = "ID_GENERATOR",
                  pkColumnName = "GEN_NAME",
                  pkColumnValue = "IntegerIdGenerator",
                  valueColumnName = "GEN_VALUE")
  protected Integer id;

  public Integer getId()
  {
    return this.id;
  }
}
```

Listing 3.24: Auslagerung eines ID-Generators in eine Basisklasse

3.3 Objektrelationen

Die bisher dargestellten Dinge betrafen einfache Entity-Klassen, die keinerlei Beziehungen zu anderen Entities haben. Im Folgenden wenden wir uns den Relationen zwischen Entities zu. Eine Beziehung drückt sich in der Datenbank durch einen Foreign Key aus, d. h. eine Spalte (oder auch mehrere), deren Wert auf einen Eintrag einer anderen Tabelle referenziert. Auf der Ebene der Java-Klassen werden Beziehungen durch Entity-Attribute ausgedrückt, die ein oder mehrere Objekte der Gegenseite enthalten. Damit wird ein konzeptioneller Unterschied dieser beiden Sichten deutlich: Während es in der Datenbank eine Stelle gibt, die die Beziehung definiert, sind es potenziell zwei Java-Klassen, die Beziehungsattribute enthalten können. Vor der Betrachtung der Details sind drei grundlegende Begriffe in Bezug auf Relationen zu klären.

- Ownership

Eine der beiden an einer Relation beteiligten Entity-Klassen hat bezüglich der Beziehung „den Hut auf". Das Relationsattribut auf ihrer Seite ist für den Wert des Foreign Keys in der Datenbank maßgeblich. Diese Entity ist der Eigentümer der Relation.

- Kardinalität

Diese Eigenschaft ist Ihnen von der Datenbankseite her bekannt: Es können verschieden viele Einträge der einen Seite mit einer unterschiedlichen Anzahl von Sätzen der anderen Seite verknüpft sein. In diesem Sinne kennen wir 1:1-, 1:n- und n:m-Beziehungen. Auf der Java-Seite bestimmt die Kardinalität, ob das Relationsattribut den Typ der Gegenseite hat oder eine *Collection* darüber ist.

- Direktionalität

Ist in einer Entity-Klasse ein Relationsattribut vorhanden, enthält es das bzw. die durch die Relation referenzierte(n) Element(e). Man kann also von einem Objekt der Klasse durch Benutzung des Attributs sozusagen zur Gegenseite hinübernavigieren. Die Existenz der Relationsattribute auf beiden Seiten der Beziehung ist Ausdruck der Direktionalität: Bei unidirektionalen Relationen ist nur auf der Owner-Seite ein Relationsattribut vorhanden, bei bidirektionalen auf beiden Seiten.

Kombiniert man diese drei grundlegenden Eigenschaften von Relationen, erhält man aufgrund der Symmetrie einiger Kombinationen nicht 12, sondern nur 7 Fälle, die im Folgenden betrachtet werden.

3.3.1 Unidirektionale n:1-Relationen

Hier ist einem Objekt der Owner-Seite der Relation jeweils ein Objekt der Gegenseite zugeordnet. Umgekehrt können es mehrere sein. Da es eine unidirektionale Verbindung ist, enthält nur die Owner-Seite ein Relationsattribut. Es hat den Typ der Gegenseite und wird mit *@ManyToOne* annotiert. In Listing 3.25 ist als Beispiel die Verbindung zwischen

Flugzeugen und Flügen dargestellt. Ein Flugzeug absolviert mehrere Flüge, wobei hier angenommen wurde, dass im Flug das Flugzeug eingetragen wird, umgekehrt das Flugzeug aber nichts von seinen Flügen „weiß“.

```
@Entity
@Access(AccessType.FIELD)
public class AirCraft extends GeneratedIntegerIdEntity
{
  private String  maker;
  …
}

@Entity
@Access(AccessType.FIELD)
public class Flight extends GeneratedIntegerIdEntity
{
  private int     flightNo;

  @ManyToOne
  private AirCraft airCraft;
  …
}
```

Listing 3.25: Unidirektionale n:1-Beziehung

Die zugeordnete Struktur in der Datenbank zeigt Abbildung 3.9. Die Relation wird durch den Foreign Key *airCraft_id* in der Tabelle *flight* repräsentiert. Der Name der Spalte ergibt sich aus dem Namen des Relationsattributs, einem '_' und dem Namen des referenzierten Primärschlüsselattributs. Mithilfe der Annotation *@JoinColumn* kann ein anderer Name vorgegeben werden (Listing 3.26).

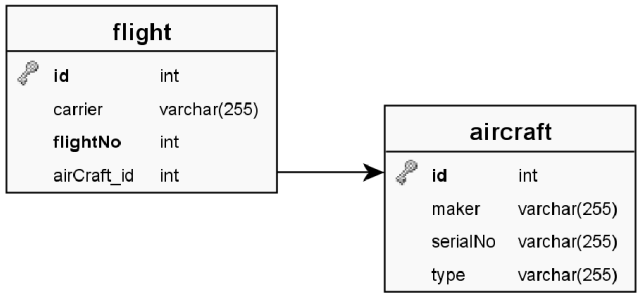

Abbildung 3.9: Tabellenstruktur zu Listing 3.25

```
@ManyToOne
@JoinColumn(name = "PLANE_ID")
private AirCraft airCraft;
```

Listing 3.26: Explizite Benennung des Fremdschlüssels

Die gezeigte Tabellenstruktur ist sicher das, was man regelmäßig für solche Relationen erwartet und was man „auf der grünen Wiese“ auch so anlegen würde. Übernimmt man dagegen eine bestehende Datenbankstruktur, ist man vielleicht mit einem Layout wie in Abbildung 3.10 konfrontiert. Hier ist die Beziehung über eine Verknüpfungstabelle abgebildet worden. So etwas lässt sich mit der Annotation *@JoinTable* erreichen (Listing 3.27). Die Parameter der Annotation sind optional. Wenn nicht angegeben, ergeben sich die Namen der Verknüpfungstabelle aus den Namen der verknüpften Tabellen mit einem '_' dazwischen. Die Fremdschlüssel werden als Vorgabe wie oben beschrieben benannt.

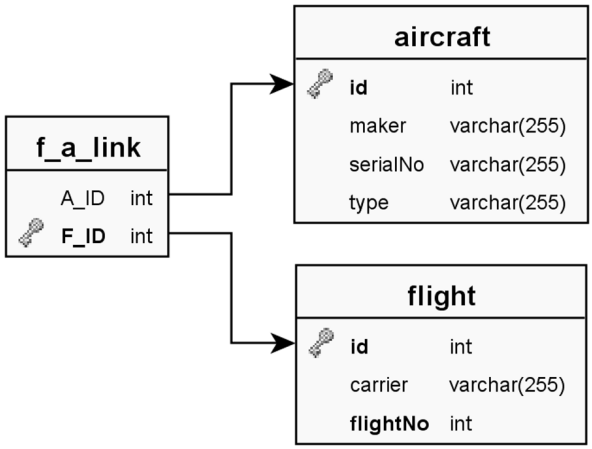

Abbildung 3.10: Alternatives Tabellenlayout mit Verknüpfungstabelle

```
@ManyToOne
@JoinTable(name = "F_A_LINK",
          joinColumns = { @JoinColumn(name = "F_ID") },
          inverseJoinColumns = { @JoinColumn(name = "A_ID") })
private AirCraft airCraft;
```

Listing 3.27: Mapping-Annotationen zu Abbildung 3.10

Die Benutzung der Objektstruktur zur Laufzeit ist trivial: Nach einem Einlesen eines *Flight*-Objekts z. B. per *find* enthält das darin befindliche Attribut *airCraft* direkt das zugeordnete *AirCraft*-Objekt. Umgekehrt führt eine Änderung des Attributs – bspw. die Zuweisung eines anderen *AirCraft*s – schließlich zum Eintrag des dazu passenden Werts in das Fremdschlüsselattribut bzw. in die Verknüpfungstabelle (Listing 3.28). Ein derart zugewiesenes Relationsobjekt muss bereits persistent sein oder separat persistent gemacht werden. Später werden Kaskadierungsoptionen vorgestellt, die diese Einschränkung aufheben.

Bei der Arbeit mit Relationsattributen ist es wichtig, in Objekten zu denken – die Fremdschlüsselbeziehung ist auf der Java-Ebene vollständig verdeckt. Man weist dem Relationsattribut ein komplettes Objekt zu, nicht etwa nur seine ID.

```
Flight flight = entityManager.find(Flight.class, id);
System.out.println("AirCraft: " + flight.getAirCraft());

AirCraft otherPlane = entityManager.find(AirCraft.class, somePlaneId);
flight.setAirCraft(otherPlane); // Wird am TX-Ende gespeichert
```

Listing 3.28: Nutzung der unidirektionalen n:1-Beziehung zur Laufzeit

Ob der Provider in dem Fall, dass er Tabellen anlegt, auch Foreign Key Constraints für Relationen erzeugt, ist ohne weiteres nicht festgelegt. Ab JPA 2.1 gibt es allerdings die Möglichkeit, mit dem Parameter *foreignKey* in *@JoinColumn* und ähnlichen Annotationen explizit ein Foreign Key Constraint anzufordern (Listing 3.29).

```
@ManyToOne
@JoinColumn(name = "PLANE_ID",
            foreignKey = @ForeignKey(ConstraintMode.CONSTRAINT))
private AirCraft airCraft;
```

Listing 3.29: Explizite Anforderung eines Foreign Key Constraints

Neben dem im Beispiel gezeigten Wert akzeptiert @ForeignKey weitere Parameter, um Foreign Key Constraints zu unterdrücken oder sogar ein SQL-Fragment dafür anzugeben. Für Details sei auf die Spezifikation verwiesen.

3.3.2 Unidirektionale 1:n-Relationen

Andersherum geht es auch. Die Vorgehensweise ist ähnlich, nur enthält das Relationsattribut nun mehrere Objekte der Gegenseite. Es muss dementsprechend eine *Collection* sein – nutzbar sind die Typen *java.util.Collection*, *java.util.List* und *java.util.Set*[8]. Die Relations-Annotation heißt nun umgekehrt wie im vorigen Fall: *@OneToMany* (Listing 3.30).

```
@Entity
@Access(AccessType.FIELD)
public class Person
{
  @Id @GeneratedValue
  private Integer          id;
  private String           name;

  @OneToMany
  private List<MailAddress> mailAddresses;
  …
}

@Entity
@Access(AccessType.FIELD)
```

8 *java.util.Map* ist auch erlaubt, aber für Relationsattribute kaum sinnvoll.

```
public class MailAddress
{
  @Id @GeneratedValue
  private Integer id;
  …
}
```

Listing 3.30: Unidirektionale 1:n-Beziehung

Für die Deklaration des Relationsattributs sollten unbedingt die generischen Typen genutzt werden sollten, da dann der Typ der Gegenseite daraus geschlossen werden kann. Bei Nutzung eines Raw Types müsste der referenzierte Typ als Parameter *targetEntity* der Annotation *@OneToMany* übergeben werden.

Bemerkenswerterweise wird im Standard-Mapping einer solchen unidirektionalen 1:n-Relation eine Verknüpfungstabelle verwendet. Soll stattdessen ein Fremdschlüssel auf der Many-Seite eingesetzt werden, muss dies explizit mit der Annotation *@JoinColumn* angefordert werden (Abbildung 3.11).

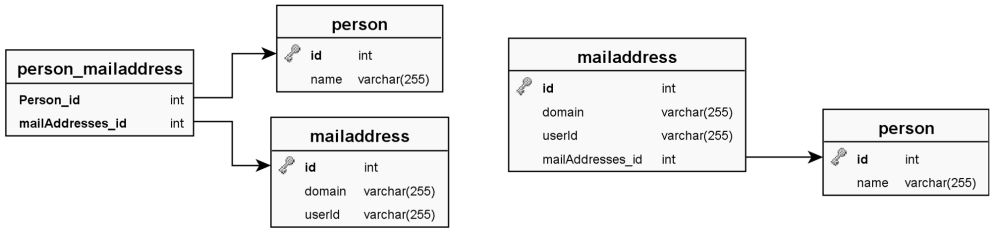

Abbildung 3.11: Tabellenlayout zu Listing 3.30 links ohne und rechts mit „@JoinColumn“

Gibt man *@JoinColumn* keinen Spaltennamen als Parameter mit, wird der Fremdschlüssel in der Tabelle wiederum aus dem Namen des Relationsattributs, einem '_' und dem Namen des referenzierten Primärschlüsselattributs zusammengesetzt. Da das Attribut im Java-Code auf der anderen Seite der Relation definiert ist, ergibt sich i. d. R. ein missverständlicher Spaltenname. So ist im gezeigten Beispiel *mailAddresses_id* ein eher unglücklich gewählter Name. Besser wäre hier eine explizite Angabe des Fremdschlüsselnamens, z. B. mit *@JoinColumn(name = "person_id")*.

Die Arbeit mit den so verknüpften Objekten zur Laufzeit des Programms ist genauso unspektakulär wie im entgegengesetzt gerichteten Fall: Beim Lesen eines Objekts der Owner-Seite (im Beispiel *Person*) ist die *Collection* der referenzierten Objekte (im Beispiel *MailAddress*) ohne weiteres nutzbar. Ob diese direkt mitgelesen wurde oder ob dazu zusätzliche DB-Zugriffe genutzt werden, hängt davon ab, ob Eager oder Lazy Loading verwendet wird. Dazu folgt später ein separater Abschnitt. Eine Änderung der *Collection* – Modifikation ihres Inhalts oder Zuweisung einer kompletten neuen *Collection* – führt analog zu oben am Transaktionsende zur passenden Änderung der Verweise in der Datenbank.

Bei der Wahl des *Collection*-Typs hat man die Qual der Wahl: Sollte man *List* verwenden oder lieber *Set*? Nun, das kommt darauf an … *Set* entspricht eher der Semantik einer DB-Tabelle, in der ebenso nicht zwei gleiche Objekte liegen können. *List*-Elemente können dagegen angeordnet werden, was sicher auch nützlich sein kann.

Statt eines der einfachen Collection-Typen kann für mengenwertige Relationsattribute auch *java.util.Map* genutzt werden. Anders als bei Element Collections finden sich dafür aber im Bereich der Relationen kaum sinnvolle Anwendungen.

3.3.3 Bidirektionale 1:n-Relationen

Nach der Betrachtung der beiden unidirektionalen Varianten stellt eine bidirektionale Beziehung nun keine große Hürde mehr dar. Hier sind nun auf beiden Seiten Relationsattribute aufzunehmen, und zwar auf der einen Seite eine *Collection*, auf der anderen Seite ein einzelnes Attribut vom Typ der Gegenseite. Diese Attribute stellen allerdings nicht zwei unabhängige Relationen dar, sondern sind nur zwei Sichten auf eine Beziehung. Um dies auszudrücken, muss der Annotation *@OneToMany* als Parameter *mappedBy* der Name des Relationsattributs der Gegenseite mitgegeben werden (Listing 3.31).

```
@Entity
@Access(AccessType.FIELD)
public class Publisher
{
  @Id @GeneratedValue
  private Integer    id;
  private String     name;

  @OneToMany(mappedBy = "publisher")
  private List<Book> books;
  …
}

@Entity
@Access(AccessType.FIELD)
public class Book
{
  @Id @GeneratedValue
  private Integer   id;
  private String    name;
  private String    isbn;

  @ManyToOne
  private Publisher publisher;
  …
}
```

Listing 3.31: Bidirektionale 1:n-Beziehung

Im Beispiel definiert also das Attribut *publisher* der Klasse *Book* die eine Sicht der Relation, auf die sich die umgekehrte Sicht – das Attribut *books* in *Publisher* – mithilfe des *mappedBy*-Parameters bezieht.

Die Platzierung des *mappedBy*-Parameters bestimmt zudem, welche Seite der Owner der Relation ist, nämlich die Seite ohne diesen Parameter. In der aktuellen JPA-Version muss dies für 1:n-Relationen stets die Many-Seite sein. Diese Beschränkung wird vermutlich in einer der nächsten Versionen des Standards aufgehoben.

Das Mapping zur Datenbank entspricht dem von unidirektionalen n:1-Relationen, wobei mithilfe der Annotationen *@JoinColumn* bzw. *@JoinTable* auf der Owner-Seite die Namen von Fremdschlüsseln und der ggf. vorhandenen Verknüpfungstabelle wie zuvor gezeigt bestimmt werden können.

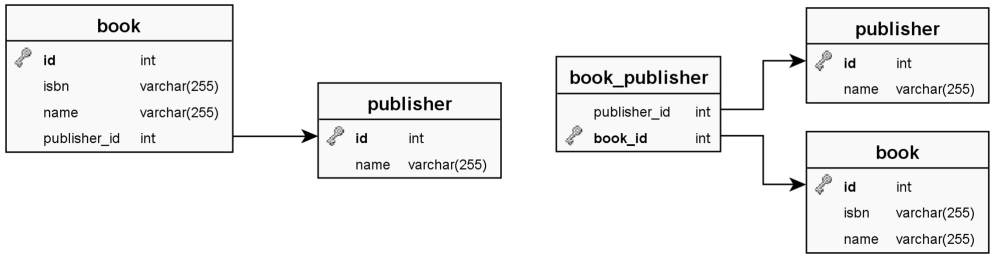

Abbildung 3.12: Tabellen zu Listing 3.31 links ohne und rechts mit „@JoinTable“

Bei der Benutzung der bidirektional verbundenen Entities zur Programmlaufzeit können nun wiederum die Relationsattribute zur Navigation durch die Beziehung genutzt werden: Ein per *find* gelesenes Objekt vom Typ *Publisher* enthält in seinem Attribut *books* alle zugeordneten *Book*s, und umgekehrt enthält das Attribut *publisher* eines eingelesenen *Book*-Objekts den zugehörigen *Publisher*, wobei das Füllen dieser Attribute durchaus bis zur ersten Verwendung verzögert sein kann.

Beim Verknüpfen von Objekten zur Laufzeit ist etwas Vorsicht geboten: Nur das Relationsattribut auf der Owner-Seite ist schlussendlich für den DB-Eintrag maßgeblich. Das Attribut der Gegenseite – die umgekehrte Sicht der bidirektionalen Relation – wird damit aber nicht automatisch mitgepflegt.

```
Publisher publisher = entityManager.find(Publisher.class, 1);
Book book = entityManager.find(Book.class, 4711);
book.setPublisher(publisher);   // (a) Publisher in Book setzen
publisher.getBooks().add(book); // (b) Book zu Publisher hinzufügen
```

Listing 3.32: Objektzuordnung in einer bidirektionalen 1:n-Relation

In Listing 3.32 wird einem *Publisher* ein neues *Book* zugeordnet. Es wird hier angenommen, dass der gezeigte Codeausschnitt innerhalb einer Transaktion ausgeführt wird. Dann

wird die neue Zuordnung beim Commit der Transaktion in die Datenbank geschrieben, d. h. der entsprechende Fremdschlüsselwert eingetragen. Dafür ist aber nur das mit (a) markierte Statement zuständig. Die darauf folgende Anweisung (b) hat dagegen keinerlei Konsequenz im Hinblick auf den DB-Inhalt. Sie führt nur zu einer konsistenten Änderung des *Publisher*-Objekts im Hauptspeicher.

Es stellt natürlich eine gewisse Gefahr dar, dass die Bedienung beider Relationsattribute vom Benutzer der Klassen konsistent vorgenommen werden muss. Eine mögliche Alternative ist die Verkapselung der Zugriffe in der entsprechenden Methode bspw. der Owner-Seite. So könnte man im Beispiel der *Publisher-Book*-Relation den schreibenden Zugriff auf *Publisher.books* unterdrücken, indem man die entsprechenden Methoden nicht *public* macht. Stattdessen würde dann die Methode *Book.setPublisher* die Relationsattribute der Gegenseite mit pflegen. Um den Quereinstieg über den Getter zu verhindern, sollte dieser eine unveränderliche *Collection* liefern (Listing 3.33).

```
@Entity
@Access(AccessType.FIELD)
public class Publisher
{
  …
  public List<Book> getBooks()
  {
    return Collections.unmodifiableList(this.books);
  }

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

  void removeBook(Book book)
  {
    this.books.remove(book);
  }
  …
}

@Entity
@Access(AccessType.FIELD)
public class Book
{
  …
  public void setPublisher(Publisher publisher)
  {
    if (this.publisher != null)
    {
      this.publisher.removeBook(this);
    }
```

```
    this.publisher = publisher;

    if (this.publisher != null)
    {
      this.publisher.addBook(this);
    }
  }
  …
}
```

Listing 3.33: Pflege des Attributs der Gegenseite im Setter einer Entity

Kritische Gemüter werden hier einwenden, dass einerseits die konsistente Pflege des Relationsattributs auf der Nicht-Owner-Seite nun immer durchgeführt wird und damit auch den entsprechenden Laufzeitaufwand auslöst, selbst wenn die Daten anschließend gar nicht mehr benötigt werden, und dass andererseits *Book.setPublisher* nun deutlich mehr tut, als man von einem Setter gemeinhin erwartet. Und ja, sie haben damit durchaus recht. Sie müssen für sich entscheiden, welche Variante für Sie das geringere Übel darstellt.

3.3.4 Uni- und bidirektionale 1:1-Relationen

Mit Blick auf die Ausführungen zu 1:n-Relationen sind die 1:1-Beziehungen leicht erklärt: Hier hat man auf der Owner-Seite ein Relationsattribut vom Typ der Gegenseite. Bei bidirektionalen Relationen ist dort ein Attribut der Owner-Seite zu finden. Beide sind mit *@OneToOne* annotiert, wobei auf der Nicht-Owner-Seite mithilfe des Parameters *mappedBy* wiederum Bezug auf die andere Seite genommen wird. Listing 3.34 zeigt das Beispiel der 1:1-Beziehung zwischen einer Firma und dem zugehörigen Handelsregistereintrag. Das Mapping auf DB-Tabellen entspricht dem der n:1-Beziehungen, kann aber genau wie dort mittels *@JoinColumn* bzw. *@JoinTable* beeinflusst werden.

```
@Entity
@Access(AccessType.FIELD)
public class Company
{
  @Id @GeneratedValue
  private Integer     id;
  private String      name;

  @OneToOne
  private CommRegEntry commRegEntry;
  …
}

@Entity
@Access(AccessType.FIELD)
public class CommRegEntry
{
  @Id
```

```
  public Integer id;

  private String legalForm;
  …
}
```

Listing 3.34: Beispiel für eine unidirektionale 1:1-Relation

Die Relation wird in der Datenbank normalerweise mithilfe eines Foreign Keys abgebildet. So enthält im Beispiel die Tabelle *company* die Spalte *commRegEntry_id* als Referenz auf die Tabelle *commRegEntry* (Abbildung 3.13 links). Diesen Fremdschlüssel kann man einsparen, wenn man die beiden verbundenen Tabellen mit gemeinsamen Primärschlüsselwerten versorgt (Abbildung 3.13 rechts). Dazu muss zusätzlich zu *@OneToOne* auf der Owner-Seite die Annotation *@PrimaryKeyJoinColumn* verwendet werden (Listing 3.35).

```
@OneToOne
@PrimaryKeyJoinColumn
private CommRegEntry commRegEntry;
```

Listing 3.35: Verknüpfung über den gemeinsamen Primärschlüssel

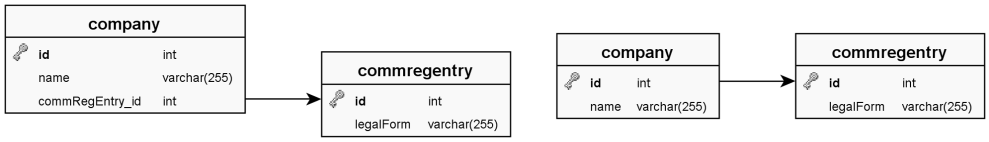

Abbildung 3.13: Tabellen zu Listing 3.34 (links) und Listing 3.35 (rechts)

Die Nutzung eines gemeinsamen Primärschlüssels scheint elegant zu sein, erzeugt aber ein Mismatch mit der Deklaration der Java-Klassen. Aus deren Deklaration könnte man entnehmen, dass die Zuordnung zweier Objekte innerhalb der 1:1-Relation durch Zuweisung des einen Objekts an das Relationsattribut des Owner-Objekts geschieht. So ist das ja im Allgemeinen bei JPA-Relationen – nur nicht bei 1:1-Relationen mit gemeinsamem Primärschlüssel. Hier müssen die IDs der zu verbindenden Objekte explizit auf den gleichen Wert gesetzt werden. Das Relationsattribut wird zwar beim Lesen korrekt besetzt, sein Wert spielt aber beim Schreiben in die DB keine Rolle. Zudem müssen die Einfügeoperationen in der richtigen Reihenfolge geschehen, um die Foreign Key Constraints der DB nicht zu verletzen. Um also z. B. eine neue *Company* mit dem zugeordneten *CommRegEntry* in die DB einzutragen, muss man wie in Listing 3.36 gezeigt vorgehen.

```
Company company = new Company();
company.setId(5812);
company.setName("Gierschlund & Raffke OHG");

CommRegEntry commRegEntry = new CommRegEntry();
commRegEntry.setId(company.getId());   // Gleiche ID erzeugt Zuordnung
commRegEntry.setLegalForm("OHG");
```

```
company.setCommRegEntry(commRegEntry); // Unwichtig für DB-Eintrag

entityManager.persist(commRegEntry);   // Reihenfolge!
entityManager.persist(company);
```

Listing 3.36: Objektzuordnung bei gemeinsamem Primärschlüssel

Damit finden wir bei 1:1-Relationen die Situation, dass eine Änderung des Mappings durch Hinzufügen von *@PrimaryKeyJoinColumn* eine andere Behandlung der Objekte zur Laufzeit nach sich zieht. Die Kopplung der Java-Klassen an das Tabellenlayout ist hier also offensichtlich stärker als bei anderen Mapping-Optionen. Viele machen aus diesem Grund einen Bogen um *@PrimaryKeyJoinColumn*, wenn es möglich ist.

3.3.5 Uni- und bidirektionale n:m-Relationen

N:m-Relationen werden analog zu denen der bislang dargestellten Kardinalitäten definiert. Die Relationsfelder sind hier allerdings auf beiden Seiten – so vorhanden – *Collection*s und als Relationsannotation wird *@ManyToMany* verwendet. In Listing 3.37 ist eine bidirektionale n:m-Relation zwischen Anwendungen und Benutzern – z. B. als Basis eines Rechtesystems – dargestellt.

```
@Entity
@Access(AccessType.FIELD)
public class Application
{
  @Id @GeneratedValue
  private Integer    id;
  private String     name;

  @ManyToMany(mappedBy = "usableApplications")
  private List<User> authorizedUsers;

  // ...
}

@Entity
@Access(AccessType.FIELD)
@Table(name = "USERS") // USER ist für viele DB ein reserviertes Wort
public class User
{
  @Id
  private String           id;
  private String           name;

  @ManyToMany
  private List<Application> usableApplications;

  // ...
}
```

Listing 3.37: Beispiel für eine bidirektionale n:m-Relation

In der Datenbank lässt sich eine n:m-Relation naturgemäß nur mithilfe einer Verknüpfungstabelle realisieren (Abbildung 3.14). Auf deren Gestaltung kann man wiederum mithilfe von *@JoinTable* auf der Owner-Seite Einfluss nehmen.

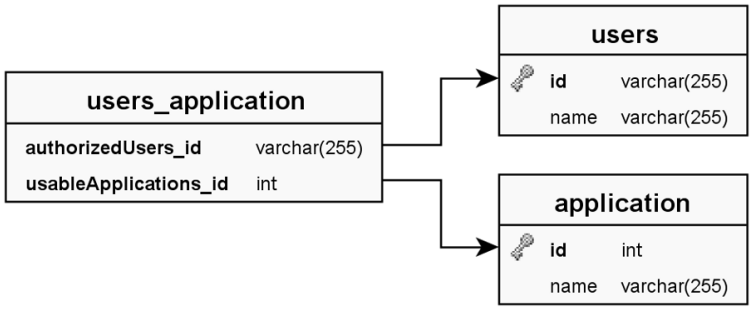

Abbildung 3.14: Tabellen zu Listing 3.37

3.3.6 Eager und Lazy Loading

Nach dem Lesen einen Objekts aus der Datenbank ist gewährleistet, dass wir die Attribute des Objekts inklusive der Relationsattribute direkt benutzen können. Das umfasst auch die per Relation zugeordneten Objekte. Das heißt aber nicht, dass der gesamte Datenumfang bereits beim ersten Zugriff zur Datenbank geladen wird. Die referenzierten Objekte können vielmehr verzögert nachgeladen werden, wenn sie benutzt werden.

Zur Steuerung dieses Ladeverhaltens akzeptieren die Relationsannotationen *@OneToOne*, *@OneToMany*, *@ManyToOne* und *@ManyToMany* den Parameter *fetch*, dem ein Wert der Aufzählung *FetchType* übergeben wird:

- *FetchType.EAGER*

Die referenzierten Objekte werden direkt gelesen, wenn das führende Objekt eingelesen wird (Eager Loading)

- *FetchType.LAZY*

Die referenzierten Objekte werden zunächst nicht gelesen, sondern nachgeladen, wenn sie zum ersten Mal verwendet werden (Lazy Loading)

Wird kein *FetchType* angegeben, gilt für die *Collection*-basierten Attribute *LAZY* als Voreinstellung, während alle anderen Attribute *EAGER* geladen werden.

Im weiter oben besprochenen Beispiel der 1:n-Relation zwischen *Publisher* und *Book* gilt als Vorgabe Lazy Loading für das Attribut *Publisher.books*. Diese *Collection* wird also beim Lesen eines *Publisher*-Objekts noch nicht mit Werten gefüllt. Dies geschieht mit einem erneuten Zugriff auf die Datenbank, wenn die Daten benutzt werden. Die Spezifikation

beschreibt nicht exakt, wann der Nachladevorgang ausgelöst wird. In aller Regel reicht der Aufruf einer der *Collection*-Methoden – und sei es nur *size()*.

Wäre das Relationsattribut dagegen mit *EAGER* parametrisiert worden (Listing 3.38), würden mit jedem gelesenen *Publisher* direkt die zugeordneten Books eingelesen.

```
@OneToMany(mappedBy = "publisher", fetch=FetchType.EAGER)
private List<Book> books;
```

Listing 3.38: Konfiguration von Eager Loading bei einer 1:n-Relation

Achtung: Mit Eager Loading ist die Gefahr verbunden, dass man unbewusst (viel) mehr Daten einliest, als man für den aktuellen Geschäftsprozess benötigt. Denken Sie daran, dass die Menge der referenzierten Objekte groß sein kann und dass Eager Loading transitiv funktioniert. Sind also in den direkt geladenen Objekten wiederum *EAGER*-Attribute, werden auch diese geladen usw. Bei ungeschickter Konfiguration der Relationen einer Anwendung hat man so vielleicht mit einem *find* direkt die ganze Datenbank im Speicher …

Gehen Sie also vorsichtig mit Eager Loading um. Die Voreinstellung, dass nur Einzelobjekte *EAGER* geladen werden, und zunächst nur für *Collection*-basierte Werte Lazy Loading gilt, ist in vielen Fällen sehr gut. Nur wenn Sie wissen, dass Ihre Anwendung in allen (!) Fällen die referenzierten Objekte benötigt, und dass weiterhin die Menge dieser Objekte handhabbar ist – nur dann sollten Sie für *Collection*s Eager Loading einsetzen.

Ein Problem kann Lazy Loading im Zusammenhang mit Detached Objects erzeugen: Das Nachladen von LAZY-Attributen ist nur für persistente Objekte möglich. Verlassen diese den Bereich des Entity Managers, können bis dahin nicht geladene Werte nicht mehr gefüllt werden. Greift die Anwendung dann darauf zu, wird eine (providerspezifische) Lazy Load Exception ausgeworfen.

Wollen Sie also mit Detached Objects arbeiten, müssen Sie dafür sorgen, dass alle von der Anwendung genutzten Attribute geladen wurden, bevor die Objekte vom Entity Manager abgekoppelt werden – entweder durch Eager Loading oder durch einen expliziten Zugriff. Bei der Ausführung einer Query besteht die Möglichkeit, durch einen sog. Fetch Join Relationsattribute direkt zu laden, auch wenn sie mit *LAZY* parametrisiert wurden. Queries und Fetch Joins werden später im Detail erläutert.

Lazy Loading kann auch für andere als Relationsattribute angegeben werden: *@ElementCollection* akzeptiert den Parameter *fetch* ebenso wie *@OneToMany*. Auch hier ist der voreingestellte Wert *LAZY*.

Für einfache Attribute steht die Annotation *@Basic* zur Verfügung. Auch hier kann *fetch* verwendet werden, allerdings mit dem Vorgabewert *EAGER*. Sinn macht die Verwendung von Lazy Loading für solche Attribute aber allenfalls für sehr große Objekte wie bspw. LOBs (Listing 3.39).

```
@Lob
@Basic(fetch = FetchType.LAZY)
private byte[] picture;
```

Listing 3.39: Lazy Loading für ein großes Einzelattribut

Die Angabe von *LAZY* ist nur ein Hinweis an den Persistenzprovider, dass Lazy Loading genutzt werden darf. Die Spezifikation verlangt nicht, dass dieses Verfahren für alle Attributtypen verfügbar ist, allerdings unterstützen alle mir bekannten Provider (EclipseLink, Hibernate, OpenJPA) Lazy Loading durchgängig. Ggf. muss allerdings ein Bytecode Enhancement durchgeführt werden, d. h. eine Manipulation des Bytecodes der betroffenen Klassen. Dies kann zur Build-Zeit mit speziellen Werkzeugen des jeweiligen Providers – meist Ant- oder Maven-basiert – oder beim Laden der Klassen durch den Container geschehen. In der SE-Umgebung kann ggf. ein sog. Java-Agent genutzt werden. Details dazu finden Sie in der Dokumentation Ihres Providers.

3.3.7 Entity Graphs

Seit JPA 2.1 ist es möglich, mit Entity Graphs das Ladeverhalten flexibler zu beeinflussen: Zum einen können mehrere Entity Graphs für jede persistente Klasse konfiguriert werden, um so den Bedürfnissen unterschiedlicher Geschäftsprozesse entgegenzukommen. Zum anderen können in einem Entity Graph Attribute von referenzierten Entities aufgenommen und somit die Ladeoperation eines beliebig tiefen Objektbaums bestimmt werden.

Ein Entity Graph kann einer Entity-Klasse mithilfe der Annotation *@NamedEntityGraph* hinzugefügt werden. Sollen mehrere Entity Graphs definiert werden, muss wie schon an anderen Stellen eine „Plural-Annotation“ verwendet werden, hier *@NamedEntityGraphs*.

Der Entity Graph Name muss für die Persistence Unit eindeutig sein; der Vorgabewert ist der Entity-Name selbst. Mit dem Parameter *attributeNodes* werden die Attribute angegeben, die bei Benutzung des Entity Graph geladen werden sollen. Das Beispiel in Listing 3.40 zeigt einen einfachen Entity Graph für die Klasse *Publisher* namens *Publisher_books*, der das Relationsattribut *books* umfasst. Mit dem Parameter *subGraphs* können Entity Graphs für referenzierte Objekte angegeben werden. Dies zeigt das zweite Beispiel in Listing 3.40: Der Entity Graph *Publisher_booksAndAuthors* enthält über das Attribut *books* hinaus das Relationsattribut *authors* der Klasse *Book*.

```
@Entity
@NamedEntityGraphs({
  @NamedEntityGraph(
    name = "Publisher_books",
    attributeNodes = @NamedAttributeNode("books")),
  @NamedEntityGraph(
```

```
    name = "Publisher_booksAndAuthors",
    attributeNodes = @NamedAttributeNode(value = "books",
                                         subgraph = "Book_authors"),
    subgraphs = @NamedSubgraph(
                 name = "Book_authors",
                 attributeNodes = @NamedAttributeNode("authors"))) })
public class Publisher
{
  …
  @OneToMany(fetch = FetchType.LAZY, …)
  private List<Book> books;
```

Listing 3.40: Entity Graph

Zur Nutzung von Entity Graphs bedient man sich sog. Hints, die u. a. der Methode *EntityManager.find* als dritter Parameter übergeben werden können. Diese Hints sind Key-Value-Paare in einem *Map*-Objekt. Im Zusammenhang mit Entity Graphs definiert JPA 2.1 zwei Hints:

- *javax.persistence.fetchgraph*: Nur die im genannten Graphen enthaltenen Attribute werden geladen
- *javax.persistence.loadgraph*: Die im genannten Graphen enthaltenen Attribute werden zusätzlich zu den ohnehin mit *EAGER* konfigurierten Attributen geladen

Aus technischen Gründen werden ID- und Versionsattribute in jedem Fall geladen.

Der Beispielcode in Listing 3.41 würde somit in dem gefundenen *Publisher* neben dem ID-Attribut auch die dem *Publisher* zugeordneten *books* laden. Die *Book*-Attribute selbst würden mit dem Standardverfahren geladen.

```
Map<String, Object> hints = new HashMap<>();
hints.put("javax.persistence.fetchgraph", "Publisher_books");
Publisher publisher = em.find(Publisher.class, someId, hints);
```

Listing 3.41: Entity Graph als Fetch Graph

Im Unterschied dazu verwendet Listing 3.42 den Hint *javax.persistence.loadgraph* und einen Entity Graph, der auch die Entity *Book* beeinflusst: Neben den standardmäßig geladenen Attributen von *Publisher* und *Book* würden hier auf jeden Fall *Publisher.books* und *Book.authors* geladen.

```
Map<String, Object> hints = new HashMap<>();
hints.put("javax.persistence.loadgraph", "Publisher_booksAndAuthors");
Publisher publisher = em.find(Publisher.class, someId, hints);
```

Listing 3.42: Entity Graph als Load Graph

Entity Graphs können auch dynamisch erzeugt und ggf. verändert werden. Dazu dienen die Methode *EntityManager.createEntityGraph* und die Methoden des *Interfaces EntityGraph<T>* zum Erstellen eines Entity Graph sowie die Methode *EntityManagerFactory.addNamedEntityGraph* zum Registrieren des Entity Graph. Für Details sei auf die Dokumentation der Methoden verwiesen.[9]

3.3.8 Kaskadieren

Bislang war eine Voraussetzung für die Arbeit mit den durch eine Relation referenzierten Objekten, dass diese jeweils unabhängig vom referenzierenden Objekt mit dem Entity Manager bearbeitet werden müssen. Um also z. B. einen neuen *Publisher* mit seinen *Book*s in die Datenbank einzufügen, müssen zunächst die *Book*s an *EntityManager.persist* übergeben werden, bevor auch das neue *Publisher*-Objekt persistiert werden kann.

Mithilfe des Parameters *cascade*, den alle Relationsannotationen annehmen, kann dies bequemer gestaltet werden. Enthält der Parameter z. B. den Wert *CascadeType.PERSIST*, so wird ein auf das Entity-Objekt angewendetes *persist* automatisch auch auf die davon abhängigen Objekte angewendet. Sind darin wiederum Relationen mit *CascadeType.PERSIST* vorhanden, setzt sich das Verfahren entsprechend fort. *CascadeType* ist ein Aufzählungstyp, der die Konstanten *PERSIST*, *MERGE*, *REMOVE*, *REFRESH* und *DETACH* enthält. Sie beziehen sich in der beschriebenen Weise auf die Methoden *persist*, *merge* etc. des Entity Managers. Der Parameter *cascade* der Annotationen akzeptiert ein Array von *CascadeType*-Werten, sodass hier eine beliebige Kombination der Entity-Manager-Grundoperationen kaskadierend gestaltet werden kann. Sollen alle genannten Methoden einbezogen werden, kann dafür der Wert *CascadeType.ALL* verwendet werden, der alle restlichen umfasst. Listing 3.43 zeigt als Beispiel eine klassische Master-Detail-Kombination: Einem *Order*-Objekt sind mehrere *OrderLine*s zugeordnet. Abbildung 3.15 zeigt die zugehörige Tabellenstruktur. Logisch betrachten wir die Kombination als einen Auftrag, der zusammen mit seinen Auftragspositionen gespeichert werden soll. Daher ist in *@OneToMany* für *cascade* u. a. *PERSIST* eingetragen. Umgekehrt sollen beim Löschen eines Auftrags auch die zugeordneten *OrderLine*-Objekte aus der DB entfernt werden, daher ist auch *REMOVE* eingesetzt.

```
@Entity
// Achtung: ORDER ist ein reserviertes Wort in SQL
@Table(name = "ORDERS")
@Access(AccessType.FIELD)
public class Order
{
  @Id @GeneratedValue
```

9 Die Implementierung von Entity Graphs war zum Zeitpunkt der Drucklegung leider weder in EclipseLink 2.5.0 noch in Hibernate 4.3.0 vollständig und fehlerfrei. Weitere Hinweise finden Sie im Begleitprojekt zum Kapitel und in unserem Java-EE-Blog: *http://javaeeblog.wordpress.com/*

```
  private Integer        id;

  @OneToMany(cascade = { CascadeType.PERSIST, CascadeType.REMOVE })
  @JoinColumn(name = "ORDER_ID")
  private List<OrderLine> orderLines;
  …
}

@Entity
@Access(AccessType.FIELD)
public class OrderLine
{
  @Id @GeneratedValue
  private Integer id;
  …
}
```

Listing 3.43: Kaskadieren von „persist“ und „remove“

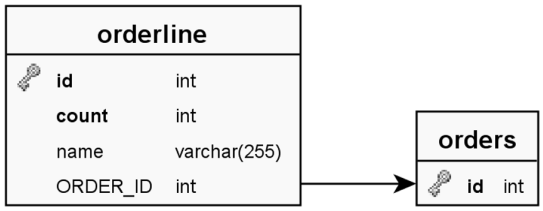

Abbildung 3.15: Tabellen zu Listing 3.43

Im Beispiel sind die restlichen *CascadeType*s weggelassen worden. Die wären hier aber auch sinnvoll, denn die *OrderLine*-Objekte sind komplett abhängige Objekte. Sie sollten alle Operationen des jeweiligen *Order*-Objekts mitmachen, also auch *merge*, *refresh* und *detach*. In einem solchen Fall kann natürlich statt einer Aufzählung aller Werte *CascadeType.ALL* verwendet werden.

Kaskadierende Operationen sind recht bequem und erlauben eine elegante Programmierung von Zugriffen auf komplexere Objekte. Trotzdem sollten Sie beim Einsatz von *cascade* Vorsicht walten lassen. *REMOVE* entspricht einem Cascading Delete in der DB, was i. A. nur für abhängige Objekte sinnvoll ist, d. h. für Objekte, die ohne das sie referenzierende Objekt irrelevant sind. Die anderen Operationen sind zwar etwas harmloser, sind aber bei ganz genauer Betrachtung auch nur bei abhängigen Objekten sinnvoll.

3.3.9 Orphan Removal

Von Orphans, verwaisten Einträgen, sprechen wir, wenn sie zwar noch vorhanden sind, aber nicht mehr genutzt werden können. Diese Situation entsteht, wenn die Referenz auf einen abhängigen Eintrag entfernt wird, der Eintrag selbst aber bestehen bleibt. Genau

das würde passieren, wenn im Beispiel oben ein Auftrag, bestehend aus einem *Order*-Objekt und einigen *OrderLine*-Objekten, so verändert wird, dass die Anzahl *OrderLine*s verkleinert wird. Beim Abspeichern des Auftrags würden die betroffenen Referenzen in der DB entfernt, d. h. die Fremdschlüssel auf *null* gesetzt, die Einträge in der Tabelle aber erhalten bleiben.

Angenommen, wir hätten zunächst einen Auftrag mit vier Positionen. Nach Entfernen der letzten Position ergibt sich damit die in Abbildung 3.16 dargestellte Situation. Der betroffene Eintrag existiert weiterhin in der Datenbank, allerdings nun ohne Verbindung zu einem *Order*-Eintrag. Fachlich ist dieser Eintrag nun unsichtbar.

id	name	count	ORDER_ID
14	Apfel	10	13
15	Pflaume	50	13
16	Limette	5	13
17	Birne	30	13

⇨

id	name	count	ORDER_ID
14	Apfel	10	13
15	Pflaume	50	13
16	Limette	5	13
17	Birne	30	(null)

Abbildung 3.16: „OrderLine"-Tabelle vor und nach dem Entfernen einer Position aus der „Order 13" ohne Orphan Removal …

Man könnte zur Bereinigung dieses Zustands bspw. regelmäßig alle Einträge aus der Tabelle löschen, deren *ORDER_ID null* ist. Das geht aber im Fall von 1:1- oder 1:n-Relationen eleganter mithilfe des Parameters *orphanRemoval*, den die entsprechenden Annotationen *@OneToOne* bzw. *@OneToMany* akzeptieren. Hat er den Wert *true*, werden verwaiste Einträge automatisch gelöscht (Abbildung 3.17). Die Voreinstellung für den Parameter ist *false*.

id	name	count	ORDER_ID
14	Apfel	10	13
15	Pflaume	50	13
16	Limette	5	13
17	Birne	30	13

⇨

id	name	count	ORDER_ID
14	Apfel	10	13
15	Pflaume	50	13
16	Limette	5	13

Abbildung 3.17: … und mit Orphan Removal

Orphan Removal ist also auch eine Art kaskadierende Löschoperation, diesmal aber ausgelöst durch die Änderung eines Relationsattributs. Man findet *orphanRemoval=true* daher häufig, wenn *cascade* u. a. *REMOVE* enthält.

3.3.10 Anordnung von Relationselementen

Die Elemente von *Collection*-basierten Relationsattributen werden beim Einlesen in einer nicht vorbestimmten Reihenfolge in die *Collection* eingetragen – üblicherweise durch Eigenschaften der Datenbank bestimmt. Man kann die Elemente allerdings in eine fachliche Reihenfolge bringen lassen, indem man das Relationsattribut mit der Annotation *@OrderBy* versieht. Als Parameter gibt man dabei den Namen des Attributs der *Collection*-Elemente an, nach dem sortiert werden soll. Mit dem Zusatz *ASC* oder *DESC* kann

eine aufsteigende oder absteigende Sortierung erreicht werden; *ASC* ist voreingestellt. Mehrere Sortierangaben können durch Komma voneinander getrennt gemacht werden. Das Beispiel in Listing 3.44 zeigt, wie die *OrderLine*s eines *Order*-Objekts anhand ihres Attributs *name* absteigend sortiert werden können (was hier vermutlich nicht sonderlich sinnvoll ist …).

```
@OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
@JoinColumn(name = "ORDER_ID")
@OrderBy("name DESC")
private List<OrderLine> orderLines;
```

Listing 3.44: Anordnung von Relationselementen mit „@OrderBy“

Die geforderte Anordnung der Elemente wird durch einen entsprechenden Zusatz im zum Lesen verwendeten Datenbankbefehl erzeugt. Demzufolge ist danach natürlich nur die Anwendung weiter für die Einhaltung der Reihenfolge verantwortlich. Soll also im Beispiel ein neues *OrderLine*-Objekt in die *Collection* eingefügt werden, ist es die Aufgabe der Anwendung, dies an der in Bezug auf die Sortierung richtigen Stelle zu tun.

Wird *@OrderBy* ohne Parameter verwendet, ergibt sich die Anordnung durch die Sortierung der ID-Werte.

@OrderBy kann auch in Kombination zu *@ElementCollection* verwendet werden. Wird hier keine Sortierangabe gemacht, werden die Elemente entsprechend ihrer natürlichen Ordnung angeordnet. Das ist aber nur bei *Collection*s über einfachen Typen erlaubt. Für *Embeddable*s ist die Sortierangabe obligatorisch.

Eine Alternative zu @OrderBy ist die sog. persistente Ordnung, bei der die Anordnung der *Collection*-Elemente selbst persistent gemacht wird, d. h. die Reihenfolge wird in der Datenbank mit abgespeichert und beim späteren Lesen wieder genauso hergestellt. Dazu muss das Relationsattribut mit der Annotation *@OrderColumn* versehen werden. Das führt zur Verwendung einer zusätzlichen Spalte in der Tabelle der referenzierten Entity bzw. in der Verknüpfungstabelle. Beim Speichern wird diese Spalte zum Durchnummerieren der Einträge verwendet (aufsteigend, ohne Lücken, beginnend mit *0*). Beim späteren Lesen wird anhand dieser Zahlen die alte Ordnung wieder hergestellt. Der Name der Nummerierungsspalte ergibt sich aus dem Namen des Relationsattributs, ergänzt um *'_ORDER'*. Er kann aber auch explizit als Parameter *name* der Annotation *@OrderColumn* mitgegeben werden (Listing 3.45, Abbildung 3.18, Abbildung 3.19).

```
@OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
@JoinColumn(name = "ORDER_ID")
@OrderColumn(name = "ORDERLINES_ORDER")
private List<OrderLine> orderLines;
```

Listing 3.45: Persistente Anordnung der Relationselemente

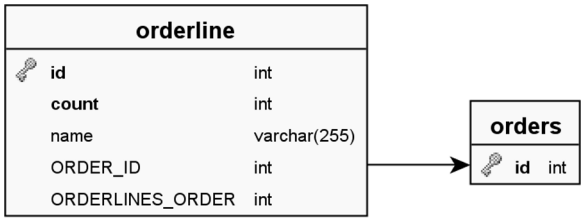

Abbildung 3.18: Tabellenlayout zu Listing 3.45

id	count	name	ORDER_ID	ORDERLINES_ORDER
14	10	Apfel	13	0
15	50	Pflaume	13	1
16	5	Limette	13	2
17	30	Birne	13	3

Abbildung 3.19: Nummerierungsspalte in der „OrderLine“-Tabelle

Auch dieses Anordnungsverfahren kann für Element-Collections verwendet werden.

3.4 Queries

Neben den bisher betrachteten Einzeleintragsoperationen spielen bei der Arbeit mit Datenbankinhalten Suchanfragen eine große Rolle. Java Persistence bietet dafür gleich drei Möglichkeiten an: Eine objektorientierte Abfragesprache, direktes SQL und eine Programmschnittstelle zum Aufbau von Queries.

3.4.1 JPQL

Die Java Persistence Query Language ist eine SQL-ähnliche Abfragesprache, die allerdings nicht mit Tabellen und Spalten arbeitet, sondern mit Klassen und Objekten.

Queries werden zur Laufzeit durch Objekte des Typs *TypedQuery<T>* repräsentiert, wobei der Typparameter angibt, welchen Ergebnistyp man bei Ausführung der Query erwarten kann. Zur Erzeugung eines Query-Objekts stellt *EntityManager* die Methode *createQuery* zur Verfügung. Die Query-Methode *getResultList* führt die Abfrage durch und liefert das (evtl. leere) Ergebnis in einer passend getypten Liste (Listing 3.46).

```
EntityManager em = …
TypedQuery<Country> query
  = em.createQuery("select c from Country c where c.name like 'D%'",
                   Country.class);
List<Country> countries = query.getResultList();
```

Listing 3.46: Ausführen einer Query mit Ergebnisliste

Die grundsätzliche Form des Abfragetextes wird Ihnen vermutlich von SQL her bekannt sein: *select*, *from* etc. sind übernommen worden und haben die gleiche Bedeutung wie in SQL. Auch die Operatoren wie *like* sind über weite Strecken deckungsgleich. Unterschiede bestehen u. a. darin, dass im Beispiel komplette Objekte selektiert werden anstelle der in SQL üblichen Ergebnis-Tupel. Zudem sind *Country* und *name* Java-Bezeichner und nicht SQL-Namen.

Wird von einer Query nur ein einzelnes Objekt als Ergebnis erwartet, kann man statt *getResultList* die Methode *getSingleResult* verwenden(Listing 3.47). Sie liefert ein passend getyptes Objekt als Rückgabewert und wirft eine Exception aus, wenn nicht genau ein Ergebnis gefunden wird:

- *NoResultException*, falls kein Eintrag gefunden wurde
- *NonUniqueResultException*, falls mehr als ein Eintrag gefunden wurde

```
EntityManager em = …
TypedQuery<Country> query
  = em.createQuery("select c from Country c where c.carCode='D'",
                   Country.class);
Country country = query.getSingleResult();
```

Listing 3.47: Ausführen einer Query mit Einzelergebnis

Die Methode *EntityManager.createQuery* steht auch in einer älteren Variante ohne den Typparameter zur Verfügung. Sie liefert dann statt eines *TypedQuery<T>*-Objekts eines vom Typ *Query*. Es ist grundsätzlich gleichwertig, liefert aber bspw. bei der Query-Ausführung nur eine ungetypte *List* bzw. ein *Object* als Ergebnis. Wo immer es möglich ist, sollten Sie daher die neue, getypte Methodenvariante bevorzugen.

Der grundsätzliche Aufbau eines JPQL-Ausdrucks ist der folgende, wobei die in Klammern gesetzten Teile optional sind:

Select-Klausel
From-Klausel
[Where-Klausel]
[Groupby-Klausel]
[Having-Klausel]
[Orderby-Klausel]

Select- und From-Klausel

Mit diesem obligatorischen Anteil eines JPQL-Ausdrucks wird das Query-Ergebnis strukturell beschrieben. In der einfachsten Form werden im Select-Teil ein oder mehrere Werte ausgewählt, deren Ursprung im From-Teil erklärt wird. Die Verbindung beider Teile geschieht über Identifikationsvariablen. Die selektierten Werte sind komplette Objekte oder

Teile davon, die wie in Java üblich mithilfe des Punktoperators angegeben werden (Listing 3.48).

```
select c from Country c
select c.population from Country c
select c.name, c.population from Country c
```

Listing 3.48: Selektion kompletter Objekte oder Einzelwerte

Die hinter dem Punkt stehenden Attributnamen sind bei Field Access die Namen der Instanzvariablen der Entity, bei Property Access die Namen der Properties, d. h. die Namen der Getter ohne das führende *get* mit kleinem Anfangsbuchstaben. Steckt hinter dem Attribut ein eingebettetes Objekt oder ein 1:1- oder n:1-Relationsattribut, so kann mit einem weiteren Punkt auf dessen Teilattribute zugegriffen werden.

Das Ergebnis einer Abfrage mit nur einem Selektionswert ist passend zu diesem Wert getypt. Die erste Abfrage im Beispiel würde also eine Liste von *Country*-Objekten liefern, die zweite eine Liste von *Long*-Zahlen.

Werden mehrere Werte selektiert, ist das Ergebnis der Query ein *Object*-Array bzw. eine Liste darüber. Die letzte Abfrage im Beispiel würde also eine Liste von *Object[2]*-Objekten liefern, die jeweils mit einem *String* und einem *Long* gefüllt sind. Eine Alternative zum *Object[]*-Ergebnis stellen die weiter unten beschriebenen Constructor Expressions dar.

Die Werte der Select-Klausel beziehen sich immer auf Identifikationsvariablen, die in der From-Klausel definiert werden, und zwar wiederum wie in Java üblich durch Angabe des Variablentyps und eines frei gewählten, für die Abfrage eindeutigen Namens. Der Variablentyp ist dabei der Entity-Name, der mit der *@Entity*-Annotation bestimmt wird und als Voreinstellung dem einfachen Klassennamen der Entity entspricht. Mehrere Identifikationsvariablen können durch Kommas getrennt deklariert werden, was aber erst später bei der Formulierung von Joins sinnvoll ist.

Ein bisschen „Syntactic Sugar" ist auch erlaubt: Statt *select c* darf auch *select object(c)* geschrieben werden und *from Country c* darf auch als *from Country as c* formuliert werden.

Wie oben bereits angeführt, kann mithilfe des Punktoperators durch 1:1- und n:1-Relationen navigiert werden. Die Join-Bedingung ist bereits durch die Konfiguration der Relation definiert, muss daher in der Query nicht mehr angegeben werden (Listing 3.49).

```
// Bücher zu Verlagen, deren Name mit O beginnt (n:1-Relation)
select b from Book b where b.publisher.name like 'O%'
```

Listing 3.49: Punktoperator zum Zugriff auf Elemente einer 1:1- oder n:1-Relation

Für 1:n- und n:m-Relationen kann der Punktoperator leider nicht in der gleichen Weise angewendet werden. Vielmehr muss zum Zugriff auf mengenwertige Attribute mithilfe

des Operators *join* eine zusätzliche Query-Variable deklariert werden. Die Join-Bedingung wird analog zu oben nicht explizit angegeben, sondern aus der Konfiguration der Entities entnommen (Listing 3.50).

```
// Verlage mit Büchern zum Thema JPA (1:n-Relation)
select distinct p from Publisher p join p.books b
  where b.name like '%JPA%'
```

Listing 3.50: Deklaration einer Join-Variablen zum Zugriff auf 1:n- und n:m-Relationen

Im Beispiel wird neben der Haupt-Query-Variablen *p* die zusätzliche Variable *b* als Repräsentant der Elemente aus *p.books* deklariert. *b* kann dann wie schon gesehen in den restlichen Klauseln des JPQL-Ausdrucks verwendet werden. Bei der gezeigten Verwendung von *join* handelt es sich um einen sog. Inner Join, bei dem das Kreuzprodukt der beteiligten Objektmengen zur weiteren Verarbeitung benutzt wird. Bei der Selektion können daher Doppelergebnisse auftreten, was man mit *distinct* unterdrücken kann.

Ein Inner Join berücksichtigt nur Daten, die bzgl. der Join-Bedingung eine Zuordnung haben. Mithilfe eines Outer Joins kann die Ergebnismenge auch auf solche Elemente ausgedehnt werden, die keine zugeordneten Elemente in der betroffenen Relation haben. Java Persistence kennt nur Left Outer Joins, wo das Gesagte für die linke Seite des Joins gilt. Der Operator dazu heißt *left join* (Listing 3.51).

```
// Verlags- und Buchnamen, nur von Verlagen, die Bücher haben
select p.name, b.name from Publisher p join p.books b

// ebenso, nun allerdings für alle Verlage (b.name ist dann ggf. null)
select p.name, b.name from Publisher p left join p.books b
```

Listing 3.51: Left Outer Join

Auch hier gibt es wieder ein bisschen Syntactic Sugar: Statt *join* dürfen Sie *inner join* schreiben, statt *left join* auch *left outer join*, und die Join-Variablen können analog zu oben mit *as* abgetrennt werden.

Beachten Sie, dass Sie neben dem Join-Ausdruck keine explizite Bedingung für den Join im Query-Text angeben – diese ist ja bereits durch die Definition der Relation implizit vorgegeben. Das zweite Beispiel aus Listing 3.51 erzeugt somit ein SQL-Statement, in dem sich die *ON*-Klausel zur Verknüpfung der beiden beteiligten Tabellen aus der Definition der Relation zwischen *Publisher* und *Book* ergibt (Listing 3.52).

```
SELECT p.NAME, b.NAME FROM PUBLISHER p LEFT OUTER JOIN BOOK b
  ON b.PUBLISHER_ID = p.ID
```

Listing 3.52: Generiertes SQL-Statement für einen Left Outer Join

Seit JPA 2.1 kann ein Join-Ausdruck um eine weitere Join-Bedingung ergänzt werden, und zwar durch einen *ON*-Operator ähnlich dem in SQL. Das Beispiel (Listing 3.53) selektiert alle *Publisher*-Namen mit den Namen der Bücher, die mehr als 300 Seiten haben. Da es ein Outer Join ist, sind auch *Publisher* ohne solche Bücher im Ergebnis vertreten, wobei dort der Büchername *null* ist. Würde statt *on where* genutzt, würden diese *Publisher* nicht selektiert.

```
select p.name, b.name from Publisher p left join p.books b on b.pages>300
```

Listing 3.53: Zusätzliche Join-Bedingung

Für den Inner Join gibt es eine weitere Notation, die man aus der EJB-QL, der EJB Query Language, einem Vorläufer von JPQL, übernommen hat: *in(a.n) b* entspricht *join a.n b* (Listing 3.54).

```
// Verlags- und Buchnamen, nur von Verlagen, die Bücher haben
select p.name, b.name from Publisher p, in(p.books) b
```

Listing 3.54: Alternative Angabe eines Inner Join (vgl. Listing 3.51 oben)

Where-Klausel

Mit *where* kann die Menge der selektierten Daten eingeschränkt werden. Die darin angegebene Bedingung kann aus Attributen, Konstanten, Operatoren und Parametern zusammengesetzt werden.

Attribute werden wie schon in der Select-Klausel mithilfe des Punktoperators auf eine der Identifikationsvariablen der Query bezogen.

Konstanten sind wie in SQL üblich Texte in Hochkommas, Zahlen in üblicher Notation, die Boole'schen Konstanten TRUE und FALSE sowie Aufzählungswerte, die allerdings mit voll qualifiziertem Klassennamen angegeben werden müssen (Listing 3.55).

```
// Länder mit mehr als 50 Mio Einwohnern
select c from Country c where c.population>50000000

// Länder in Europa
select c from Country c
  where c.continent=de.gedoplan.buch.eedemos.entity.Continent.EUROPE

// Mitarbeiter aus Bielefeld (address ist ein @Embeddable)
select e from Employee e where e.address.town='Bielefeld'
```

Listing 3.55: Einschränkung der Ergebnismenge mit „where"

Sollten in einem Text Hochkommas vorkommen, müssen sie verdoppelt werden.

Operatoren

Als Operatoren stehen größtenteils die von SQL bekannten zur Verfügung:

- +, -, *, /: Grundrechenarten für numerische Werte
- =, >, >=, <, <=, <>: Vergleiche (Achtung: <> für ‚ungleich', nicht *!=*)
- [*not*] *between … and …*: Bereichsabfrage (Grenzen sind inklusive)
- [*not*] *like '…'*: Pattern-Test
- [*not*] *in (…)*: Elementtest (Menge kann Liste, Collection oder Subquery sein)
- *is* [*not*] *null*: Test auf *null* bzw. *not null*
- *is* [*not*] *empty*: Test ob ein Collection-Attribut (nicht) leer ist
- [*not*] *member of …*: Test, ob ein Element Teil einer Collection ist
- [*not*] *exists (…)*: Test, ob eine Subquery min. ein bzw. kein Ergebnis liefert
- *not*, *and*, *or*: Logische Negation bzw. Verknüpfung von Bedingungen

```
// Länder mit 50 bis 100 Mio Einwohnern
select c from Country c where c.population between 50000000 and 100000000

// Länder, deren Name mit G beginnt
select c from Country c where c.name like 'G%'

// Länder mit Vorwahl 1 oder 86
select c from Country c where c.phonePrefix in ('1','86')

// Länder mit gesetztem Autokennzeichen
select c from Country c where c.carCode is not null

// Mitarbeiter mit eingetragenen Skills
select e from Employee e where e.skills is not empty

// Mitarbeiter mit Kenntnissen in JPA
select e from Employee e where 'JPA' member of e.skills

// Verlage mit min. einem Buch über JPA
select p from Publisher p where exists
  (select b from Book b where b.publisher=p and b.name like '%JPA%')
```

Listing 3.56: Weitere Selektionsbeispiele

Das Muster des *like*-Operators kann die Zeichen *'_'* und *'%'* enthalten als Platzhalter für ein bzw. mehrere (auch 0) beliebige Zeichen. Sollen diese Zeichen ohne ihre Sonderbedeutung verwendet werden, müssen ihnen ein frei wählbares Escape-Zeichen vorangestellt werden: *select … where c.name like '_xyz' excape '\'* sucht nach Einträgen mit Namen *'_xyz'*.

Die Operatoren *all* und *any* lassen sich nutzen, um einen Wert mit allen von einer Subquery selektierten Werten zu vergleichen. *all* liefert dabei ein positives Ergebnis, wenn der Vergleich für alle Sub-Ergebnisse gilt, *any* benötigt dafür nur einen positiven Vergleich. *some* kann synonym für *any* verwendet werden (Listing 3.57).

```
// Verlage mit nur dünnen Büchern (bis 200 Seiten)
select p from Publisher p
  where 200 >= all ( select b.pages from Book b where b.publisher=p)

// Verlage mit sehr dicken Büchern (mehr als 800 Seiten)
select p from Publisher p
  where 800 < any ( select b.pages from Book b where b.publisher=p)
```

Listing 3.57: Vergleich mit den Ergebnissen einer Subquery

Parameter

JPQL kann nummerierte oder benannte Parameter in der Where-Klausel enthalten. Nummerierte oder Positionsparameter haben die Form *?n*, wobei *n* eine ganze Zahl beginnend mit *1* ist. Benannte Parameter werden als *:name* notiert, worin *name* ein für die Query eindeutiger Identifier ist. Die Parameter dürfen in der Query mehrfach in beliebiger Reihenfolge auftauchen. Zur Versorgung der Query-Parameter mit konkreten Werten dient die Methode *Query.setParameter* (Listing 3.58).

```
TypedQuery<Country> query = em.createQuery(
  "select c from Country c where c.carCode=?1 or c.phonePrefix=:pp",
  Country.class);
query.setParameter(1, "D");
query.setParameter("pp", "39");
```

Listing 3.58: Query-Parameter

Da die JPQL-Ausdrücke dynamisch sind, d. h. erst zur Laufzeit erstellt werden, könnte man auf die Idee kommen, den Query-Text mittels String-Konkatenation zusammenzubauen, statt Parameter einzusetzen (Listing 3.59). Das ist aber aus zweierlei Gründen eine schlechte Vorgehensweise. Zum einen wird der Programmcode eher unleserlicher, zumal man sich mit den Besonderheiten der Datentypen explizit herumschlagen muss (z. B. Texte in Hochkommas, Hochkommas als Textinhalt verdoppeln etc.). Zum anderen verhindert man damit eine Optimierung in der Datenbank, die bei parametrisierten Ausdrücken möglich ist, nämlich die einmalige Analyse des Query-Befehls und Aufstellung eines entsprechenden Ausführungsplans. Also: Liebe Programmierer, machen Sie das hier nicht nach!

```
String cc = …;
String pp = …;
TypedQuery<Country> query = em.createQuery(
```

```
"select c from Country c where c.carCode='" + cc + "'
                        or c.phonePrefix='" + pp + "'",
Country.class);
```

Listing 3.59: String-Konkatenation als schlechter Ersatz für Query-Parameter

Aggregationsfunktionen

In der Select-Klausel können die Aggregationsfunktionen *avg*, *count*, *max*, *min*, und *sum* für Durchschnitt, Anzahl, Maximum, Minimum und Summe verwendet werden. Das Selektionsergebnis reduziert sich dann auf einen Eintrag, d. h. die Ergebnismenge wird entsprechend den verwendeten Funktionen verdichtet (Listing 3.60).

```
// Maximale Seitenzahl aller Bücher
select max(b.pages) from Book b

// Anzahl Bücher eines Verlages
select count(b) from Book b where b.publisher.name='O''Melly Publishing'
```

Listing 3.60: Aggregationsfunktionen

In Zusammenhang mit der weiter unten eingeführten Gruppierung können auch mehrzeilige Ergebnisse geliefert werden.

Funktionen

JPQL kennt weitere Funktionen, die in Select-, Where- und Having-Klausel verwendet werden können. Die folgenden Funktionen liefern einen Text:

- *concat(text, text …)*: Textverkettung
- *substring(text, start* [*,länge*]*)*: Teiltext
- *trim(text)*: Text ohne führende und folgende Leerzeichen
- *lower(text)*: Text in Kleinbuchstaben
- *upper(text)*: Text in Großbuchstaben

Bei *trim* kann sogar noch angegeben werden, ob nur führende oder folgende Zeichen entfernt werden sollen, und welches Zeichen entfernt werden soll: *trim(*[*leading* | *trailing* | *both*] [*zeichen*] *from text)*.

Als numerische Funktionen stehen zur Verfügung:

- *abs(zahl)*: Absolutwert
- *index(entity)*: Position eines Entity-Objekts in einer geordneten Liste
- *length(text)*: Textlänge
- *locate(text, teiltext* [*,start*]*)*: Teiltextposition

- *mod(zahl, teiler)*: Ganzzahliger Rest
- *size(collection)*: Anzahl Elemente
- *sqrt(zahl)*: Quadratwurzel

Zur Ermittlung von aktuellem Datum bzw. Zeit:

- *current_date*: Aktuelles Datum
- *current_time*: Aktuelle Uhrzeit
- *current_timestamp*: Aktueller Timestamp

Schließlich ermöglichen Case-Ausdrücke eine Art eingebettete Fallunterscheidung ähnlich dem Java-Operator *?:*:

- *case when b then w1 else w2 end*: Wenn Bedingung *b* gilt, dann *w1*, sonst *w2*

Seit JPA 2.1 ist es darüber hinaus möglich, weitere vordefinierte oder selbstentwickelte Funktionen der Datenbank aufzurufen:

- *function(name* [*,parameter* …]*)*: Aufruf der Funktion *name*

Für weitere Details sei auf die Spezifikation verwiesen („Scalar Expressions“).

Groupby- und Having-Klauseln

Die Selektionsergebnisse können gruppenweise verdichtet werden. Dazu gibt man nach *group by* ein oder mehrere (kommagetrennte) Variablen oder Attribute an. Im Ergebnis gibt es dann für jede vorkommende Kombination der Gruppierungsattribute einen Eintrag. Die dorthin selektierten Werte müssen für jede Gruppe konstant sein oder mithilfe einer Aggregationsfunktion ermittelt werden (Listing 3.61).

```
// Kontinente mit dem Durchschnitt der Landeseinwohnerzahlen
select c.continent, avg(c.population) from Country c group by c.continent
```

Listing 3.61: Gruppierung des Selektionsergebnisses

having ist das *where* der Gruppierung. Die Bedingungen in der Having-Klausel filtern also die zuvor gebildeten Gruppen (Listing 3.62).

```
// Kontinente mit Einwohnerdurchschnitt, aber nur für mehr als 10 Mio
select c.continent, avg(c.population) from Country c
  group by c.continent having avg(c.population)>100000000
```

Listing 3.62: Filterung der gebildeten Gruppen

Orderby-Klausel

Das Selektionsergebnis kann schließlich noch sortiert werden, indem nach *order by* ein oder mehrere (kommagetrennte) Attribute angegeben werden, nach denen die Ergebnis-

liste angeordnet werden soll. Die Sortierattribute können mit dem Zusatz *asc* oder *desc* für aufsteigendes oder absteigendes Sortieren versehen werden, voreingestellt ist *asc* (Listing 3.63).

```
// Länder absteigend nach Einwohnerzahl sortiert
select c from Country c order by c.population desc
```

Listing 3.63: Anordung des Selektionsergebnisses

Constructor Expressions

Werden mehrere Werte in der Select-Klausel angegeben, wird also ein Tupel von Werten selektiert, so wird dieses Tupel wie oben dargestellt in ein *Object[]*-Objekt verpackt. Damit geht ein Teil Übersichtlichkeit und Fehlersicherheit im Programmcode verloren, da man in der weiteren Verarbeitung des *Object*-Arrays wissen muss, unter welchem Index man welches fachliche Datum mit welchem konkreten Typ abgreifen kann (Listing 3.64).

```
EntityManager em = …
TypedQuery<Object[]> query = em.createQuery(
  "select c.continent, avg(c.population)"
  + " from Country c group by c.continent", Object[].class);
for (Object[] continentDescription : query.getResultList())
{
  Continent continent = (Continent) continentDescription[0];
  Number averagePopulation = (Number) continentDescription[1];
  …
```

Listing 3.64: Selektion von Tupeln als Object-Arrays

Mithilfe einer sog. Constructor Expression in der Select-Klausel können die Selektion und Weiterverarbeitung typsicherer und „sprechender" gestaltet werden. Dazu ist eine Hilfsklasse nötig, die einen zu den selektierten Werten passenden Konstruktor anbietet (Listing 3.65).

```
public class ContinentDescription
{
  public ContinentDescription(Continent continent, Number avgPopulation)
  {
  …
```

Listing 3.65: Hilfsklasse für die Selektion einer Constructor Expression

Eine Constructor Expression ist nun die Nutzung des beschriebenen Konstruktors in der Select-Klausel. Die Syntax entspricht einem normalen Kostruktoraufruf im Java-Code inklusive des Operators *new*, allerdings muss hier der Klassenname voll qualifiziert angegeben werden. Die Query liefert dann statt *Object[]* Objekte des Hilfstyps (Listing 3.66).

```
EntityManager em = …
TypedQuery<Object[]> query = em.createQuery(
  "select new de.….ContinentDescription(c.continent, avg(c.population))"
  + " from Country c group by c.continent", ContinentDescription.class);
for (ContinentDescription continentDescription : query.getResultList())
{
  Continent continent = continentDescription.getContinent();
  Number averagePopulation = continentDescription.getAveragePopulation();
…
```

Listing 3.66: Selektion von Tupeln mithilfe einer Constructor Expression

Paging

Eine Query, die ein Listenergebnis liefert, kann auf einen bestimmten Teil des Ergebnisses eingeschränkt werden, z. B. auf die Anzahl Einträge, die sich auf einer Dialogseite o. Ä. anzeigen lassen. Dazu bietet Query die Methoden *setFirstResult* und *setMaxResults* an, mit denen die Startposition in der Liste – beginnend mit 0 – und die Anzahl der gelieferten Einträge angegeben werden können (Listing 3.67).

```
// Länder nach Namen sortiert, Einträge 20 bis 29
EntityManager em = …
TypedQuery<Country> query
  = em.createQuery("select c from Country c order by c.name",
                Country.class);
query.setFirstResult(20);
query.setMaxResults(10);
List<Country> countries = query.getResultList();
```

Listing 3.67: Einschränken der Treffermenge einer Query

Query Hints

Java Persistence hat natürlich den Anspruch, die Entwicklung von DB-Zugriffen möglichst unabhängig von Provider und Datenbank zu gestalten. Dennoch hat man sich an einigen Stellen ein Hintertürchen offen gelassen, so auch bei Queries über die sog. Hints. Diese können mithilfe der Methode *setHint* als Schlüssel/Wert-Paare der Query hinzugefügt werden. Der einzige derzeit in der Spezifikation definierte Hint hat den Schlüssel *javax.persistence.query.timeout* und setzt eine Maximalzeit für die Durchführung von Queries in Millisekunden (Listing 3.68).

```
// Query-Ausführung auf 1 s einschränken
TypedQuery<Country> query = …
query.setHint("javax.persistence.query.timeout", 1000);
List<Country> countries = query.getResultList();
```

Listing 3.68: Einschränkung der Query-Laufzeit mit einem Hint

Hints müssen vom Provider nicht beachtet werden. Darüber hinaus dürfen die Hersteller weitere Hints definieren, solange der Schlüssel nicht mit *javax.persistence* beginnt. Portable Anwendungen sollten nicht von der Wirksamkeit der Hints abhängig sein.

Flush-Modus

Wenn eine Query ausgeführt wird, werden normalerweise zunächst alle im Entity Manager befindlichen Objektänderungen der aktuellen Transaktion in die Datenbank geschrieben, damit die Query die veränderten Daten auch „sieht". Dieses Verhalten kann mit der Methode *setFlushMode* beeinflusst werden, und zwar in *Query* für eine einzelne Query oder in *EntityManager* als Default für alle zukünftigen Queries. Die Methoden akzeptieren einen Parameter vom Typ *FlushModeType* – einem Aufzählungstyp mit zwei Konstanten:

- *FlushModeType.AUTO*: Flush der Änderungen vor Queries oder bei Commit
- *FlushModeType.COMMIT*: Flush der Änderungen nur bei Commit

Wird eine Query außerhalb von Transaktionen ausgeführt, werden keine Änderungen in die Datenbank geschrieben.

Fetch Joins

Im Abschnitt 3.3.6 wurde beschrieben, dass Relationsattibute ggf. *LAZY* geladen werden, d. h. sie werden nicht direkt mit dem sie enthaltenden Objekt geladen, sondern erst beim ersten Zugriff. Dieses Verhalten lässt sich für eine Query mit einem Fetch Join übersteuern. Die Syntax dazu ähnelt einem Outer Join, allerdings mit dem Zusatz *fetch* und ohne Angabe einer Query-Variablen.

```
// Normale Query; books werden nicht geladen (1:n-Relation, LAZY)
select p from Publisher p

// Query mit direktem Laden der books
select p from Publisher p left join fetch p.books
```

Listing 3.69: Fetch Join

Der Fetch Join ist auch als Inner Join möglich – ohne das Schlüsselwort *left* –, was aber meist weniger sinnvoll ist, da es hier nicht um eine Einschränkung der Ergebnismenge geht, sondern um das Übersteuern des Lazy Loadings.

Seit JPA 2.1 ist es möglich, das Ladeverhalten einer Query mithilfe von Entity Graphs zu beeinflussen. Das Vorgehen ist analog zur Nutzung von Entity Graphs für die *find*-Methode (s. 3.3.7 Entity Graphs). Die Hints werden der Query mithilfe der Methode *setHint* übergeben.

Named Queries

Neben der bislang gezeigten Möglichkeit, JPQL-Ausdrücke dynamisch zu erstellen, d. h. als String-Parameter an *EntityManager.createQuery* zu übergeben, bietet Java Persistence auch an, Queries unter einem frei gewählten Namen vorzudefinieren und sich bei der Ausführung dann nur auf den Namen zu beziehen. Dazu dient die Annotation *@NamedQuery*, die einer Entity-Klasse (oder einer Mapped Superclass – s. später) mitgegeben wird. Sie verknüpft einen Namen mit einem JPQL-Ausdruck. Achtung: Der Name der Named Query muss für die Persistence Unit eindeutig sein, er ist also nicht etwa der Entity-Klasse zugeordnet, die annotiert wird (Listing 3.70).

```
@Entity
@Access(AccessType.FIELD)
@NamedQuery(name = "Country_findByPhonePrefix",
            query = "select c from Country c where c.phonePrefix=?1")
public class Country
{
  …
}
```

Listing 3.70: Definition einer benannten Query

Für die Ausführung einer Named Query verwendet man *EntityManager.createNamedQuery* in der gleichen Weise wie zuvor *EntityManager.createQuery* (Listing 3.71).

```
EntityManager em = …
TypedQuery<Country> query
  = em.createNamedQuery("Country_findByPhonePrefix", Country.class);
query.setParameter(1, phonePrefix);
Country country = query.getSingleResult();
```

Listing 3.71: Ausführung einer Named Query

Sollen mehrere Named Queries einer Klasse hinzugefügt werden, müssen die dazu nötigen *@NamedQuery*-Annotationen in eine *@NamedQueries*-Annotation eingepackt werden, da der Java-Sprachstandard ja nicht mehrere gleichnamige Annotationen an einem Platz erlaubt.

Die *@NamedQuery*-Annotation akzeptiert die optionalen Parameter *lockMode* und *hints*, mit den der Query ein Lock-Modus und Query Hints beigefügt werden können. Locking wird in einem späteren Abschnitt beschrieben.

Die Verwendung von Named Queries anstelle der dynamischen Queries hat verschiedene Vorteile. Zum einen kann man Named Queries an zentraler Stelle – z. B. bei der Entity-Klasse, auf die sie sich i. W. beziehen – sammeln und dadurch für mehr Überblick sorgen. Zum anderen können Named Queries ggf. vorweg analysiert und mit einem Ausführungsplan versehen werden, was aber abhängig vom genutzten Provider und der Datenbank ist.

Mit JPA 2.1 ist es möglich geworden, Named Queries zur Laufzeit zu (re-)definieren, und zwar mithilfe der Methode *EntityManagerFactory.addNamedQuery*. Das übergebene Query-Objekt wird unter dem angegebenen Namen registriert. Eine evtl. schon existierende Named Query mit gleichem Namen wird dadurch ersetzt (Listing 3.72).

```
// Definition einer neuen Named Query
TypedQuery<Country> newQuery
  = em.createQuery("select c from Country c where c.continent=:cont",
                   Country.class);
emf.addNamedQuery("Country_findByContinent", newQuery);

…

// Nutzung der Named Query
TypedQuery<Country> query
  = em.createNamedQuery("Country_findByContinent", Country.class);
```

Listing 3.72: Definition einer Named Query zur Laufzeit

3.4.2 Native Queries

Für die Arbeit mit persistenten Daten sollten die Möglichkeiten von JPQL eigentlich ausreichen. Nun ist „eigentlich" in einem Satz – insbesondere in der Informationsverarbeitung – ein Signalwort, das auf Ausnahmen und höhere Aufwände hindeutet. Sollten Sie also mit JPQL nicht zum Ziel gelangen, können Sie SQL zum Aufbau von sog. Native Queries verwenden.

Die grundsätzliche Vorgehensweise entspricht dem zuvor gesagten: Mithilfe der Methode *EntityManager.createNativeQuery* wird ein Query-Objekt erstellt und mit *getResultList* oder *getSingleResult* ausgeführt (Listing 3.73).

```
EntityManager em = …
Query query = em.createNativeQuery(
  "SELECT NAME, POPULATION/AREA FROM CITY");
List<Object[]> resultList = query.getResultList();
for (Object[] entry : resultList)
{
  String name = (String) entry[0];
  Number populationDensity = (Number) entry[1];
  …
```

Listing 3.73: Aufbau und Ausführung einer SQL Query

Der Query-Text darf auch die von JDBC bekannten Positionsparameter '?' enthalten. Sie können mit der Methode *Query.setParameter* mit konkreten Werten besetzt werden. Namensparameter werden nicht unterstützt.

Leider liefert *createNativeQuery* „nur“ *Query* als Ergebnis, d. h. es stehen nicht wie bei *TypedQuery<T>* noch Informationen über den selektierten Typ zur Verfügung. Die Übernahme des Ausführungsergebnisses erzeugt also im Falle von *getResultList* ein Unchecked Warning, das man ggf. mit *@SuppressWarnings("unchecked")* unterdrückt, im Fall von *getSingleResult* ist sogar eine explizite Typwandlung nötig.

Native Queries bieten die Möglichkeit, beliebige SQL-Befehle auszuführen, also bspw. auch DB-spezifische Funktionen oder Stored Procedures zu nutzen. Diese Flexibilität kommt zu dem Preis, dass die im SQL genutzten Namen nun die DB-Namen sind, man also das Mapping zur Datenbank in jeder Query erneut berücksichtigen muss. Nutzt man darüber hinaus spezielle Eigenschaften der Zieldatenbank aus, wird ein DB-Wechsel aufwändig oder vielleicht sogar unmöglich.

Wie schon JPQL-Queries, lassen sich auch Native Queries vorformulieren. Dazu dient die Annotation *@NamedNativeQuery*, mit der auf Ebene einer Entity-Klasse (oder einer Mapped Superclass – siehe Abschnitt 3.5.4) SQL-Befehle mit einem Namen versehen werden können, der im Aufruf von *createNativeQuery* wiederum benutzt werden kann. *@NamedNativeQuery* akzeptiert den Parameter *resultSetMapping*, mit dem auf eine weitere Annotation , *@SqlResultSetMapping*, referenziert wird. Damit wird deklariert, welche Entities und Werte von der SQL Query geliefert werden. Ein einfaches Beispiel zeigt Listing 3.74.

```
@Entity
@Access(AccessType.FIELD)
@NamedNativeQuery(
  name = "City_populationDensity",
  query = "SELECT NAME, POPULATION/AREA AS DENSITY FROM CITY",
  resultSetMapping = "City_populationDensity")
@SqlResultSetMapping(
  name = "City_populationDensity",
  columns = { @ColumnResult(name = "NAME"),
              @ColumnResult(name = "DENSITY") })
public class City
{
  …
```

Listing 3.74: Vorwegdefinition einer SQL Query mit Mapping

Werden mehrere *@NamedNativeQuery*-Annotationen benötigt, müssen sie in die „Plural-Annotation“ *@NamedNativeQueries* verpackt werden.

Die Ausführung einer Named Native Query geschieht analog zu der einer Named (JPQL) Query (Listing 3.75).

```
EntityManager em = …
Query query = em.createNamedQuery("City_populationDensity");
List<Object[]> resultList = query.getResultList();
…
```

Listing 3.75: Ausführung einer Named Native Query

Die Aussage der *@SqlResultSetMapping*-Annotation im gezeigten Beispiel ist, dass die von der SQL-Abfrage gelieferten Spaltenwerte *NAME* und *DENSITY* als Einzelwerte im Ergebnis zu finden sein werden.

Seit JPA 2.1 ist es mithilfe von *@SqlResultSetMapping* möglich, analog zu den Constructor Expressions von JPQL mit den selektierten Werten den Konstruktor einer Klasse aufrufen zu lassen. Dazu dient der neue Parameter *classes*, der ein oder mehrere *@ConstructorResult*-Werte annimmt. Darin ist wiederum die Zielklasse angegeben sowie die Spaltenwerte in der Reihenfolge, in der der Konstruktor der Klasse sie erwartet (Listing 3.76).

```
@Entity
@NamedNativeQuery(
  name = "City_populationDensityCtorResult",
  query = "SELECT NAME, POPULATION/AREA AS DENSITY FROM EEDEMOS_CITY",
  resultSetMapping = "City_populationDensityCtorResult")
@SqlResultSetMapping(
  name = "City_populationDensityCtorResult",
  classes = @ConstructorResult(
             targetClass = PopulationDensity.class,
             columns = { @ColumnResult(name = "NAME"),
                         @ColumnResult(name = "DENSITY") }))
public class City
{
  …
```

Listing 3.76: Nutzung von „@ConstructorResult“ in native Queries

Die damit erzeugten Objekte befinden sich dann anstelle der Einzelwerte im Query-Ergebnis. Im Beispiel (Listing 3.77) enthält die Ergebnisliste statt Einträgen des Typs *Object[]* *PopulationDensity*-Objekte.

```
Query query = em.createNamedQuery("City_populationDensityCtorResult");
List<PopulationDensity> resultList = query.getResultList();
```

Listing 3.77: Constructor Result im Query-Ergebnis

Weitere Möglichkeiten von *@SqlResultSetMapping* sind die Zuordnung der Spaltenwerte zu Entity-Objekten oder eine Kombination von Objekten und Einzelwerten. Ein solches komplexes Mapping ist auch ohne Named Query mit einer Variante von *EntityManager. createNativeQuery* möglich, die als zweiten Parameter den Namen eines Result Set Mappings annimmt. Komplexe Mappings werden in der Praxis selten benötigt. Für Details dazu sei daher hier nur auf die Java-Persistence-Spezifikation (SQL Queries) verwiesen.

Ebenfalls neu in JPA 2.1 ist die Möglichkeit, Queries auf Stored Procedures abzustützen. Dazu dienen die Annotation *@NamedStoredProcedureQuery*, mit der Stored Procedures als benannte Queries deklariert werden können, sowie die zugehörige Methode *EntityManager. createNamedStoredProcedureQuery*. Gegenüber dem direkten Aufruf von Stored Proce-

dures in Native Queries bieten Stored Procedure Queries eine erweiterte Behandlung der Prozedurparameter: Nach der Ausführung einer Query können die Werte der bei Stored Procedures verbreiteten OUT- oder INOUT-Parameter abgefragt werden. Für Details sei erneut auf die Spezifikation verwiesen („Stored Procedures“).

3.4.3 Criteria Queries

Java Persistence bietet mit JPQL eine leistungsfähige Abfragesprache an, deren Befehle aus Sicht des Java-Programms allerdings einfache Strings sind. Dadurch können sich leicht Fehler einschleichen, die in aller Regel erst zur Laufzeit erkannt werden. Neben so trivialen Fehlern wie vergessenen oder falsch geschriebenen Befehlswörtern sowie Objekt- und Attributnamen (Case-sensitiv!) kommen auch subtilere Fehlersituationen vor. So ist bspw. *select c from Country* fehlerhaft, da in JPQL verpflichtend eine Query-Variable genutzt werden muss, anders als in SQL, wo dies nur optional ist. Auch ist die Typsicherheit nicht 100 %tig, denn kein Compiler kann prüfen, ob der in *createQuery* angegebene JPQL-Text zum Typparameter passt (Listing 3.78).

```
TypedQuery<City> query
  = em.createQuery("select c from Country c", City.class);
```

Listing 3.78: Mismatch zwischen JPQL und Typparameter

An dieser Stelle setzt das Criteria API an, eine Schnittstelle zum objektorientierten Aufbau von Queries. *EntityManager* liefert durch die Methode *getCriteriaBuilder* ein Objekt vom Typ *CriteriaBuilder*, das den Zugang zur Criteria API darstellt. Ein damit erzeugtes Query-Objekt wird anstelle des bisherigen JPQL-Texts verwendet. Die Ausführung der Query geschieht auf gleiche Weise wie zuvor (Listing 3.79).

```
EntityManager em = …
CriteriaBuilder cBuilder = em.getCriteriaBuilder();
CriteriaQuery<Cocktail> cQuery = cBuilder.createQuery(Cocktail.class);
// Füllen des Query-Objektes
…
TypedQuery<Cocktail> query = em.createQuery(cQuery);
List<Cocktail> cocktails = query.getResultList();
```

Listing 3.79: Grundsätzliche Vorgehensweise zur Nutzung von Criteria Queries

Eine Anmerkung des Autors: Die Beispiele dieses Abschnitts verwenden die Entity-Klasse *Cocktail*. Ich habe mich natürlich gefragt, ob es politisch korrekt ist, indirekt Alkohol in einem Fachbuch zu verwenden. Die Beispieldaten enthalten aber auch einen nichtalkoholischen Cocktail!

Query Roots und Selektion

Für eine Abfrage muss zunächst bestimmt werden, auf welche Entities sie sich bezieht. Diese sog. Query Roots entsprechen des Query-Variablen der From-Klausel in JPQL. Sie werden der *CriteriaQuery* mithilfe der Methode *from* hinzugefügt.

Sollen die durch ein Query Root repräsentierten Objekte Ergebnis der Query sein, wird das Root-Objekt der Query mittels *select* als Selektion übergeben. Eine mit diesen beiden Mitteln aufgebaute Query, die alle Einträge der Entity als Ergebnis liefert, zeigt Listing 3.80.

```
// Criteria Query äquivalent zu "select c from Cocktail c"
EntityManager em = …
CriteriaBuilder cBuilder = em.getCriteriaBuilder();
CriteriaQuery<Cocktail> cQuery = cBuilder.createQuery(Cocktail.class);
Root<Cocktail> c = cQuery.from(Cocktail.class);
cQuery.select(c);
TypedQuery<Cocktail> query = em.createQuery(cQuery);
…
```

Listing 3.80: Criteria Query zur Selektion aller Objekte einer Entity-Klasse

An dem Beispiel wird deutlich, dass der Programmaufwand im Vergleich zu einer in JPQL formulierten Query zwar signifikant höher ist, dafür aber auch mehr Schutz vor Fehlern geboten wird. So muss bspw. der Parameter von *select* zum Typ der Query passen.

Der Programmaufwand kann übrigens etwas verringert werden, da die Methoden in den allermeisten Fällen einen Return-Wert liefern, mit dem direkt die nächste Methode aufgerufen werden kann – eine Ausprägung eines sog. Fluent API (Listing 3.81).

```
CriteriaQuery<Cocktail> cQuery = cBuilder.createQuery(Cocktail.class);
TypedQuery<Cocktail> query
  = em.createQuery(cQuery.select(cQuery.from(Cocktail.class)));
…
```

Listing 3.81: Alternative „Fluent API“

Welche Schreibweise Sie übersichtlicher finden, ist natürlich Ihnen überlassen. Im Folgenden wird nicht vom Fluent API Gebrauch gemacht, da Einzelanweisungen im Buch etwas satzfreundlicher sind.

Neben einzelnen kompletten Objekten können mithilfe von Criteria Queries auch mehrere Werte – Entities, Attribute, berechnete Werte – selektiert werden. Details dazu folgen später.

Attributzugriffe

Auf die Teile eines Entity-Objekts kann ausgehend vom Query Root mithilfe der Methode *get* zugegriffen werden. Sie liefert ein Objekt des Typs *Path* als Repräsentant für das referenzierte Attribut. *Path*-Objekte können dann bspw. zur Selektion verwendet werden (Listing 3.82).

```
CriteriaQuery<String> cQuery = cBuilder.createQuery(String.class);
Root<Cocktail> c = cQuery.from(Cocktail.class);
Path<String> cName = c.get("name");
cQuery.select(cName);
TypedQuery<String> query = em.createQuery(cQuery);
…
```

Listing 3.82: Zugriff auf Attribute über ihren Namen

Auf ein *Path*-Objekt lässt sich wieder *get* anwenden, sodass man damit auf Teile von eingebetteten Objekten oder 1:1- bzw. n:1-Relationsattributen zugreifen kann.

Achtung: Die im Beispiel genutzte Version von *get* erhält den gewünschten Attributnamen als *String*-Parameter. Damit ist zur Compile-Zeit wieder keine Prüfung möglich, ob das Attribut überhaupt existiert!

Statisches Metamodell

Das beschriebene Problem lässt sich lösen, wenn schon zur Übersetzungszeit statische Informationen zum dynamischen Aufbau der Entity-Objekte vorliegen. Anders als z. B. C++ bietet Java diese statischen Metadaten nicht von Hause aus an, sodass sie mit einem entsprechenden Werkzeug generiert werden müssen. Zu jeder persistenten Klasse *E* wird dabei eine Metadatenklasse *E_* erzeugt, die für jedes persistente Attribut von *E* eine statische Variable gleichen Namens enthält, deren Typ Auskunft über die Art des persistenten Attributs gibt (Listing 3.83).

```
@Entity
@Access(AccessType.FIELD)
public class Cocktail
{
  @Id @GeneratedValue
  private Integer    id;

  private String     name;

  @ManyToMany
  private Set<CocktailZutat> zutaten = new HashSet<CocktailZutat>();
  …
}

@StaticMetamodel(Cocktail.class)
public abstract class Cocktail_
```

```
{
  public static volatile SingularAttribute<Cocktail, Integer> id;
  public static volatile SingularAttribute<Cocktail, String> name;
  public static volatile SetAttribute<Cocktail, CocktailZutat> zutaten;
}
```

Listing 3.83: Metamodellklasse zu einer Entity-Klasse

Der Name *E_* entspricht nur einer Namenskonvention. Die Verbindung zwischen den beiden Klassen wird über die Annotation *@StaticMetamodel*, was für den hier gezeigten Einsatz der Metamodellklasse aber unerheblich ist.

Die Metadaten enthalten offensichtlich Informationen über die betroffene Klasse und den Typ der Attribute. Die weiteren Details werden hier übergangen, da sie für das weitere Verständnis nicht wichtig sind. Bei Interesse schauen Sie in den entsprechenden Abschnitt der Java-Persistence-Spezifikation („Metamodel API“).

Die Metamodellklassen lassen sich mit einem Annotation Processor generieren, z. B. dem Hibernate-Modellgenerator. Seine Benutzung ist natürlich abhängig vom genutzten Build-System und daher hier nicht allgemeingültig darstellbar. Das Beispielprojekt enthält dazu entsprechende Maven-Plug-in-Aufrufe (Listing 3.84). In Eclipse kann der Annotation Processor in den Projekteigenschaften im Abschnitt Java Compiler | Annotation Processing eingetragen werden (Abbildung 3.20).

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <inherited>true</inherited>
      <configuration>
        <generatedSourcesDirectory>
          target/generated-sources/annotations
        </generatedSourcesDirectory>
      </configuration>
    </plugin>

    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>build-helper-maven-plugin</artifactId>
      <inherited>true</inherited>
      <executions>
        <execution>
          <id>add-source</id>
          <phase>generate-sources</phase>
          <goals>
            <goal>add-source</goal>
          </goals>
          <configuration>
```

```
            <sources>
              <source>target/generated-sources/annotations</source>
            </sources>
          </configuration>
        </execution>
      </executions>
    </plugin>
```

Listing 3.84: Ausschnitt aus der Maven-Projektkonfiguration zur Erzeugung des Metamodells

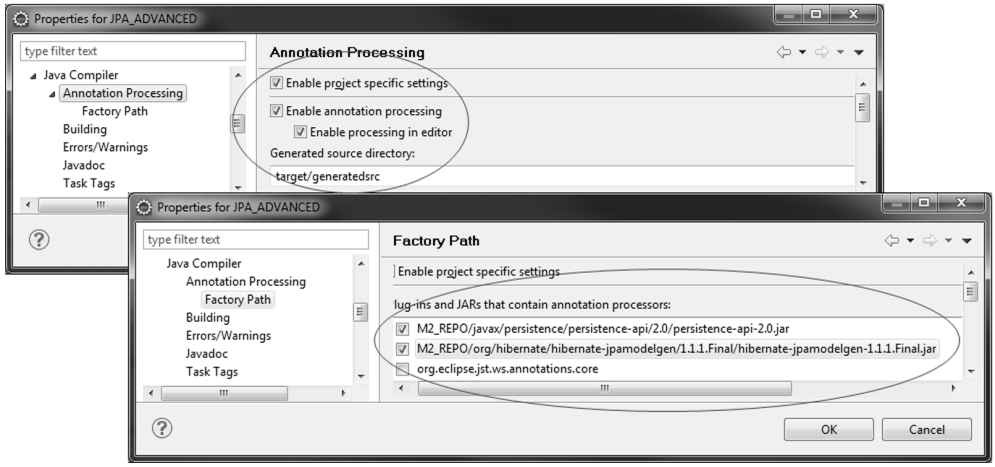

Abbildung 3.20: Eclipse-Projektkonfiguration zur Generierung von Metamodellklassen

Mithilfe des statischen Metamodells lässt sich die oben gezeigte Anweisung zum Zugriff auf ein Attribut nun sicherer gestalten (Listing 3.85).

```
Path<String> cName = c.get(Cocktail_.name);
```

Listing 3.85: Attributzugriff mittels Metamodellattribut

Bedingungen

CriteraBuilder bietet diverse Methoden an, mit denen sich Attribute – in Form von *Path*-Objekten – und andere Werte zu Prädikaten, d. h. Bedingungsobjekten, kombinieren lassen. So prüft ein mit *CriteriaBuilder.equal* erstelltes Prädikat seine beiden Parameter auf Gleichheit. Analog funktionieren *greaterThan*, *isFalse*, *isNotNull* etc.

Ein oder mehrere Prädikate (implizit Und-verknüpft) lassen sich der Query als Where-Bedingung mithilfe der Methode *where* hinzufügen (Listing 3.86).

```
CriteriaQuery<Cocktail> cQuery = cBuilder.createQuery(Cocktail.class);
Root<Cocktail> c = cQuery.from(Cocktail.class);
```

```
Path<String> cName = c.get(Cocktail_.name);
Predicate ipanema = builder.equal(cName, "Ipanema");
cQuery.where(ipanema);
…
```

Listing 3.86: Nutzung eines Prädikats als Where-Bedingung

Joins

Für die Navigation in Relationen werden *Join*-Objekte benötigt. Sie werden ausgehend von einem Query Root mithilfe der Methode *join* erstellt – am besten wieder unter Verwendung statischer Metamodelldaten – und können in der Folge wiederum wie Query Roots genutzt werden, z. B. um darauf Prädikate zu definieren (Listing 3.87).

```
// Criteria Query äquivalent zu
// select distinct c from Cocktail c JOIN c.zutaten z where z.volProz<>0"
CriteriaBuilder cBuilder = em.getCriteriaBuilder();
CriteriaQuery<Cocktail> cQuery = cBuilder.createQuery(Cocktail.class);
Root<Cocktail> c = cQuery.from(Cocktail.class);
Join<Cocktail, Zutat> z = c.join(Cocktail_.zutaten);
Path<Double> zVolProz = z.get(Zutat_.volProz);
Predicate enthaeltAlkohol = cBuilder.equal(zVolProz, 0);
cQuery.where(enthaeltAlkohol);
cQuery.select(c);
cQuery.distinct(true);
TypedQuery<Cocktail> query = em.createQuery(cQuery);
```

Listing 3.87: Nutzung eines Joins in einer Criteria Query

Das Beispiel zeigt auch die Anwendung von *distinct* zur Vermeidung von Dubletten im Ergebnis.

Die Methode *join* akzeptiert als zweiten Parameter einen Wert aus der Aufzählung *JoinType*:

- *JoinType.INNER*: Normaler (innerer) Join
- *JoinType.LEFT*: Left Outer Join
- *JoinType.RIGHT*: Right Outer Join

Die Variante mit nur einem Parameter nutzt implizit *JoinType.INNER*. Right Outer Joins müssen in JPA 2.x nicht unterstützt werden.

Parameter

CriteriaBuilder bietet mit der Methode *parameter* die Möglichkeit an, Query-Parameter in die Query einzubauen, z. B. als Teil eines Vergleichs. Diese werden vor der Ausführung der Query wie gewohnt mit aktuellen Werten besetzt (Listing 3.88).

```
CriteriaBuilder cBuilder = em.getCriteriaBuilder();
CriteriaQuery<Cocktail> cQuery = cBuilder.createQuery(Cocktail.class);
Root<Cocktail> c = cQuery.from(Cocktail.class);
cQuery.where(cBuilder.equal(c.get(Cocktail_.name),
            cBuilder.parameter(String.class, "name")));
cQuery.select(c);
TypedQuery<Cocktail> q = em.createQuery(cQuery);
q.setParameter("name", name);
```

Listing 3.88: Criteria Query mit Parameter

Tupel-Selektion

Zur Selektion mehrerer Werte bietet das Criteria API wie schon JPQL zwei Varianten an. Zum einen kann man sich des Query-Typs *Tuple* bedienen. Ein *Tuple*-Objekt ist ein Container für mehrere Teilobjekte, die daraus per Index oder Alias-Namen geliefert werden können. Für die Selektion wird dann statt *select* die Methode *multiSelect* verwendet, die eine unbrenzte Menge von Einzelselektionswerten als Parameter annimmt. Dabei kann man den einzelnen Selektionswerten mithilfe der Methode *alias* einen Aliasnamen mitgeben, den man später bei der Verarbeitung der Ergebnismenge nutzt (Listing 3.89).

```
CriteriaBuilder cBuilder = em.getCriteriaBuilder();
CriteriaQuery<Tuple> cQuery = cBuilder.createQuery(Tuple.class);
Root<Cocktail> c = cQuery.from(Cocktail.class);
cQuery.multiselect(c.get(Cocktail_.name).alias("cocktailName"),
                c.get(Cocktail_.basisZutat).get(CocktailZutat_.name));
TypedQuery<Tuple> query = em.createQuery(cQuery);
for (Tuple entry : query.getResultList())
{
  String name = entry.get("cocktailName", String.class);
  String basisZutat = entry.get(1, String.class);
  …
```

Listing 3.89: Tuple-Selektion

Die Alternative ist wie bei JPQL die Nutzung einer Constructor Expression. *CriteriaBuilder* bietet die Methode *construct* an, mit der die Constructor Expression erstellt werden kann (Listing 3.90).

```
CriteriaBuilder cBuilder = em.getCriteriaBuilder();
CriteriaQuery<NameAndBasisZutat> cQuery
  = cBuilder.createQuery(NameAndBasisZutat.class);
Root<Cocktail> c = cQuery.from(Cocktail.class);
cQuery.select(cBuilder.construct(NameAndBasisZutat.class,
  c.get(Cocktail_.name),
  c.get(Cocktail_.basisZutat).get(CocktailZutat_.name)));
TypedQuery<NameAndBasisZutat> query = em.createQuery(cQuery);
for (NameAndBasisZutat entry : query.getResultList())
{
```

```
String name = entry.getName();
String basisZutat = entry.getBasisZutat();
…
```

Listing 3.90: Criteria Query mit Constructor Expression

Funktionen

Für die im Abschnitt über JPQL beschriebenen Funktionen – inklusive der Aggregationsfunktionen – bietet CriteriaBuilder gleichnamige Methoden an, mit denen Selektions- oder Vergleichsausdrücke zusammengestellt werden können. Ein Beispiel folgt im nächsten Abschnitt. Dort wird die Aggregationsfunktion *avg* eingesetzt.

Gruppierung

Mithilfe der Methode *groupBy* können einer *CriteriaQuery* ein oder mehrere Gruppierungsparameter hinzugefügt werden. Wie bei JPQL beschrieben, wird die Ergebnismenge dann so verdichtet, dass für jede Kombination der Gruppierungsparameterwerte nur eine Ergebniszeile erscheint. Die Selektion muss dementsprechend aus für die jeweiligen Gruppen konstanten Werten oder Aggregationswerten bestehen. Sollen die Gruppenergebnisse weiter eingeschränkt werden, kann dies mit der Methode *having* geschehen, die so wirkt wie *where* in Bezug auf die nicht gruppierten Werte (Listing 3.91).

```
CriteriaBuilder cBuilder = em.getCriteriaBuilder();
CriteriaQuery<Tuple> cQuery = cBuilder.createTupleQuery();
Root<Cocktail> c = cQuery.from(Cocktail.class);
Join<Cocktail, Zutat> z = c.join(Cocktail_.zutaten);
Path<String> cName = c.get(Cocktail_.name);
Path<Double> zVolProz = z.get(Zutat_.volProz);
cQuery.multiselect(cName, cBuilder.avg(zVolProz));
cQuery.groupBy(cName);
TypedQuery<Tuple> q = em.createQuery(cQuery);
…
```

Listing 3.91: Criteria Query mit Gruppierung

Sortierung

Die Anordnung des Query-Ergebnisses kann mithilfe der Methode *orderBy* festgelegt werden. Sie erhält ein oder mehrere Sortierangaben als Parameter, die wiederum durch die Methoden *CriteriaBuilder.asc* oder *CriteriaBuilder.desc* für auf- bzw. absteigende Sortierung erstellt werden (Listing 3.92).

```
CriteriaBuilder cBuilder = em.getCriteriaBuilder();
CriteriaQuery<Cocktail> cQuery = cBuilder.createQuery(Cocktail.class);
Root<Cocktail> c = cQuery.from(Cocktail.class);
cQuery.select(c);
```

```
cQuery.orderBy(cBuilder.asc(c.get(Cocktail_.name)));
TypedQuery<Cocktail> query = em.createQuery(cQuery);
…
```

Listing 3.92: Sortierung des Selektionsergebnisses einer Criteria Query

Fetch Joins

Wie bei JPQL gibt es auch bei Criteria Queries eine Möglichkeit, Relationsattribute direkt laden zu lassen, auch wenn sie für Lazy Loading konfiguriert sind. Dazu bieten die Query Roots und Join-Objekte die Methode *fetch* an. Der erste Parameter bestimmt das *EAGER* zu ladende Attribut, der zweite gibt wie bei *join* den Join-Typ an (Listing 3.93).

```
Root<Publisher> p = cQuery.from(Publisher.class);
p.fetch(Publisher.books, JoinType.LEFT);
```

Listing 3.93: Fetch Join in einer Criteria Query

Wie schon bei JPQL-Fetch-Joins ausgeführt, ist der Join-Typ *LEFT* hier am sinnvollsten. Auf das Ergebnis der Methode *fetch* kann erneut *fetch* angewendet werden, sodass hier sogar mehrstufige Fetch-Joins möglich sind.

3.5 Vererbungsbeziehungen

Vererbung ist ein wichtiges Prinzip der Objektorientierung, mit dem man „Ist ein"-Beziehungen ausdrücken kann: Ein abgeleitetes Objekt **ist ein** Objekt der Basisklasse, d. h. es hat alle Eigenschaften und Funktionalitäten der Basisklasse, ergänzt um weitere. Technisch drückt sich die Vererbung auf der Attributebene dadurch aus, dass ein abgeleitetes Objekt alle Attribute seiner Basisklasse enthält.

Relationale Datenbanken haben ein solches Feature im Allgemeinen nicht. Diese konzeptionelle Lücke gilt es nun beim Mapping der Daten zu überbrücken. Dazu bietet Java Persistence drei verschiedene Strategien an, die im Folgenden anhand der bewusst einfach gehaltenen Klassen *Vehicle*, *Car* und *Ship* betrachtet werden sollen (Listing 3.94).

```
@Entity
@Access(AccessType.FIELD)
public abstract class Vehicle
{
  @Id @GeneratedValue
  private Integer id;
  private String  name;
  …
}

@Entity
```

```
@Access(AccessType.FIELD)
public class Car extends Vehicle
{
  private int noOfDoors;
  …
}

@Entity
@Access(AccessType.FIELD)
public class Ship extends Vehicle
{
  private double tonnage;
  …
}
```

Listing 3.94: Basisklasse „Vehicle“ mit davon abgeleiteten Klassen „Car“ und „Ship“

3.5.1 Mapping-Strategie SINGLE_TABLE

Diese Strategie ist vorgegeben, kann aber auch durch Annotation der Basisklasse mit *@Inheritance(strategy = InheritanceType.SINGLE_TABLE)* explizit gewählt werden (Listing 3.95). Die Daten der gesamten Ableitungshierarchie werden in einer einzelnen Tabelle abgespeichert. Eine zusätzliche Spalte – der Diskriminator – speichert für jeden Eintrag seinen Typ in Form des einfachen Klassennamens. Da die Tabelle Spalten für alle Attribute der beteiligten Klassen enthält, werden für einen Eintrag i. a. R. nicht alle benötigt. Die überzähligen Werte bleiben *null*.

```
@Entity
@Access(AccessType.FIELD)
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public abstract class Vehicle
{
  …
```

Listing 3.95: Explizite Wahl der Strategie zur Abbildung der Vererbung

vehicle	
DTYPE	varchar(31)
id	int
name	varchar(255)
noOfDoors	int
tonnage	double

Abbildung 3.21: Tabelle bei Strategie „SINGLE_TABLE“

DTYPE	id	name	noOfDoors	tonnage
Car	1	Peugeot 407SW	5	(null)
Car	2	Fiat Panda	3	(null)
Ship	3	Queen Mary II	(null)	76000.0

Abbildung 3.22: Beispielinhalt bei Strategie „SINGLE_TABLE“

Die Diskriminatorwerte können auch explizit gewählt werden, und zwar mithilfe der Annotation *@DiscriminatorValue*.

Der Name und der Typ der Diskriminatorspalte können mithilfe der Annotation *@DiscriminatorColumn* bestimmt werden. Als Parameter *discriminatorType* wird dabei eine Konstante aus dem Aufzählungstyp *DiscriminatorType* übergeben: Neben dem Vorgabewert *STRING* sind hier auch *CHAR* und *INTEGER* möglich, um statt Strings einzelne Zeichen oder ganze Zahlen als Diskriminatoren zu verwenden. In diesen Fällen muss für die beteiligten Klassen jeweils ein passender Diskriminatorwert mittels *@DiscriminatorValue* angegeben werden (Listing 3.96).

```
@Entity
@Access(AccessType.FIELD)
@DiscriminatorColumn(name = "T",
                     discriminatorType = DiscriminatorType.INTEGER)
@DiscriminatorValue("1234")
public abstract class Vehicle
{
  …
}

@Entity
@Access(AccessType.FIELD)
@DiscriminatorValue("2345")
public class Car extends Vehicle
{
  …
}
```

Listing 3.96: Einsatz von „@DiscriminatorColumn“ und „@DiscriminatorValue“

Die Strategie *SINGLE_TABLE* hat den Vorteil des einfachen Mappings zur Datenbank: Es wird nur eine Tabelle für die gesamte Ableitungshierarchie benötigt. Das ist für die Performanz von Queries durchaus positiv, da alle zur Konstruktion eines Objekts nötigen Attributwerte in einer Tabellenzeile enthalten sind. So sind sowohl für Queries nach den „konkretesten“ Objekten des Ableitungsbaums (z. B. *"select c from Car c …"*) – wie auch für polymorphe Queries auf Basisklassenebene (z. B. *"select v from Vehicle v …"*) nur einfache SQL-Befehle nötig.

Nachteilig ist hingegen, dass die Tabelle recht breit werden kann, da sie ja sämtliche Attribute der beteiligten Klassen enthält. Weiterhin können kaum einschränkende Constraints

wie *"not null"* auf die Spalten der Tabelle angewendet werden, da der überwiegende Teil nicht für alle Eintragstypen gefüllt ist. Schließlich können Fremdschlüssel in anderen Tabellen nur auf die oberste Basisklasse referenzieren: Ein Foreign Key auf *Ship* ist in der Datenbank nicht darstellbar, da für *Ship* ja keine (separate) Tabelle existiert.

3.5.2 Mapping-Strategie TABLE_PER_CLASS

Hier bekommt jede nicht abstrakte Klasse der Ableitungshierarchie eine Tabelle zugeordnet (Abbildung 3.23, Abbildung 3.24).

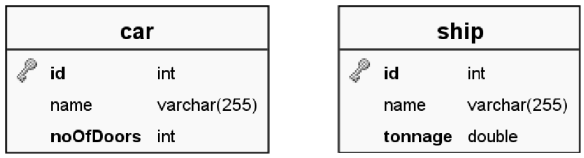

Abbildung 3.23: Tabellen bei Strategie „TABLE_PER_CLASS"

id	name	noOfDoors
1	Peugeot 407SW	5
2	Fiat Panda	3

id	name	tonnage
3	Queen Mary II	76000.0

Abbildung 3.24: Beispielinhalt bei Strategie „TABLE_PER_CLASS"

Jedes konkrete Java-Objekt ist nun in einem Tabelleneintrag ohne weiteren Ballast abgespeichert. Not-Null-Constraints können darin problemlos deklariert werden. Ebenso sind Fremdschlüsselbeziehungen auf die konkreten Objekte möglich, nicht aber auf Basisklassen.

Während konkrete Queries hier wiederum performant umgesetzt werden können, sind für polymorphe Queries komplexere SQL-Befehle wie bspw. *UNION* notwendig, was sich je nach DB-Fabrikat stark bemerkbar macht. Sie sollten diese Strategie also vermeiden, wenn die Anwendung häufiger Basisklassenabfragen durchführt.

Eine Einschränkung besteht auch in Bezug auf automatisch generierte Schlüssel. Der Identity-Generator ist hier prinzipbedingt nicht einsetzbar, da für die betroffenen Klassen übergreifend eindeutige ID-Werte generiert werden müssen, der Identity-Generator dies aber nur für jede Tabelle tun würde[10].

Die Spezifikation verlangt in der Version 2.x nicht, dass die Strategie *TABLE_PER_CLASS* unterstützt wird. Das ist aber ein nur theoretischer Nachteil, da alle mir bekannten Provider dies tun.

10 Wenn Sie die Generatorstrategie *AUTO* benutzen, wählt Hibernate (zumindest bis zur Version 4.0.0) bei einer Zieldatenbank, die Identity Columns unterstützt, den Generator *IDENTITY*, obwohl das für eine Ableitungshierarchie nicht möglich ist. Um dies zu umgehen, setzen Sie explizit *SEQUENCE* oder *TABLE* ein.

3.5.3 Mapping-Strategie JOINED

Diese Strategie legt die Daten der Ableitungshierarchie voll normalisiert ab: Jede Klasse hat eine ihr zugeordnete Klasse, die neben dem Primärschlüssel nur die in ihr deklarierten Attribute enthält (Abbildung 3.25, Abbildung 3.26)[11].

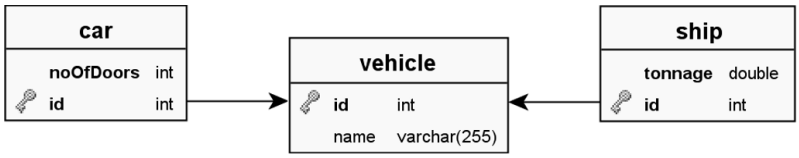

Abbildung 3.25: Tabellen bei Strategie „JOINED“

id	noOfDoors
1	5
2	3

id	name
1	Peugeot 407SW
2	Fiat Panda
3	Queen Mary II

id	tonnage
3	76000.0

Abbildung 3.26: Beispielinhalt bei Strategie „JOINED“

Durch das normalisierte Tabellenlayout sind Constraints auf den Spalten und Fremdschlüsselbeziehungen jeglicher Art unproblematisch. Nachteilig ist allerdings, dass für jeden Objektzugriff die Inhalte mehrerer Tabellen miteinander verknüpft werden müssen, was sich in teilweise dramatisch geringerer Performanz niederschlägt.

3.5.4 Non-Entity-Basisklassen

Nicht immer hat eine Basisklasse auch eine fachliche Bedeutung. So könnte man bspw. eine Basisklasse für Entities schaffen, die als ID eine UUID erhalten, d. h. einen (weitgehend) global eindeutigen *String*. Die Eigenschaften der Klasse sollen denen einer normalen Entity entsprechen, es sollen also die gleichen Möglichkeiten des Mappings bestehen etc. Wegen der nicht vorhandenen fachlichen Bedeutung wird aber keine eigene Tabelle benötigt, es werden keine Abfragen auf diese Klasse gerichtet usw. Für diesen Zweck bietet Java Persistence die Annotation *@MappedSuperclass*, die anstelle von *@Entity* verwendet wird. Im Beispiel stellt *UuidEntity* die UUID-basierte ID zusammen mit einem entsprechenden Getter und den Standardmethoden *equals* und *hashCode* zur Verfügung, sodass *Department* durch Ableitung auf einfache Art und Weise erstellt werden kann. (Listing 3.97).

11 Die Spezifikation erlaubt dem Provider die Nutzung einer Diskriminatorspalte wie bei *SINGLE_TABLE* auch bei der Strategie *JOINED*. Davon macht derzeit nach meiner Beobachtung nur EclipseLink Gebrauch.

```
@MappedSuperclass
@Access(AccessType.FIELD)
public abstract class UuidEntity
{
  @Id
  protected String id;

  public UuidEntity()
  {
    this.id = java.util.UUID.randomUUID().toString();
  }

  public String getId() { … }
  public boolean equals(Object obj) { … }
  public int hashCode() { … }
}

@Entity
@Access(AccessType.FIELD)
public class Department extends UuidEntity
{
  private String name;
  …
}
```

Listing 3.97: Einsatz von „@MappedSuperclass“

@MappedSuperclass findet man häufig in Kombination mit *abstract*, was aber keine Bedingung ist. *@MappedSuperclass* darf in einer Ableitungshierarchie auch zwischendrin vorkommen – wenngleich das auch eine recht unübliche Situation ist.

Vorsicht ist geboten, wenn man eine Entity-Klasse von einer Klasse ableitet, die nicht mit *@Entity* oder *@MappedSuperclass* markiert ist: Deren Attribute werden nicht persistent eingebunden, verhalten sich also so, als wären sie *@Transient*!

3.5.5 Polymorphe Queries

Queries sind per Default polymorph, d. h. es werden auch Objekte abgeleiteter Klassen einbezogen. Mithilfe der Funktion *type(x)* kann der Typ der selektierten Objekte erfragt und bspw. zur Einschränkung der Ergebnismenge verwendet werden (Listing 3.98).

```
// Selektion von Car-Objekten inkl. abgeleiteten Typen
em.createQuery("select v from Car v", Car.class) …
// Selektion von Objekten mit dem exakten Typ Car
em.createQuery("select v from Car v where type(v)=Car", Car.class) …
```

Listing 3.98: Einschränkung des Selektionsergebnnisses anhand des Typs

Seit JPA 2.1 ist es darüber hinaus möglich, mithilfe der Funktion *treat(x as T)* auf Eigenschaften abgeleiteter Klassen zuzugreifen. *Treat* stellt einen sog. Downcast auf den angegebenen Typ dar (Listing 3.99).

```
select v from Vehicle v
where treat(v as Lorry).payLoad>30
   or treat(v as Ship).tonnage>130000
```

Listing 3.99: Downcast von Query-Variablen

Treat kann auch zur Formulierung von Joins genutzt werden. Für weitere Details sei auf die Spezifikation verwiesen („Downcasting“).

3.6 Dies und das

3.6.1 Secondary Tables

Bislang war einer persistenten Klasse genau eine Tabelle zugeordnet, in der alle Attribute mit Ausnahme von *Collection*s abgespeichert wurden. Das muss aber nicht so sein: Eine Entity-Klasse kann neben ihrer Haupttabelle noch weitere Tabellen benutzen, die Attribute können also über mehrere Tabellen verteilt sein. Zur Deklaration einer Zusatztabelle dient die Annotation *@SecondaryTable*. Die Attribute der Klasse können dann der zweiten Tabelle zugeordnet werden, indem sie mit *@Column* annotiert werden und dabei der Tabellenname angegeben wird. Attribute ohne Tabellenangabe liegen in der Haupttabelle der Entity (Listing 3.100).

```
@Entity
@Access(AccessType.FIELD)
@Table(name = "EEDEMOS_COUNTRY")
@SecondaryTable(name = "EEDEMOS_COUNTRY_EXT")
public class Country
{
  @Id
  private String isoCode;

  private String name;

  @Column(table = "EEDEMOS_COUNTRY_EXT")
  private String phonePrefix;

  @Column(table = "EEDEMOS_COUNTRY_EXT")
  private String carCode;
  …
```

Listing 3.100: Deklaration einer Zusatztabelle

Die Zuordnung geschieht dabei über gemeinsame Primärschlüsselwerte. Entities mit Secondary Tables stellen somit in etwa eine implizite 1:1-Relation mit Join über die Primärschlüssel dar (Abbildung 3.27).

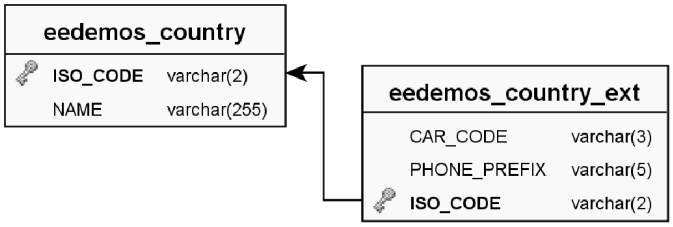

Abbildung 3.27: Tabellen zu Listing 3.100

Falls nötig, kann der Name der Primärschlüsselspalte(n) der Zusatztabelle angepasst werden. Dazu wird *@SecondaryTable* der Parameter *pkJoinColumns* mitgegeben.

Soll mehr als nur eine Zusatztabelle verwendet werden, müssen die entsprechenden *@SecondaryTable*-Annotationen in eine *@SecondaryTables*-Annotation verpackt werden (Listing 3.101).

```
@Entity
@Access(AccessType.FIELD)
@Table(name = "EEDEMOS_COUNTRY")
@SecondaryTables({
  @SecondaryTable(name = "EEDEMOS_COUNTRY_EXT"),
  @SecondaryTable(name = "EEDEMOS_COUNTRY_EXT2",
                  pkJoinColumns = @PrimaryKeyJoinColumn(name = "IC"))
})
public class Country
{
  …
```

Listing 3.101: Zuordnung mehrerer Zusatztabellen

3.6.2 Zusammengesetzte IDs

Es ist nicht immer möglich, auf mehrteilige IDs zu verzichten. In einem solchen Fall muss eine ID-Klasse entworfen werden, die die gewünschten ID-Attribute enthält. Die Klasse muss serialisierbar sein, einen parameterlosen Konstruktor sowie gültige Implementierungen von *equals* und *hashCode* besitzen (Listing 3.102).

```
@Embeddable
@Access(AccessType.FIELD)
public class BranchId implements Serializable
{
  private int companyId;
```

```
  private int branchNo;

  public boolean equals(Object obj) { … }
  public int hashCode() { … }

  // Getter, Setter, Konstruktoren, …
}
```

Listing 3.102: ID-Klasse

In der Entity-Klasse kann die ID-Klasse nun zur Deklaration eines ID-Attributs genutzt werden. Anstelle der sonst üblichen Annotation *@Id* wird nun *@EmbeddedId* verwendet (Listing 3.103).

```
@Entity
@Access(AccessType.FIELD)
public class SuperMarket
{
  @EmbeddedId
  private BranchId id;
  …
```

Listing 3.103: Eingebettetes ID-Attribut

Alternativ können die ID-Attribute auch in der Entity-Klasse einzeln aufgeführt werden. Die ID-Klasse muss dann mittels *@IdClass* deklariert werden. Sie benötigt für dieses Szenario die *@Embeddable*-Annotation nicht unbedingt. Wichtig ist hier die Namensgleichheit der Attribute in der ID-Klasse und der Entity-Klasse (Listing 3.104). Dieses Verfahren wird Composite ID genannt.

```
@Entity
@Access(AccessType.FIELD)
@IdClass(BranchId.class)
public class SuperMarket
{
  @Id
  private int    companyId;

  @Id
  private int    branchNo;
  …
```

Listing 3.104: Mehrfache Anwendung von „@Id“

3.6.3 Dependent IDs

Bei mehrteiligen IDs – aber nicht nur da – kommt es häufig vor, dass ein ID-Attribut selbst wieder eine Relation (1:1 oder n:1) zu einer anderen Entity darstellt. Schon das im vorigen Abschnitt verwendete Beispiel wäre in der Praxis vermutlich in der Art aufgebaut, dass *SuperMarket* eine n:1-Beziehung zu einer Entity *Company* hätte, deren ID wiederum Teil des zusammengesetzten Schlüssels von *SuperMarket* wäre.

Eine solche Relation innerhalb der ID kann seit JPA 2.0 deklariert werden, wobei die konkrete Vorgehensweise von der Art der mehrteiligen ID abhängt.

Bei einer Composite ID, d. h. im Fall der Wiederholung der Attribute der ID-Klasse in der Entity-Klasse, wird das betroffene Attribut als *@OneToOne-* bzw. *@ManyToOne*-Relationsattribut deklariert. Die Namen der Attribute müssen wiederum übereinstimmen, während der Typ in der ID-Klasse der Typ des ID-Attributs der referenzierten Entity sein muss (Listing 3.105).

```
@Entity
@Access(AccessType.FIELD)
public class Department extends UuidEntity
{
  // erbt von UuidEntity eine String-ID
  …
}

@Entity
@Access(AccessType.FIELD)
@IdClass(ProjectId.class)
public class Project
{
  @Id @ManyToOne
  private Department department;

  @Id
  private String     prjId;
  …
}

@Embeddable
@Access(AccessType.FIELD)
public class ProjectId implements Serializable
{
  private String department;
  private String prjId;
  …
}
```

Listing 3.105: Relationsattribut als Teil einer Composite ID

Etwas unglücklich ist hier, dass aufgrund der Namensgleichheit nur eines der aufeinander bezogenen Attribute fachlich korrekt benannt werden kann: In der ID-Klasse des Beispiels etwa wäre statt des Feldnamens *department* wohl eher *departmentId* sinnvoll. Dieser kleine Nachteil lässt sich natürlich durch passend benannte Zugriffsmethoden ausgleichen.

Arbeitet man dagegen mit einer Embedded ID, wird ein zusätzliches Relationsattribut benötigt, das mit der Annotation *@MapsId* dem entsprechenden ID-Attribut zugeordnet wird (Listing 3.106).

```
@Entity
@Access(AccessType.FIELD)
public class Department extends UuidEntity
{
  // erbt von UuidEntity eine String-ID
  …
}

@Entity
@Access(AccessType.FIELD)
@IdClass(ProjectId.class)
public class Project
{
  @EmbeddedId
  private ProjectId  id;

  @ManyToOne @MapsId("departmentId")
  private Department department;
  …
  public Project_EmbeddedId(Department department, String prjId, …)
  {
    this.id = new ProjectId(department.getId(), prjId);
    this.department = department;
    …
  }
}

@Embeddable
@Access(AccessType.FIELD)
public class ProjectId implements Serializable
{
  private String departmentId;
  private String prjId;

  public ProjectId(String departmentId, String prjId)
  {
    this.department = departmentId;
    this.prjId = prjId;
  }
  …
}
```

Listing 3.106: Relationsattribut als Teil einer Embedded ID

Hier ist man in der Benennung der Attribute flexibler, dafür gibt es nun einen kleinen Fallstrick: Für die Speicherung des Fremdschlüsselwerts in die Datenbanktabelle ist das Relationsattribut zuständig. Im Beispiel muss also bspw. vor dem Aufruf von *persist* das Attribut *Project.department* besetzt sein. Während der Aktion des Entity Managers wird dann *Project.id.departmentId* passend gesetzt und in der Folge in der entsprechenden Spalte abgespeichert. Zwischen der Besetzung des Relationsattributs und dem Speichern des Objekts ist also der zugehörige Teil der ID nicht (richtig) gesetzt. Es empfiehlt sich daher, beide Attribute konsistent zu setzen. Die Beispielklasse tut dies in ihrem Konstruktor.

3.6.4 Locking

Enterprise-Anwendungen werden i. d. R. nicht von nur einem Benutzer verwendet. Vielmehr greifen ggf. mehrere Nutzer gleichzeitig auf die gleichen Daten zu. Dabei kann es zu Konflikten kommen, evtl. sogar zu inkonsistenten Daten, wenn zwei Änderungen zeitlich verzahnt durchgeführt werden. Betrachten wir einmal das folgende Szenario:

- User A liest Objekt x aus der DB
- User B liest Objekt x aus der DB
- User A ändert seine Kopie von x und speichert sie ab
- User B ändert seine Kopie von x und speichert sie ab

Die Änderungen von B sind am Ende dauerhaft gespeichert, wenn keine der im Folgenden beschriebenen Schutzmaßnahmen ergriffen wird. Problematisch ist nun nicht, dass B „gewinnt" (das sieht A vielleicht anders …), sondern dass er seine Änderungen zu einem Zeitpunkt abspeichert, zu dem die Voraussetzungen dafür ggf. gar nicht mehr vorliegen, ohne dass es dabei eine Warnung oder Fehlermeldung gibt.

Abhilfe schafft eine Blockierung der zu verarbeitenden Daten, wofür Java Persistence zwei Verfahren anbietet: Optimistic und Pessimistic Locking.

Das Setzen der Blockierung kann beim Lesen von Objekten geschehen. Dazu gibt es Varianten der Entity-Manager-Methoden *find* und *refresh*, die einen Parameter vom Typ *LockModeType* annehmen. Für eine *Query* kann vor ihrer Ausführung die Methode *setLockMode* aufgerufen werden (Listing 3.107).

```
EntityManager em = …
someEntity = em.find(…, LockModeType.PESSIMISTIC_WRITE);
em.refresh(…, LockModeType.NONE);

Query q = …
q.setLockMode(LockModeType.PESSIMISTIC_READ);
```

Listing 3.107: Anfordern eines Lock-Modus beim Lesen von Objekten

Für bereits gelesene Daten kann die Methode *lock* im *EntityManager* aufgerufen werden (Listing 3.108).

```
EntityManager em = …
someEntity = em.find(…);
em.lock(someEntity, LockModeType.PESSIMISTIC_WRITE);
```

Listing 3.108: Listing 3.108: Anfordern eines Lock-Modus für ein bereits gelesenes Objekt

Die folgenden *LockModeType*s stehen zur Verfügung:

- *LockModeType.NONE*: Keine Blockierung verwenden
- *LockModeType.OPTIMISTIC*: Optimistic Locking verwenden. Dies ist die Vorgabeeinstellung
- *LockModeType.OPTIMISTIC_FORCE_INCREMENT*: Wie *OPTIMISTIC*, allerdings wird das Versionsattribut (s. u.) inkrementiert bzw. aktualisiert, ohne dass zwingend eine Änderung der Daten vorgenommen wurde
- *LockModeType.PESSIMISTIC_READ*: Der Eintrag in der Tabelle wird mit einem Shared Lock versehen
- *LockModeType.PESSIMISTIC_WRITE*: Der Tabelleneintrag wird mit einem Exclusive Lock gesperrt
- *LockModeType.PESSIMISTIC_FORCE_INCREMENT*: Wie *PESSIMISTIC_WRITE*, allerdings mit gleichzeitiger Erhöhung bzw. Aktualisierung des Versionsattributs (s. u.)

Aus Kompatibilitätsgründen existieren zwei weitere Konstanten in *LockModeType*, die in neuen Anwendungen nicht mehr verwendet werden sollen:

- *LockModeType.READ* entspricht *OPTIMISTIC*
- *LockModeType.WRITE* entspricht *OPTIMISTIC_FORCE_INCREMENT*

Mit Ausnahme von *NONE* dürfen diese Modi nur innerhalb einer Transaktion aktiviert werden. Die Blockierung bleibt dann bis zum Transaktionsende bestehen.

Optimistic Locking

Beim Optimistic Locking wird keine echte Blockierung des Eintrags in der Datenbank genutzt, sondern stattdessen vor dem Abspeichern geprüft, ob die Daten in der Tabelle noch den Zustand haben, den sie beim Lesen zuvor hatten. Das wird in den allermeisten Fällen so sein, wodurch sich der Name des Verfahrens erklärt.

Der Vergleich der Daten geschieht allerdings nicht komplett. Vielmehr benötigt JPA dazu ein Versionsattribut, das i. A. zusätzlich zu den fachlichen Attributen in die Entity-Klasse und damit auch als Spalte in die Tabelle aufgenommen werden muss. Das Versionsattribut muss den Typ *short*, *Short*, *int*, *Integer*, *long*, *Long* oder *java.sql.Timestamp* haben und die Annotation *@Version* tragen (Listing 3.109).

```
@Entity
@Access(AccessType.FIELD)
public class Xyz
{
  …
  @Version
  private long version;
  …
}
```

Listing 3.109: Versionsattribut

Vor jedem Speichern wird nun automatisch geprüft, ob der Wert des Versionsattributs in der Datenbank noch mit dem im zuvor gelesenen Objekt übereinstimmt. Wenn nicht, wird die Operation mit Auswurf einer *OptimisticLockException*[12] abgebrochen. Andernfalls wird die Speicherung durchgeführt und dabei das Versionsattribut inkrementiert bzw. auf die aktuelle Zeit gesetzt. Im oben skizzierten Szenario würde B beim Speichern seiner Änderungen eine Fehlermeldung erhalten und der von A abgelegte Zustand weiter gelten.

Es ist sicher verlockend, ein Versionsattribut vom Typ *Timestamp* einzusetzen, da man damit ohne weiteren Aufwand einen Änderungszeitstempel in der DB abgelegt bekommt. Hier ist aber Vorsicht angesagt, da das Verfahren nur mit einer Datenbank funktionieren kann, die Timestamps auf Millisekunden genau abspeichert. Sicherer ist die Verwendung eines ganzzahligen Versionsattributs.

In einer Entity-Klasse darf es nur ein Versionsattribut geben. Bei der Nutzung von Zusatztabellen muss das Versionsattribut in der Haupttabelle platziert werden.

Optimistic Locking wird als vorgegebenes Verfahren verwendet, sobald ein Versionsattribut im bearbeiteten Entity-Objekt existiert. Die explizite Wahl ist mit den beiden Lock-Modi *OPTIMISTIC* und *OPTIMISTIC_FORCE_INCREMENT* möglich.

Die Spezifikation verlangt nicht, dass der Provider Optimistic Locking tatsächlich ohne Datenbanksperren implementiert, die bekannten Provider tun dies aber. Für Details sei auf die Spezifikation verwiesen („Locking and Concurrency“).

Pessimistic Locking

In manchen Fällen ist Optimistic Locking nicht ausreichend, z. B. wenn schon bei Beginn eines Geschäftsprozesses – wenn die verwendeten Daten gelesen werden –, möglichst sicher sein soll, dass die veränderten Daten später auch wieder gespeichert werden können. Dann kommt man um Pessimistic Locking nicht herum. Bei diesem Verfahren werden die betroffenen Sätze in der Datenbank mit einer Blockierung belegt, die parallele Zugriffe verhindert.

12 *javax.persistence.OptimisticLockException*

Pessimistic Locking wird durch die Lock-Modi *PESSIMISTIC_...* angewählt. Dabei dürfen ein Exclusive Lock (*PESSIMISTIC_WRITE*) für einen Eintrag existieren oder alternativ beliebig viele Shared Locks (*PESSIMISTIC_READ*). Das genaue Verfahren ist durch die Spezifikation nicht festgelegt. Insbesondere kann die Datenbank ggf. mehr Zeilen als nötig sperren.

Wird ein mit *PESSIMISTIC_READ* gesperrtes Objekt im Verlaufe des Geschäftsprozesses verändert und abgespeichert, wird der Shared Lock in diesem Moment zu einem Exclusive Lock verändert.

Der ein Lock anfordernde Prozess wartet i. d. R., bis der Eintrag gesperrt werden kann. Sollte dies aus einem schwerwiegenden Grund (bspw. ein Dead Lock) nicht möglich sein, wird der entsprechende Methodenaufruf abgebrochen und eine *PessimisticLockException*[13] ausgeworfen. Die aktuelle Transaktion wird zudem für ein Rollback markiert. Wird die Lock-Anforderung dagegen durch eine Zeitüberschreitung abgebrochen, so wird eine *LockTimeoutException*[14] ausgeworfen, ohne die Transaktion ungültig zu machen. Die entsprechende Wartezeit kann durch eine Konfigurationseinstellung der DB vorgegeben sein oder durch den unten beschriebenen Hint *javax.persistence.lock.timeout*.

Für Pessimistic Locking ist es nicht nötig, dass ein Versionsattribut existiert. Ist das jedoch der Fall, wird es wie beim Optimistic Locking beschrieben behandelt. Dadurch ist gewährleistet, dass zwei Prozesse mit unterschiedlichen Locking-Verfahren auf den gleichen Daten arbeiten können.

Locking Hints

Das Verhalten von Pessimistic Locking kann mithilfe zweier Hints beeinflusst werden: Mit *javax.persistence.lock.timeout* kann eine maximale Wartezeit in Millisekunden angegeben werden. Ohne Angabe des Hints gilt eine evtl. für die DB konfigurierte Timeoutzeit als Vorgabe.

Mit dem Hint *javax.persistence.lock.scope* kann bestimmt werden, ob ein Pessimistic Lock mehr als nur das jeweils bearbeitete Objekt blockiert. Die folgenden Werte können angegeben werden:

- *PessimisticLockScope.NORMAL*: Nur die zum Objekt gehörenden Tabelleneinträge werden blockiert. Dies ist die Voreinstellung.
- *PessimisticLockScope.EXTENDED*: Es werden zusätzlich die Einträge der Tabellen blockiert, die Element Collections darstellen. Zudem erstreckt sich die Blockierung auch auf die Einträge in Verknüpfungstabellen, die zu Relationen gehören, deren Eigentümer das bearbeitete Objekt ist. Die dadurch referenzierten Einträge werden allerdings nicht blockiert.

Die Hints können als Parameter den Methoden *EntityManager.find*, *EntityManager.lock*, *EntityManager.refresh* und *Query.setHint* übergeben oder bei der Deklaration einer Named

13 *javax.persistence.PessimisticLockException*
14 *javax.persistence.LockTimeoutException*

Query eingetragen werden. *javax.persistence.lock.timeout* kann zudem als Property im Deskriptor *persistence.xml* eingetragen werden. Es sei an dieser Stelle nochmals darauf hingewiesen, dass Hints nur Hinweise sind, die vom Provider nicht beachtet werden müssen. Für eine portable Anwendung sollten Sie auf die Hints also nicht angwiesen sein.

3.6.5 Callback-Methoden und Listener

Mithilfe von Callback-Methoden (oder auch Lifecycle-Methoden) können in die Operationen des Persistenzproviders bzgl. eines Entity-Objekts zusätzliche Aktionen eingeschleust werden. So kann man bspw. vor dem Speichern der Daten Validitätstests durchführen, um zu verhindern, dass ungültige Daten in der Tabelle abgelegt werden. Dazu können Methoden mit der Signatur *void methodName()* in die Entity-Klasse aufgenommen und mit einer der Anotationen aus Tabelle 3.1 markiert werden.

Callback-Annotation	Annotierte Methode wird aufgerufen …
@PrePersist	in *EntityManager.persist* vor der Ablage in der DB. Dies gilt auch bei *merge*, wenn das betroffene Objekt neu in die DB eingefügt wird.
@PostPersist	nach dem Einfügen in der DB. Dies kann direkt nach dem Aufruf von *persist* geschehen, ist aber normalerweise verzögert. Generierte IDs sind in der Callback-Methode bereits gesetzt.
@PostLoad	nach dem Laden eines Objekts aus der DB im Zuge eines Aufrufs von *find* oder der Ausführung einer Query.
@PreUpdate	vor dem Speichern von Änderungen in der DB. Dies kann bspw. durch Aufruf von *flush* ausgelöst werden oder bis zum Ende der Transaktion verzögert geschehen.
@PostUpdate	analog zu *@PreUpdate*, allerdings nach der DB-Operation.
@PreRemove	in *EntityManager.remove* vor dem Löschen aus der DB.
@PostRemove	nach dem Löschen aus der DB. Dies kann direkt nach dem Aufruf von *remove* geschehen, ist aber normalerweise verzögert.

Tabelle 3.1: Callback-Annotationen

Die Methoden können eine beliebige Sichtbarkeit (*private*, …, *public*) haben, dürfen aber nicht *static* oder *final* sein.

Die Deklaration von Exceptions ist nicht erlaubt, daher können die Methoden nur Unchecked Exceptions auswerfen. In den Pre-Methoden wird dadurch die entsprechende DB-Aktion nicht durchgeführt. Geschah der Aufruf innerhalb einer Transaktion, wird diese als „Rollback Only" markiert.

Ein Beispiel für die oben angesprochene Validierung der Daten zeigt Listing 3.110[15].

15 Dazu werden wir später allerdings eine elegantere Möglichkeit kennen lernen: Bean Validation.

```
@Entity
@Access(AccessType.FIELD)
public class Country
{
  …
  @PrePersist
  @PreUpdate
  private void validate()
  {
    if (this.name == null || this.name.isEmpty())
    {
      throw new IllegalArgumentException("name darf nicht leer sein");
    }
    …
  }
```

Listing 3.110: Validierung vor dem Speichern

Die Callback-Methoden können auch in eine separate Listener-Klasse ausgelagert werden. Die Signatur der Methoden ändert sich dann in *void methodName(Object)*, da in diesem Fall das gerade in der Verarbeitung befindliche Entity-Objekt an die Methode übergeben wird. Einer Entity-Klasse können mithilfe der Annotation *@EntityListeners* die gewünschten Listener-Klassen mitgegeben werden (Listing 3.111).

```
public class DebugListener
{
  @PrePersist
  public void prePersist(Object entity)
  {
    System.out.println("prePersist(" + entity + ")");
  }

  @PostPersist
  public void postPersist(Object entity)
  {
    System.out.println("postPersist(" + entity + ")");
  }
  …
}

@Entity
@Access(AccessType.FIELD)
@EntityListeners(DebugListener.class)
public class Country
{
  …
}
```

Listing 3.111: Entity Listener und seine Verknüpfung mit einer Entity-Klasse

Entity Listener können sogar als Default Listener eingetragen werden. Sie gelten dann für alle Entity-Klassen der Persistence Unit. Zur Deklaration der Default Listener dient der in Listing 3.112 gezeigte Abschnitt des Mapping-Deskriptors *META-INF/orm.xml*. Für weitere Details dazu und zur Aufrufreihenfolge im Falle von mehrfach deklarierten Callbacks sei auf die Spezifikation verwiesen („Entity Listeners and Callback Methods“).

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
                      http://java.sun.com/xml/ns/persistence/orm_2_0.xsd"
  version="2.0">
  <persistence-unit-metadata>
    <persistence-unit-defaults>
      <entity-listeners>
        <entity-listener class="de.….DebugListener"/>
      </entity-listeners>
    </persistence-unit-defaults>
  </persistence-unit-metadata>
  …
</entity-mappings>
```

Listing 3.112: Eintrag eines Default Entity Listeners im Mapping-Deskriptor

Seit JPA 2.1 können innerhalb von Entity Listenern auch CDI-Injektionen verwendet werden.

3.6.6 Bulk Update/Delete

Neben Suchabfragen können mit JPQL- und SQL-Queries auch Veränderungen der Daten in der Datenbank durchgeführt werden. Statt *select* wird dazu im JPQL- bzw. SQL-Kommando *delete* oder *update* verwendet. Im zweiten Fall wird in einer *Set*-Klausel nach der *From*-Klausel angegeben, welche Modifikationen an den von der Query betroffenen Daten durchgeführt werden. Die Ausführung der Operation geschieht mit der Methode *executeUpdate*. Sie gibt die Anzahl betroffener Datenbankeinträge zurück. (Listing 3.113).

```
Query query = em.createQuery("update Cocktail c set c.name=… where …");
int count = query.executeUpdate();

query = em.createQuery("delete Cocktail c where …");
count = query.executeUpdate();
```

Listing 3.113: Bulk Update und Bulk Delete

Seit JPA 2.1 lassen sich mit dem Criteria API ebenfalls Bulk-Operationen durchführen. Dazu wurden die beiden Interfaces *CriteriaUpdate<T>* und *CriteriaDelete<T>* sowie passende Factory-Methoden in *CriteriaBuilder* ergänzt. Die Vorgehensweise ist analog zur

Ausführung von Selektionsabfragen: Aufbau eines Objekts mit dem Criteria API, Erzeugung einer normalen Query daraus und Ausführung mittels *executeUpdate* (Listing 3.114).

```
CriteriaBuilder cBuilder = em.getCriteriaBuilder();
CriteriaUpdate<Cocktail> cUpdate
  = cBuilder.createCriteriaUpdate(Cocktail.class);
Root<Cocktail> c = cUpdate.from(Cocktail.class);
Path<String> cName = c.get(Cocktail_.name);
cUpdate.set(cName, cBuilder.concat(cName, " (a)"));
cUpdate.where(cBuilder.equal(cName, "Dummy"));
Query query = em.createQuery(criteriaUpdate);
int count = query.executeUpdate();
```

Listing 3.114: Bulk Update mittels Criteria API

Zu beachten ist, dass die Bulk-Operationen den Entity Manager zwar benutzen, ihn aber dennoch weitestgehend umgehen:

- Derzeit gemanagte Objekte werden nicht verändert. Dadurch kann der Zustand der Objekte im Entity Manager nach der Operation ggf. vom Zustand in der DB abweichen.
- Ein Optimistic Locking findet nicht statt. Evtl. vorhandene Versionswerte bleiben also unverändert.
- Eine Kaskadierung der Operation findet nicht statt.

Die Bulk-Operationen benötigen zu ihrer Ausführung eine aktive Transaktion.

3.7 Caching

Datenbankoperationen sind im Vergleich zu Aktionen innerhalb des Java-Prozesses schleichend langsam. Daher versucht man, einmal gelesene Daten für eine zweite Verwendung möglichst nicht erneut aus der Datenbank zu holen, sondern sie stattdessen im Hauptspeicher zu halten. Da die Daten für eine konsistente Verarbeitung aktuell sein müssen, funktioniert Caching besonders gut für Daten, die i. W. konstant sind – die klassischen Konfigurations- oder Stammdaten –, während es für sich häufig ändernde Daten nutzlos oder sogar kontraproduktiv sein kann.

Java Persistence definiert im Standard zwei Caches: 1st und 2nd Level Cache. Die gängigen Provider unterstützen darüber hinaus noch den sog. Query Cache.

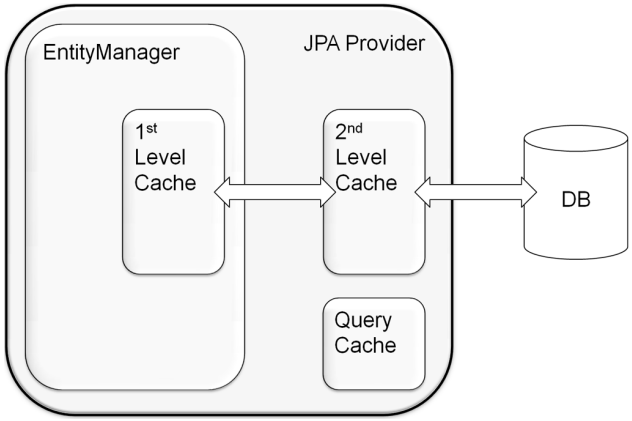

Abbildung 3.28: Caches

First Level Cache

Der Entity Manager hält alle von ihm gemanagten Objekte in einer internen *Map*-ähnlichen Datenstruktur, die mit der ID der darin befindlichen Objekte indiziert wird. Dieser 1st Level Cache ist stets vorhanden und lässt sich nicht deaktivieren. Er liegt sozusagen als Puffer zwischen der Java-Anwendung und der Datenbank. Die Entity-Manager-Methoden wirken i. W. auf den Inhalt des Caches, die zugehörige Datenbankoperation ist damit nur lose gekoppelt. So führt *find* nur beim ersten Aufruf zum Laden des Objekts aus der Datenbank. Alle weiteren *find*-Aufrufe für die gleiche ID liefern das Objekt aus dem Cache. Umgekehrt fügt *persist* ein neues Objekt zunächst in den Cache ein, der DB-Eintrag passiert erst später.

Queries werden nicht aus dem 1st Level Cache bedient. Vielmehr führt eine Query immer zu einer Datenbankabfrage, wobei geänderte Objekte i. A. zuvor gespeichert werden.

Der 1st Level Cache ist Teil des Entity Managers. Wird dieser geschlossen, geht auch der Cache verloren. Löst man ein bislang gemanagtes Objekt aus dem Entity Manager – per *detach*, *clear* oder in Folge eines Transaktions-Rollbacks – wird der entsprechende Cache-Eintrag ebenfalls entfernt. Der 1st Level Cache ist also eher kurzfristig aktiv, dient somit i. W. zur Unterstützung von Geschäftsprozessen, die die betroffenen Daten in ihrem Verlauf einlesen, bearbeiten, die Veränderungen am Transaktionsende speichern und den Entity Manager anschließend schließen.

Second Level Cache

Sollen die verarbeiteten Daten über eine längere Zeit im Cache gehalten werden, wird dazu ein Cache benötigt, der übergreifend über Geschäftsprozesse oder Transaktionen wirksam bleibt. Diese Aufgabe übernimmt der 2nd Level Cache. Er ist ähnlich dem 1st Level Cache eine Datenstruktur, deren Einträge mit ihrer ID adressiert werden, die aber nicht an einen Entity Manager oder eine Transaktion gebunden ist, sondern applikationsweit zur Verfügung steht.

Der 2nd Level Cache wird durch das Element <*shared-cache-mode*> im Deskriptor *persistence.xml* konfiguriert (Listing 3.115). Der Wert des Elements bestimmt, welche Entity-Klassen im 2nd Level Cache Berücksichtigung finden (Tabelle 3.2).

```
<persistence-unit name="…">
  <provider>…</provider>
  <shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>
  …
```

Listing 3.115: Konfiguration des 2nd Level Cache im Deskriptor „persistence.xml"

Wert von shared-cache-mode	2nd Level Cache aktiv für
ALL	alle Entity-Klassen
NONE	keine Klasse
ENABLE_SELECTIVE	Entities mit *@Cacheable(true)*
DISABLE_SELECTIVE	Entities ohne *@Cacheable(false)*

Tabelle 3.2: Verfügbare Cache-Modi

Es empfiehlt sich, Caching nicht „mit der Gießkanne" zu betreiben, d. h. den 2nd Level Cache nicht unreflektiert für alle Entity-Klassen einzuschalten, da die Performanz der Anwendung nur bei sich nicht oder nicht häufig ändernden Daten vom Caching profitiert. In anderen Fällen kann es sogar nachteilig sein. In den meisten Fällen ist daher *ENABLE_SELECTIVE* der richtige Modus. Bei ihm nehmen nur die Entity-Klassen am 2nd-Level-Caching teil, die mit *@Cacheable(true)* annotiert sind (Listing 3.116).

```
@Entity
@Access(AccessType.FIELD)
@Cacheable(true)
public class Country
{
  …
```

Listing 3.116: Explizite Aktivierung des 2nd Level Cache für eine Klasse

In gleicher Weise, nur invertiert, arbeitet *DISABLE_SELECTIVE*. Wird <*shared-cache-mode*> nicht angegeben, gilt ein providerabhängiger Vorgabewert.

Je nach eingesetztem Provider sind zur Konfiguration des Caches noch weitere Parameter notwendig, die im <*properties*>-Anteil der Datei *persistence.xml* angegeben werden. Damit lassen sind bspw. Cache-Strategien, Verweilzeiten, Replikationsverfahren im Cluster usw. konfigurieren. In einigen Fällen stehen sogar mehrere Caching-Provider zur Auswahl. Für Details dazu sei auf die Dokumentation des jeweiligen Providers verwiesen. Das Beispielprojekt enthält eine Basiskonfiguration für EclipseLink mit dem standardmäßig eingebauten Cache-Provider sowie für Hibernate mit Infinispan als Cache-Provider.

Auf den 2nd Level Cache kann im Programm auch direkt zugegriffen werden. Dazu wird ein Objekt vom Typ *Cache* benötigt, das man sich z. B. von der Entity Manager Factory liefern lassen kann. Das Interface Cache enthält die Methode *contains*, mit der man abfragen kann, ob ein Objekt mit einer bestimmten ID im Cache vorhanden ist, sowie die Methoden *evict* und *evictAll*, mit denen Objekte aus dem Cache entfernt werden können (Listing 3.117).

```
EntityManager em = …
EntityManagerFactory emf = em.getEntityManagerFactory();
Cache secondLevelCache = emf.getCache();

// Ist Country "IT" im Cache?
boolean isCached = secondLevelCache.contains(Country.class, "IT"));

// Country "DE" aus dem Cache entfernen
secondLevelCache.evict(Country.class, "DE"));
```

Listing 3.117: Programmatischer Zugriff auf den 2nd Level Cache

Die Nutzung des Cache-API wird sicher nur in speziellen Situationen oder zum Debugging benötigt, da der Cache eigentlich transparent für die Anwendung funktioniert. Aber Sie wissen ja bereits: „Eigentlich" ist ein Signalwort …

Query Cache

Queries führen jedes Mal zu einer Abfrage in der Datenbank, d. h. die zuvor beschriebenen Caches kommen dazu nicht zum Einsatz. Es gibt aber die Möglichkeit, auch dafür einen Cache einzurichten, der zu einer Query und den darin genutzten Parametern die Ergebnismenge speichert. Der Query Cache ist providerabhängig verfügbar und in aller Regel mit dem 2nd Level Cache verbunden. Er speichert die Ergebnismenge daher meist nur in Form der IDs; die zugehörigen Objekte befinden sich dann im 2nd Level Cache.

Der Query Cache ist in der Spezifikation nur rudimentär erwähnt. Seine Konfiguration ist vom eingesetzten Provider abhängig. Details finden sich in der Dokumentation der Provider. Zudem muss die Nutzung des Caches für eine Query explizit aktiviert werden, und zwar mithilfe eines providerabhängigen Hints (Listing 3.118).

```
Continent continent = …;
TypedQuery<Kunde> query
  = em.createQuery("select c from Country c where c.continent=:cont",
                   Country.class);
query.setParameter("cont", continent);
query.setHint("eclipselink.cache-usage", "CheckCacheThenDatabase");
query.setHint("org.hibernate.cacheable", true);
…
```

Listing 3.118: Aktivierung des Query Cache für EclipseLink oder Hibernate

Einträge im Query Cache werden verworfen, wenn Änderungen an den beteiligten Entities vorgenommen werden. Das Verfahren ist hier aber wiederum vom Provider abhängig.

3.8 Erweiterte Entity Manager

Bislang wurde in diesem Kapitel transaktionsgebundene Entity Manager verwendet, die sich auf einfache Weise bereitstellen lassen, z. B. per Injektion in eine CDI Bean (s. Abschnitt 3.2.3, Listing 3.7). Damit lassen sich Abläufe modellieren, die dem folgenden Schema folgen:

a) Beginn der Transaktion

b) Lesen der benötigten Daten

c) Erzeugen, Modifizieren und Löschen von Daten

d) Transaktion abschließen (und damit Änderungen speichern)

Ein gesamter Geschäftsprozess setzt sich i. d. R. aus mehreren Abläufen dieser Art zusammen, die bspw. durch Userinteraktionen voneinander getrennt sind.

Nach dem Abschluss einer Transaktion werden die verarbeiteten Objekte detacht, müssen also mittels *EntityManager.merge* wieder attacht werden, wenn sie in einem Folgeablauf wieder benötigt werden. Damit sind i. A. erneute Zugriffe auf die DB verbunden.

Zudem können im *Detached*-Zustand keine Lazy-Attribute nachgelesen werden. Soll z. B. eine Webanwendung Daten zur Anzeige bringen, die auf die beschriebene Art eingelesen wurden, so müssen vor Ende der Transaktion alle notwendigen Attribute bereits gelesen worden sein, damit die Anwendung nicht mit einer Lazy-Load-Exception abbricht.

3.8.1 Extended Entity Manager

Eine alternative Verarbeitungsmöglichkeit eröffnet sich mit einem erweiterten Entity Manager. Er ist nicht an die Transaktion gebunden, wodurch die Entity-Objekte über Transaktionsgrenzen hinweg gemanagt bleiben können.

Die EJB-Spezifikation sieht eine Möglichkeit zur Erzeugung eines Extended Entity Managers vor, und zwar wiederum durch Injektion mit der Annotation *@PersistenceContext*, nun aber mit einem Zusatzparameter zur Auswahl des erweiterten Typs.

```
@Stateful
public class SomeBusinessProcessBean
{
  @PersistenceContext(name = "ee_demos",
                      type=PersistenceContextType.EXTENDED)
  private EntityManager entityManager;
  …
```

Listing 3.119: Injektion eines Extended Entity Manager in eine Stateful EJB

Ein Extended Entity Manager ist an die Session gebunden, d. h. er „lebt“ mit der jeweiligen Instanz der Stateful EJB, in die er injiziert wurde. Die Methoden *persist*, *merge* und

remove, deren Aufruf bisher nur in einer aktiven Transaktion erlaubt war, dürfen nun auch ohne Transaktion verwendet werden. Wird später eine Transaktion begonnen und mit *Commit* beendet, werden die Änderungen abgespeichert. Damit wird es möglich, in mehreren Teilabläufen eines Geschäftsprozesses Daten zu lesen und zu modifizieren, ohne dabei Transaktionen zu nutzen. Die Daten bleiben über die ganze Zeit im Entity Manager gemanagt, d. h. unnötige DB-Zugriffe unterbleiben und Lazy-Attribute können jederzeit nachgelesen werden. Sollen die Daten schließlich gespeichert werden, reicht der Aufruf einer – ggf. sogar leeren – transaktionalen Methode (Listing 3.120).

```
@Stateful
@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
public class SomeBusinessProcessBean
{
  @PersistenceContext(name = "ee_demos",
                      type=PersistenceContextType.EXTENDED)
  private EntityManager entityManager;

  public void doStep1()
  {
    // Geschäftsprozess, Teil 1: Daten lesen, ggf. verändern
  }

  public void doStep2()
  {
    // Geschäftsprozess, Teil 2: Weiter lesen und verändern
  }

  @TransactionAttribute(TransactionAttributeType.REQUIRED)
  public void save()
  {
  }
}
```

Listing 3.120: Realisierung eines Geschäftsprozesses durch Methoden einer Stateful EJB

Dieses Pattern funktioniert recht gut, wenn die steuernde Instanz der EJB am Ende des Geschäftsprozesses zerstört wird, womit dann auch der Entity Manager abgebaut wird. Somit verbleiben keine „Leichen" im Speicher, das Big-Session-Problem wird also vermieden. Das Pattern ist allerdings darauf beschränkt, dass eine Stateful Session EJB die Teile des Geschäftsprozesses verkörpert.

3.8.2 Application Managed Entity Manager

Will man auf EJBs verzichten – zumindest für die Bereitstellung eines Entity Managers – kann man einen Application Managed Entity Manager verwenden. Hierbei wird ein Objekt vom Typ *EntityManager* in eigener Regie erzeugt und kontrolliert. Die Lebensdauer dieses Entity Managers kann im einfachsten Fall dem Request entsprechen, um so auch in

der Anzeigephase der Anwendung noch Lazy-Attribute nachladen zu können. Es ist auch denkbar, die Lebensdauer dem modellierten Geschäftsprozess anzugleichen, um wiederum dem Big-Session-Problem aus dem Weg zu gehen. Dann nutzt man den CDI Scope *Conversation* und hält eine Konversation genau über die Dauer des Geschäftsprozesses aktiv. Zur Bereitstellung des Entity Managers lässt sich sehr gut ein CDI Producer verwenden. Er liefert auf Anforderung eine neue Instanz vom Typ *EntityManager* und sieht auch eine passende Disposer-Methode zum Schließen des Entity Managers vor (Listing 3.121).

```
@ApplicationScoped
public class EntityManagerProducer implements Serializable
{
  @PersistenceUnit(unitName = "default")
  private EntityManagerFactory entityManagerFactory;

  @Produces
  @RequestScoped // oder @ConversationScoped
  public EntityManager createEntityManager()
  {
    return this.entityManagerFactory.createEntityManager();
  }

  public void disposeEntityManager(@Disposes EntityManager entityManager)
  {
    if (entityManager.isOpen())
    {
      entityManager.close();
    }
  }
}
```

Listing 3.121: CDI Producer für einen Application Managed Entity Manager

Die *EntityManager*-Instanz wird im gezeigten Producer hergestellt, indem auf ein Objekt des Typs *EntityManagerFactory* zurückgegriffen wird, das quasi die Persistence Unit darstellt. Im nächsten Abschnitt wird die Entity Manager Factory nochmals etwas näher beleuchtet. Im Kontext eines Applikationsservers kann sie mithilfe der Annotation *@PersistenceUnit* injiziert werden, wobei der Namensparameter entfallen kann, wenn *persistence.xml* nur eine Persistence Unit definiert.

Die Producer-Methode *createEntityManager* ist mit der gewünschten Scope-Annotation versehen, wodurch der gelieferte Entity Manager im entsprechenden Kontext verwaltet wird. Am Ende der Lebensdauer sorgt die Dispose-Methode *disposeEntityManager* dafür, dass der Entity Manager geschlossen wird.

Auch mit einem Application Managed Entity Manager können Datenänderungen außerhalb von Transaktionen gesammelt werden, um dann in einer transaktionalen Methode gespeichert zu werden. Anders als bei einem Extended Entity Manager gibt es hier allerdings keine automatische Zuordnung zur aktiven Transaktion; diese muss explizit mit

der Methode *joinTransaction* hergestellt werden. Es empfiehlt sich, dann ebenfalls *flush* aufzurufen. Das sollte zwar lt. Spezifikation am Transaktionsende ohnehin geschehen, unterbleibt aber erfahrungsgemäß bei einigen Implementierungen.

Die Anwendungsaspekte „Geschäftsprozess“ und „Persistenzschicht“ lassen sich in der Praxis gut trennen, um Wiederverwendbarkeit und Übersichtlichkeit zu erhöhen. Dem kommt entgegen, dass CDI Scopes sich problemlos mischen lassen. Man kann also wie im Beispiel einen Conversation Scoped Entity Manager in eine Request Scoped Bean injizieren, die wiederum von einer Conversation Scoped Bean verwendet wird (Listing 3.122).

```
@ConversationScoped
public class XyzController implements Serializable
{
  @Inject
  XyzRepository   xyzRepository;

  @Inject
  Conversation    conversation;

  public void doFirstStep()
  {
    // Geschäftsprozess, Teil 1: Daten lesen, ggf. verändern
    conversation.begin();
    … = xyzRepository.findById(…);
    …
  }

  public void doSecondStep()
  {
    // Geschäftsprozess, Teil 2: Weiter lesen und verändern
    xyzRepository.insert(…);
    …
  }

  public void doLastStep()
  {
    // Geschäftsprozess, letzter Teil: Speichern
    xyzRepository.saveAll();
    conversation.end();
  }
}

@RequestScoped
public class XyzRepository implements Serializable
{
  @Inject
  private EntityManager entityManager;

  @TransactionRequired
  public void saveAll()
```

```
  {
    entityManager.joinTransaction();
    entityManager.flush;
  }

  public Xyz findById(int id)
  {
    return entityManager.find(Xyz.class, id);
  }

  public void insert(Xyz xyz)
  {
    entityManager.persist(xyz);
  }

  // weitere, nicht-transaktionale Methoden
  …
}
```

Listing 3.122: Nutzung eines konversationsbezogenen Entity Managers

Es gibt mit der dargestellten Vorgehensweise, einen Entity Manager im Conversation Scope zu nutzen, eine Schwierigkeit, die Ihnen nicht verheimlicht werden soll: Objekte im Conversation Scope müssen serialisierbar sein. Da die JPA-Spezifikation darüber keinerlei Aussage macht, ist es providerabhängig, ob der konkrete Entity Manager tatsächlich *Serializable* implementiert. Bei Hibernate ist das bspw. der Fall, während EclipseLink einen nichtserialisierbaren Entity Manager hat. Damit bleibt derzeit nur die Möglichkeit, bei Bedarf ein serialisierbares Proxy als Hülle um den Entity Manager zu konstruieren (Listing 3.123).

```
@ApplicationScoped
public class EntityManagerProducer implements Serializable
{
  …
  @Produces @ConversationScoped
  public EntityManager createEntityManager()
  {
    EntityManager entityManager
      = this.entityManagerFactory.createEntityManager();
    if (!(entityManager instanceof Serializable))
    {
      ClassLoader classLoader = EntityManager.class.getClassLoader();
      Class<?>[] interfaces = new Class[] { EntityManager.class,
                                            Serializable.class };
      EntityManagerInvocationHandler handler
        = new EntityManagerInvocationHandler(entityManager);
      entityManager
        = (EntityManager) Proxy.newProxyInstance(classLoader,
                                                 interfaces, handler);
    }
```

```
    return entityManager;
  }
…
  private static class EntityManagerInvocationHandler
    implements InvocationHandler
  {
    private transient EntityManager entityManager;

    public EntityManagerInvocationHandler(EntityManager entityManager)
    {
      this.entityManager = entityManager;
    }

    public Object invoke(Object proxy, Method method, Object[] args)
      throws Throwable
    {
      if (this.entityManager == null)
      {
        throw new IllegalStateException("Session discarded");
      }

      return method.invoke(this.entityManager, args);
    }
  }
}
```

Listing 3.123: Workaround für nichtserialisierbare Entity Manager

Dieser Workaround ist allerdings nicht ganz wasserdicht. Da man selbst in Java nicht wirklich zaubern kann, kann aus einem nichtserialisierbaren Entity Manager nicht ein serialisierbares Objekt gemacht werden. Stattdessen hält das Proxy den *EntityManager* als transientes Objekt, das demzufolge bei einer Serialisierung/Deserialisierung des Proxy-Objekts verloren gehen würde. Wir vertrauen also darauf, dass das nicht passieren wird. Wenn doch, wird von der betroffenen Methode eine sinnvolle Exception ausgeworfen.

3.9 Java Persistence in SE-Anwendungen

Wir haben uns in diesem Kapitel bislang mit Java Persistence im Enterprise-Kontext beschäftigt, d. h. mit der Nutzung innerhalb einer auf einem Applikationsserver laufenden Java-EE-Anwendung. Man kann JPA aber auch in SE-Anwendungen verwenden, sei es für Unit Tests oder für Standalone-Anwendungen. Dabei ändern sich gegenüber einer EE-Umgebung nur die drei im Folgenden besprochenen Punkte.

3.9.1 Konfiguration der Persistence Unit im SE-Umfeld

Auch außerhalb eines Applikationsservers wird die Deskriptordatei *META-INF/persistence.xml* benötigt, allerdings mit etwas verändertem Inhalt: Das Element <*jta-data-source*> entfällt. Stattdessen werden mit den Properties *javax.persistence.jdbc.driver*, *javax.persistence.jdbc.url*, *javax.persistence.jdbc.user* und *javax.persistence.jdbc.password* die von JDBC bekannten Parameter zum Aufbau einer DB-Verbindung angegeben.

Eine Unschärfe in der JPA-Spezifikation führt zu einer weiteren Änderung des Deskriptors: Im Abschnitt „Annotated Classes in the Root of the Persistence Unit" heißt es zwar „All classes contained in the root of the persistence unit are searched for annotated managed persistence classes …", aber später in „List of Managed Classes" findet sich der Satz „The class element is used to list a managed persistence class. A list of all named managed persistence classes must be specified in Java SE environments to insure portability." Im allgemeinen Fall müssen also die von der Anwendung verwendeten persistenten Klassen im Deskriptor einzeln aufgeführt werden, wenn auch EclipseLink und Hibernate eigentlich ohne diese Information auskommen.

Das Beispielprojekt nutzt JPA in der Standalone-Variante für Tests. Einen Ausschnitt der dazu im Test-Classpath liegenden *META-INF/persistence.xml* zeigt Listing 3.124. Darin sind mehrere Persistence Units spezifiziert, um unterschiedliche Provider zum Test verwenden zu können.

```
<persistence xmlns="http://java.sun.com/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_0.xsd"
  version="2.0">
  <persistence-unit name="eclipselink">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>

    <class>de.gedoplan.buch.eedemos.entity.Car</class>
    <class>de.gedoplan.buch.eedemos.entity.City</class>
    …
    <class>de.gedoplan.buch.eedemos.entity.Vehicle</class>

    <shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>

    <properties>
      <property name="javax.persistence.jdbc.driver"
                value="com.mysql.jdbc.Driver" />
      <property name="javax.persistence.jdbc.url"
                value="jdbc:mysql://localhost:3306/ee_demos" />
      <property name="javax.persistence.jdbc.user"
                value="ee_demos" />
      <property name="javax.persistence.jdbc.password"
                value="ee_demos" />
  …
```

```
  </persistence-unit>

  <persistence-unit name="hibernate">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
  …
  </persistence-unit>
```

Listing 3.124: Deskriptor „META-INF/persistence.xml“ für eine SE-Anwendung

3.9.2 Erzeugung eines Entity Managers in SE-Anwendungen

Außerhalb von Applikationsservern stehen uns die Enterprise-Ressourcen nicht zur Injektion zur Verfügung. An die Stelle der bisher genutzten Annotationen *@PersistenceContext* bzw. *@PersistenceUnit* tritt nun eine Anweisungssequenz, die ausgehend von der Klasse *Persistence* zunächst eine Entity Manager Factory erstellt, die in der Folge zur Erzeugung von *EntityManager*-Objekten genutzt wird.

Ein Objekt des Typs *EntityManagerFactory* repräsentiert eine der in *persistence.xml* definierten Persistence Units. Bei seinem Aufbau werden die im Deskriptor angegebenen Parameter eingelesen und interpretiert. Die Erzeugung einer Entity Manager Factory ist daher eine schwergewichtige Operation, die man im Programmverlauf aber auch nur einmalig benötigt. Die Methode *Persistence.createEntityManagerFactory* erhält als Parameter den Namen der Persistence Unit. Optional können als zweiter Parameter Properties mitgegeben werden, die die in *persistence.xml* eingetragenen ergänzen oder ersetzen.

Die Methode *EntityManagerFactory.createEntityManager* liefert einen neuen Entity Manager auf Basis der zuvor aufgebauten Factory. Diese Operation ist leichtgewichtiger, kann also ohne Bedenken häufiger ausgeführt werden.

EntityManagerFactory ist threadsafe, *EntityManager* nicht. Will man also mit mehreren Threads im Programm arbeiten, wird nur eine *EntityManagerFactory* benötigt, aber für jeden Thread jeweils ein *EntityManager*.

Die Entity-Manager-Erzeugung lässt sich zum bequemen Aufruf leicht in eine Helferklasse auslagern (Listing 3.125).

```
public class JpaUtil
{
  private static EntityManagerFactory entityManagerFactory;

  public static synchronized
    EntityManagerFactory getEntityManagerFactory()
  {
    if (entityManagerFactory == null)
    {
      entityManagerFactory
        = Persistence.createEntityManagerFactory("eclipselink");
    }
```

```
    return entityManagerFactory;
  }

  public static EntityManager createEntityManager()
  {
    return getEntityManagerFactory().createEntityManager();
  }
}
```

Listing 3.125: Erzeugung von „EntityManagerFactory“ und „EntityManager“

3.9.3 Transaktionssteuerung in Java-SE-Anwendungen

Während der JPA-Provider in einer Enterprise-Umgebung die gerade aktive Transaktion des Applikationsservers übernimmt, muss zur Transaktionssteuerung in einem Java-SE-Programm selbst Hand angelegt werden. Dazu bietet *EntityManager* die Methode *getTransaction* an, die das im Entity Manager eingebaute Transaktionsobjekt vom Typ *EntityTransaction* liefert. Darauf können dann die klassischen Transaktionssteuerungsmethoden *begin*, *commit*, *rollback* etc. angewendet werden (Listing 3.126).

```
EntityManager em = JpaUtil.createEntityManager();
EntityTransaction tx = em.getTransaction();
tx.begin();
City city = new City("Berlin", 3500000, 892);
em.persist(city);
tx.commit();
em.close();
```

Listing 3.126: Transaktionssteuerung in SE-Anwendungen

3.9.4 Schema-Generierung

Im Abschnitt 3.2.2 wurde bereits gezeigt, dass mithilfe von Properties im Deskriptor *persistence.xml* erreicht werden kann, dass der Provider die benötigten Tabellen in der Datenbank anlegt, verändert oder löscht. So kann man bspw. mit den in Listing 3.127 gezeigten Properties für EclipseLink bzw. Hibernate erreichen, dass fehlende Tabellen oder Spalten automatisch hinzugefügt werden.

```
<property name="eclipselink.ddl-generation"
          value="create-or-extend-tables" />
<property name="eclipselink.ddl-generation.output-mode"
          value="database" />
<property name="hibernate.hbm2ddl.auto"
          value="update" />
```

Listing 3.127: Providerspezifische Properties zur Schema-Generierung

Mit JPA 2.1 sind einige Standard-Properties zur Schema-Verwaltung eingeführt worden. Damit können Tabellen erzeugt und gelöscht, initiale Daten geladen sowie DDL-Skripte erzeugt werden (Tabelle 3.3).

Property javax.persistence.…	Bedeutung
schema-generation.database.action *schema-generation.scripts.action*	Auswahl der Befehle, die in der Datenbank ausgeführt bzw. in die Zielskripte geschrieben werden. Erlaubte Werte sind *none*, *create*, *drop-and-create* und *drop*.
schema-generation.create-source *schema-generation.drop-source*	Auswahl, ob das Erzeugen bzw. Löschen von Tabellen auf Basis der Metadaten der persistenten Klassen oder durch die Quellskripte erfolgen soll. Erlaubte Werte: *metadata*, *script*, *metadata-then-script*, *script-then-metadata*.
schema-generation.create-script-source *schema-generation.drop-script-source*	Quellskripte zum Erzeugen bzw. Löschen von Tabellen.
schema-generation.scripts.create-target *schema-generation.scripts.drop-target*	Zielskripte zum Erzeugen bzw. Löschen von Tabellen.
sql-load-script-source	SQL-Skript zum Befüllen von Tabellen.

Tabelle 3.3: Standard-Properties zur Schema-Generierung

Durch die Kombination der Properties sind diverse Operationen möglich. Im Folgenden sollen zwei typische Anwendungen gezeigt werden. Weitere Details finden sich in der Spezifikation in den Abschnitten „persistence.xml, properties“ und „Schema Generation“.

Löschen und Erzeugen der benötigten Tabellen

Sollen die benötigten Tabellen bei Start der Anwendung – genauer: bei Erzeugung der *EntityManagerFactory* – automatisch angelegt werden, muss in *persistence.xml* für die Property *javax.persistence.schema-generation.database.action* einer der Werte *create* oder *drop-and-create* eingetragen werden. Die Tabellen werden dann auf Basis der persistenten Klassen der Anwendung erzeugt. Sind zusätzliche DDL-Befehle erwünscht, können diese in je einem Skript zum Anlegen und Löschen gespeichert werden. Diese Skripte können mit den Properties *javax.persistence.schema-generation.create/drop-script-source* als Classpath-Ressourcen der Persistence Unit referenziert werden. Mit den Properties *javax.persistence.schema-generation.create/drop-source* wird zudem gewählt, ob die Skripte statt der, vor oder nach den Metadatenbefehlen ausgeführt werden sollen (Listing 3.128).

```
<property name="javax.persistence.schema-generation.database.action"
          value="drop-and-create" />
<property name="javax.persistence.schema-generation.drop-script-source"
          value="META-INF/drop.sql" />
<property name="javax.persistence.schema-generation.drop-source"
```

```
        value="script-then-metadata" />
<property name="javax.persistence.schema-generation.create-script-source"
        value="META-INF/create.sql" />
<property name="javax.persistence.schema-generation.create-source"
        value="metadata-then-script" />
```

Listing 3.128: Persistence Unit Properties zum Löschen und Erzeugen von Tabellen

Im gezeigten Beispiel wurden die Skripte *drop.sql* und *create.sql* im *META-INF*-Verzeichnis der Anwendung untergebracht.

Mit einem zusätzlichen Skript können nach dem Erzeugen der Tabellen weitere SQL-Befehle z. B. zum Laden initialer Daten ausgeführt werden. Dieses Skript kann wiederum als Classpath-Ressource vorliegen und mit der Property *javax.persistence.sql-load-script-source* referenziert werden.

Erzeugung von DDL-Skripten

Die Klasse *Persistence* wurde in JPA 2.1 um die Methode *generateSchema* ergänzt, die den Namen einer Persistence Unit und eine Map mit den o. a. Properties als Parameter erhält. Gibt man darin bspw. den Wert *create* für die Property *javax.persistence.schema-generation.scripts.action* an, so werden die SQL-Befehle zur Erzeugung der Tabellen in das Skript geschrieben, das mit der Property *javax.persistence.schema-generation.scripts.create-target* spezifiziert wird. Dies kann ein als *String* angegebener File-URL sein oder auch ein Objekt vom Typ *java.io.Writer* (Listing 3.129).

```
StringWriter writer = new StringWriter();
Map<String, Object> properties = new HashMap<>();
properties.put(
  "javax.persistence.schema-generation.scripts.action", "create");
properties.put(
  "javax.persistence.schema-generation.scripts.create-target, writer);
Persistence.generateSchema("test", properties);
String createScript = createWriter.toString();
…
```

Listing 3.129: Erzeugung eines Skripts zur Erzeugung der Tabellen