Heiko W. Rupp

EJB 3
für Umsteiger

Neuerungen und Änderungen
gegenüber dem EJB-2.x-Standard

dpunkt.verlag

EJB 3 für Umsteiger

Heiko W. Rupp ist Diplom-Informatiker und arbeitet als Entwickler bei Red Hat im Bereich JBoss. Rupp hat zuvor in Consultingunternehmen interne und externe Kunden im Bereich JBoss, Java Enterprise und Persistenz beraten. Darüber hinaus ist er Verfasser mehrerer Artikel in deutschen Fachmagazinen. Als engagierter Open-Source-Anhänger hat er Schreibzugriff auf den JBoss-Quellcode und war in der Vergangenheit auch in anderen OSS-Projekten, wie NetBSD oder Xdoclet, involviert.

Sein erstes Buch »JBoss. Server-Handbuch für J2EE-Entwickler und Administratoren« erschien 2005 im dpunkt.verlag.

Heiko W. Rupp

EJB 3 für Umsteiger

Neuerungen und Änderungen gegenüber dem EJB-2.x-Standard

dpunkt.verlag

Heiko W. Rupp
hwr@pilhuhn.de

Lektorat: René Schönfeldt
Copy-Editing: Annette Schwarz, Ditzingen
Satz: Ulrich Kilian, science & more redaktionsbüro
Herstellung: Birgit Bäuerlein
Umschlaggestaltung: Helmut Kraus, www.exclam.de
Druck und Bindung: Koninklijke Wöhrmann B.V., Zutphen, Niederlande

Bibliografische Information Der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.ddb.de> abrufbar.

ISBN 978-3-89864-429-7

1. Auflage 2007

Ringstraße 19
69115 Heidelberg

5 4 3 2 1 0

Vorwort

Im Mai 2006 wurde die Java Enterprise Edition (Java EE) 5.0 durch Sun für die Allgemeinheit freigegeben. Diese neue Version beinhaltet viele Neuerungen, zu denen insbesondere die Version 3.0 der Enterprise JavaBeans gehört.

Dieses Buch beschreibt insbesondere Aspekte, die sich zwischen den EJB-Versionen 2.1 und 3.0 geändert haben. Es ist also kein Buch, in dem die Enterprise JavaBeans von Grund auf vermittelt werden, sondern es sind einige Vorkenntnisse aus den Vorgängerversionen der Enterprise JavaBeans notwendig, um dem Buch folgen zu können.

Dieses Vorwort erläutert, warum diese Neuerungen notwendig sind, beschreibt die Zielgruppe für dieses Buch und führt danach durch seine Struktur. Am Schluss folgen die Danksagungen.

Warum gibt es EJB 3.0?

Der Enterprise-JavaBeans-2.x-Standard ist in die Jahre gekommen; die Spezifikation ist mit der Zeit immer dicker und komplexer geworden. In EJB 2.x ist sehr viel zusätzlicher technischer Code notwendig, um die Geschäftslogik einer Anwendung zu implementieren. Ein weiterer Nachteil ist, dass sich Eigenschaften der Objektorientierung wie Vererbung nur sehr schwer realisieren lassen – speziell für Entity Beans. In der Praxis behilft man sich mit speziellen Entwurfsmustern und zusätzlichen Werkzeugen, um diese Probleme zu umschiffen.

Mit EJB 3.0 wurden viele dieser Probleme geradegezogen. Eine starke Motivation hierzu war sicher die Konkurrenz durch leichtgewichtigere Frameworks wie Spring, Picocontainer, Hibernate oder JDO.

Auf der anderen Seite standen neue nutzbare Technologien wie die aspektorientierte Programmierung (AOP) oder die mit Java SE 5.0 eingeführten Annotationen.

Die größte Änderung gegenüber früheren Versionen ist eine starke Vereinfachung. Auf der einen Seite wird nun auf einfache Java-Objekte (POJOs) gesetzt. Auf der anderen Seite versucht man dem technischen

Code durch *Dependency Injection* (DI) zu Leibe zu rücken, so dass sich der Entwickler wieder mehr auf die Geschäftslogik konzentrieren kann.

Ein weiteres Designprinzip ist auch die Konfiguration durch Ausnahmen. Dies bedeutet, dass es Standardfälle gibt, für die keine zusätzliche Konfigurationsinformation mehr notwendig ist. Erst wenn man spezielle Wünsche hat, muss man diese gesondert kundtun. Dies trägt dem Fakt Rechnung, dass in der Praxis die in Java EE propagierte Rollenaufteilung nur in seltenen Fällen auftritt; ist dies der Fall, dann liegt es meistens daran, dass nur ein spezieller Administrator Zugriff auf die Zielmaschine hat und nur er die Anwendung dort einspielen darf. Aber selbst in diesem Fall ist es nur selten notwendig, dass dieser Administrator Änderungen an der Applikation vornimmt.

Ein weiterer großer Beitrag kommt von den in Java SE 5 eingeführten Annotationen, die es erlauben, die Metainformationen, die in früheren EJB-Versionen in den Deployment-Deskriptoren hinterlegt wurden, nun in den Class Files zu hinterlegen. Diese Metadaten können weiter durch klassische Deployment-Deskriptoren be- und überschrieben werden. In der Vergangenheit hat hier Xdoclet gute Dienste geleistet und die Komplexität ein gutes Stück vor dem Anwender verborgen. Das jetzige Konzept geht aber sogar so weit, dass in der Praxis Werkzeuge wie eben Xdoclet nicht mehr notwendig sind.

Bei der Entwicklung der EJB-3.0-Spezifikation stand auch immer die Verbesserung der Testbarkeit im Mittelpunkt der Überlegungen. In den Vorgängerversionen sind einfache Unit-Test-Läufe nicht möglich. Es wird immer ein Container benötigt, was sich in langen Entwicklungszyklen niederschlägt. Komponenten in EJB 3.0 sind reine POJOs, so dass diese auch innerhalb von Unit-Tests wie gewohnt getestet werden können.

Für wen dieses Buch ist und für wen nicht

Wie bereits angedeutet, richtet sich dieses Buch in erster Linie an Entwickler, die bereits Erfahrung mit EJB-Versionen vor 3.0 oder Persistenz-Frameworks wie Hibernate, JDO oder Toplink gemacht haben und nun wissen wollen, wie Dinge in EJB 3.0 funktionieren.

Da die Entwicklung mit Hilfe der Annotationen sehr gut von der Hand geht, werden hier im Buch nur diese Annotationen beschrieben. Deployment-Deskriptoren, die genutzt werden können, um mit Annotationen gemachte Einstellungen zu überschreiben, werden im Buch nicht dargestellt.

Für Einsteiger in die Enterprise JavaBeans an sich ist das Buch weniger geeignet, da einiges Wissen vorausgesetzt wird. Die Vermittlung

von EJB 3.0 von der Pike auf sowie die Beschreibung der Deployment-Deskriptoren wird im Buch von Oliver Ihns verfolgt, welches voraussichtlich 2007 im dpunkt.verlag erscheinen wird.

Organisation dieses Buches

Kapitel 1 dokumentiert die Konzepte, die hinter der Entwicklung von EJB 3.0 stehen. Dies sind die Vereinfachung der Entwicklung, die Verbesserung der Testbarkeit und Mechanismen zum Herauslösen von Querschnittsaspekten aus dem Anwendungscode.

Beispiel

In Kapitel 2 wird ein Weblog-System als eine EJB-3.0-Beispielanwendung dokumentiert. Im weiteren Verlauf des Buchs werden Ausschnitte aus dieser Anwendung wieder herangezogen. Der Quellcode der Anwendung kann auf der Begleitseite zum Buch heruntergeladen werden.

Session und Message-Driven Beans

Kapitel 3 beleuchtet die Geschäftslogik der Enterprise JavaBeans, bestehend aus Session- und Message-Driven Beans. Hier hat sich insbesondere die Form der Beans stark vereinfacht – diese sind nun im Prinzip nur noch POJOs mit zusätzlichen Metadaten. Außerdem unterstützen Session Beans nun generische Interceptoren, mit denen Querschnittsfunktionen an zentraler Stelle zur Verfügung gestellt werden können.

Java Persistence API

Mit EJB 3.0 hat sich die Persistenz gewaltig verändert. Waren die Entity Beans in der Vergangenheit eher unflexibel, hat sich dies nun stark geändert. Kapitel 4 stellt Entity Beans als Geschäftsobjekte und die Java Persistence API vor, die sowohl innerhalb von EJB 3.0 als auch in Java-SE-Applikationen eingesetzt werden kann. Da die Abfragesprache JP-QL deutlich erweitert wurde, wird diese in Kapitel 5 eingeführt.

JNDI

Kapitel 6 behandelt die Änderungen in der Namensauflösung über JNDI. EJB-Referenzen oder Umgebungsvariablen können nun vom Container über Dependency Injection bereitgestellt werden.

Webservices

Obwohl Webservices streng genommen nicht zu den Enterprise JavaBeans gehören, wird deren veränderte Entwicklung in Kapitel 7 vorgestellt. Das Kapitel beschränkt sich dabei auf die neuen Webservices mit JAX-WS und JSR-181. Nachdem die Erstellung von Webservices in EJB 2.1 eher mühsam war (siehe z.B. [18]), hat sich dies durch die Annotationen deutlich vereinfacht, so dass es nun richtig Spaß macht.

In Kapitel 8 wird noch auf den Entwicklungsprozess mit EJB 3 und die Generierung von Entity Beans eingegangen.

Der Anhang bietet eine Liste der beschriebenen Annotationen mit einer Kurzerklärung und dem Verweis auf die Stelle der Definition sowie einen Abschnitt über Referenzen und weitere Informationsquellen.

Einige Textstellen sind mit speziellen Symbolen in der Randspalte hervorgehoben. Sie haben folgende Bedeutung:

Das »Achtung!«-Zeichen weist auf mögliche Stolperfallen hin.

Das Informationszeichen weist auf Stellen hin, die zusätzliche Informationen zu Vorgehensweisen oder der unterliegenden Implementierung (eines Dienstes) liefern.

Danksagungen

Wie auch schon beim letzten Buch [18] haben mich einige Personen bei der Entstehung unterstützt. Ihnen gebührt mein Dank:

- Elke Ackermann sowie meinen Eltern und den Schwiegereltern für die Unterstützung und das Babysitten während der Erstellung des Buches. Das Schreiben hat auch bei diesem Buch wieder mehr Zeit benötigt, als ich mir das im Vorfeld vorgestellt hatte.
- Meinem Lektor, René Schönfeldt vom dpunkt.verlag, der mich mit vielen Ratschlägen durch den Buchprozess begleitet hat.
- Den vom Verlag besorgten und mir nicht bekannten Gutachtern, die das Manuskript gelesen und kommentiert haben.

Begleitseite zum Buch

Unter der URL `http://bsd.de/e3fu/` werden Aktualisierungen und Errata abgelegt. Auch die Referenzliste aus Anhang A.2 wird auf dieser Begleitseite in aktualisierter Form bereitstehen.

Inhaltsverzeichnis

1 Konzepte hinter EJB 3.0

Dieses Kapitel beschreibt die neuen Konzepte hinter EJB 3.0. Diese teilen sich in drei grobe Kategorien auf:

- Vereinfachung der Entwicklung
- Bessere Testbarkeit
- Gemeinsame Nutzung von Code

Einzelne Aspekte lassen sich dabei nicht immer eindeutig einer der Kategorien zuordnen.

1.1 Vereinfachung der Entwicklung

Die in diesem Abschnitt vorgestellten Konzepte erlauben eine vereinfachte und damit auch beschleunigte Entwicklung.

1.1.1 Zurück zum POJO

Anstelle aufwändiger Komponentenschnittstellen und der Implementierung von EJB-Schnittstellen sind Enterprise JavaBeans in EJB 3.0 nur noch Java-Objekte (POJOs), die keine speziellen technischen Schnittstellen mehr implementieren müssen.

POJO

Diese Rückkehr zum einfachen Java-Objekt schließt auch die Vererbung mit ein, so dass EJBs problemlos von anderen EJB-Klassen erben können. In den EJB-Versionen vor 3.0 war hier spätestens beim Erstellen der Komponentenschnittstellen Schluss. Entity Beans haben keinerlei Abbildungsstrategie von hierarchischen Objektmodellen auf verschiedene Tabellen geboten. In EJB 3.0 hingegen ist die natürliche Vererbung für alle Enterprise JavaBeans möglich.

Da die EJBs nun POJOs sind, können sie natürlich auch einfach außerhalb eines Applikationsservers genutzt und auch getestet werden. Der Applikationsserver stellt natürlich eine gewisse Infrastruktur bereit, die dann eben in Teilen nachgebildet werden muss.

1.1.2 Konvention statt Konfiguration

In früheren Versionen der EJB-Spezifikation müssen viele Artefakte implementiert werden, selbst wenn diese innerhalb der Applikationslogik nicht benötigt werden oder sie direkt eine gute Voreinstellung darstellten. Beispielsweise ist es notwendig, Callbacks wie *ejbActivate()* zu implementieren, selbst wenn sie nicht genutzt werden.

Die Praxis hat gezeigt, dass in vielen Anwendungsfällen die Voreinstellungen passend sind. Dies greift der EJB-3.0-Standard auf und stellt die Konvention vor die Konfiguration. Dies bedeutet, dass sinnvolle Standards vorgegeben sind. Für exotische Anwendungsfälle können diese dann überschrieben werden.

Als Beispiel seien hier Entity Beans genannt, bei denen Spalten- und Tabellennamen aus den Namen der Klasse und der Felder automatisch abgeleitet werden. Erst wenn diese Vorgaben nicht passen, müssen die Namen explizit gesetzt werden.

1.1.3 Reduktion der notwendigen Artefakte

Wie man in Abbildung 1.1 am Beispiel eines Session Beans sieht, hat sich die Implementierung von Enterprise JavaBeans durch den Wegfall einiger nicht mehr benötigter Artefakte gegenüber EJB 2.x deutlich vereinfacht. Diese Artefakte wurden in der Vergangenheit oftmals via Xdoclet erzeugt, was zwar Tippaufwand ersparte, aber einen zusätzlichen Xdoclet-Lauf im Build erforderte. Neben den weggefallenen Artefakten wurden auch die noch zu implementierenden Artefakte verschlankt.

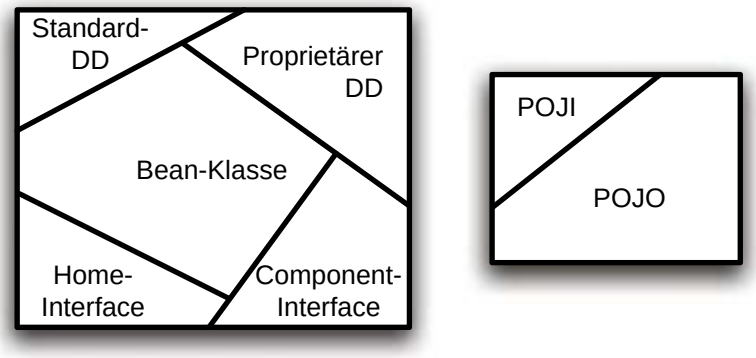


Abbildung 1.1 *Vergleich der für ein Session Bean notwendigen Artefakte in EJB 2.1 (links) und EJB 3.0 (rechts)*

Wie man sieht, sind nur noch die Schnittstellenklasse und die Implementierung dieser Schnittstelle als Session Bean zwingend notwendig. Und auch in der Bean-Klasse ist der Zwang zur Implmentierung

der Callback-Methoden weggefallen, so dass die Implementierung hier wirklich schnell von der Hand geht.

1.1.4 Nutzung von Annotationen für Metadaten

Mit den Annotationen von Java SE 5 gibt es nun einen Weg, Metadaten zu Java-Elementen direkt im Quellcode der Klasse anzugeben. Von dieser Möglichkeit wird in EJB 3.0 reger Gebrauch gemacht. Es ist weiterhin möglich, Metadaten im Deployment-Deskriptor anzugeben, es ist allerdings nicht mehr notwendig.

In sehr vielen Anwendungsfällen wird man nur die Annotationen im Quellcode hinterlegen und keinen Deployment-Deskriptor schreiben. Die Erfahrung hat gezeigt, dass in 80% aller Fälle der Entwickler die Version einer Anwendung erstellt, die auch in der Produktion zum Einsatz kommt. In den restlichen 20% der Fälle gibt es eine Trennung der Rollen – aber auch hier wird nur selten mehr angepasst als der Name einer Datenquelle.

Dies ist aber – entgegen landläufiger Meinungen – immer noch ohne Änderung des Quelltexts über Anpassen des Deployment-Deskriptors möglich. Ist ein Deployment-Deskriptor vorhanden, überschreiben dessen Einträge die Vorgaben aus den Annotationen. Hier hat man also beide Welten miteinander vereint.

Umgekehrt ist es oftmals so, dass eine Änderung von Metadaten nicht ohne eine Änderung am Code sinnvoll ist. Benötigt beispielsweise eine Methode für das korrekte Funktionieren einen Transaktionskontext, so kann man zwar in den Deployment-Deskriptor schreiben, dass die Methode keine Transaktionen verträgt, die Anwendung wird danach aber nicht mehr vernünftig funktionieren. Hier ist es also im Gegenteil sinnvoll, die Metadaten direkt am betreffenden Code zu haben und sie nicht im Deployment-Deskriptor zu ändern oder zu überschreiben.

1.1.5 Injektion von Ressourcen

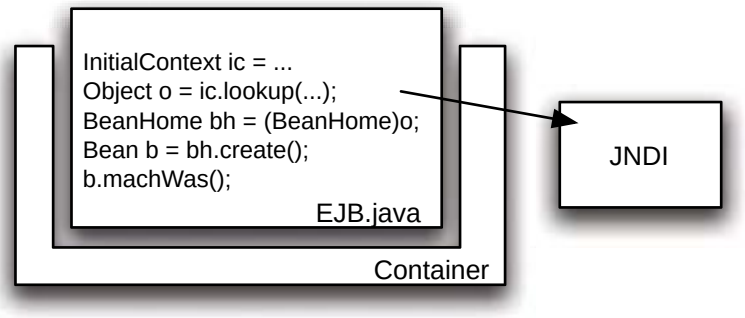
»Don't call us. We'll call you.« Dieser Satz wird auch gerne als das Hollywood-Prinzip bezeichnet. Es soll darstellen, dass Agenten nicht wollen, dass ihre Klienten sie dauernd anrufen, ob sie ein Angebot haben, sondern dass die Agenten die Schauspieler anrufen, wenn sie einen Auftrag haben.

Die sogenannte *Dependency Injection* (DI) folgt diesem Prinzip, indem der Container Abhängigkeiten im Code von externen Daten in den verwalteten Ressourcen von außen setzt. Dies wird auch als *Injektion* bezeichnet.

Injektion

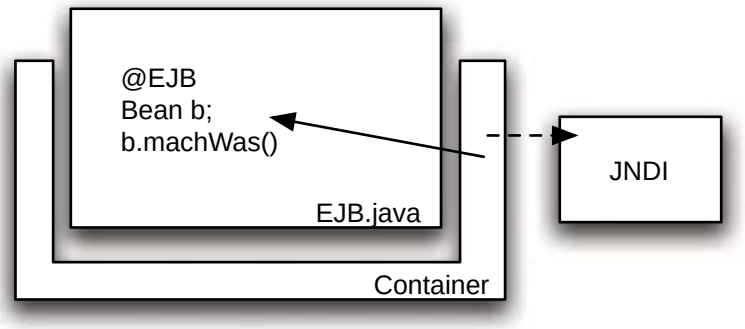
Bis J2EE 1.4 und damit EJB 2.1 musste ein Bean Einträge im JNDI explizit über den *InitialContext* suchen. Dies ist in Abbildung 1.2 gezeigt.

Abbildung 1.2
Naming Lookup klassisch



In EJB 3.0 reicht es aus, die Deklaration des gewünschten Objekts mit einer entsprechenden Annotation zu dekorieren. Der Container stellt daraufhin zur Laufzeit das Objekt zur Verfügung. Dies ist in Abbildung 1.3 dargestellt.

Abbildung 1.3
Naming-Injektion



Inversion of Control

Da bei der DI nicht mehr der Geschäftscode die Anfrage an das JNDI stellt, sondern der Container das Objekt bereitstellt, wird dieses Vorgehen auch Inversion of Control (IoC) genannt. Interessanterweise gab es schon vor EJB 3.0 IoC im Standard: Alle Callback-Methoden werden ja auch vom Container aufgerufen, ohne dass der Programmierer dies explizit veranlassen muss. Auf diese Weise wurde auch der EJBContext zur Verfügung gestellt. Der Hauptunterschied ist nun, dass es durch die Annotationen deutlich eleganter geworden ist.

1.1.6 Verbesserung der Datenbankabfragen

Die Finder in EJB 2.x haben einen relativ beschränkten Funktionsumfang. Sobald man etwas anspruchsvollere Abfragen benötigt, waren entweder herstellerspezifische Erweiterungen oder handgeschriebener SQL-Code notwendig. Letzterer war oftmals sehr kryptisch (jeder, der schon einmal Fragezeichen in einem Statement gezählt hat, kennt dies) und oft auch nicht zwischen Datenbanken portabel.

Die Java Persistence API (JPA) als Teil von EJB 3.0 bietet hier eine sehr umfangreiche Abfragesprache, so dass der Griff zu proprietärem SQL nur noch selten notwendig ist. Innerhalb der Abfragen wird mit Objekten hantiert, so dass von der unterliegenden Datenbank abstrahiert wird. Sollte es einmal notwendig werden, sind native SQL-Abfragen aber weiterhin möglich.

1.2 Verbesserung der Testbarkeit

Alle vorher genannten Änderungen haben neben der Vereinfachung der Programmierung auch die Verbesserung der Testbarkeit zum Ziel. Dadurch dass die EJBs nun einfache Java-Objekte sind und Ressourcen nicht mehr aktiv von EJB im JNDI gesucht werden müssen, sondern über Dependency Injection injiziert werden können, kann das Bean relativ einfach aus der Anwendung herausgelöst werden, um es über JUnit-Tests außerhalb des Applikationsservers zu testen.

Dieses Vorgehen wird auch dadurch unterstützt, dass Entity Beans in EJB 3.0 im Gegensatz zu EJB 2.x normale JavaBeans mit konkreten Zugriffsmethoden (Accessoren) sind, so dass sie keinen Container benötigen, der abstrakte Accessoren implementiert. Ein Entity Bean wird einfach über `new` instanziiert und kann in Unit-Tests vom Test-Framework bereitgestellt werden.

Auf diese Weise erhält man nun sehr kurze Testzyklen, ohne dass man zusätzliche, oftmals schwierig zu nutzende Test-Frameworks für den EJB-Server benötigt.

Die Firma Red Hat stellt beispielsweise in der JBoss-EJB-3-Implementierung auch einen EJB-3-Container zur Verfügung, der innerhalb von Servlet-Containern oder Unit-Tests genutzt werden kann. Die Startzeit des Containers ist sehr kurz, so dass man auf aktueller Hardware bereits nach wenigen Sekunden Tests laufen lassen kann. Startet man Entity Beans innerhalb dieses Containers, bekommt man beispielsweise sehr früh Rückmeldung, ob Named Queries syntaktisch gültig sind oder nicht.

Nutzt man die sogenannte Setter Injection (siehe Abschnitt 6.1 ab Seite 153) für die Enterprise JavaBeans, ist das Test-Framework in der

Lage, anstelle der eigentlichen EJBs oder anderer Ressourcen auch sog. *Mock-Objekte* über die Setter zu setzen, welche dann in den Unit-Tests genutzt werden. Diese Mock-Objekte implementieren die benötigte Schnittstelle oder Klasse und weisen die minimal notwendige Logik auf. Würde die zu injizierende Klasse beispielsweise eine Datenbankabfrage durchführen, könnte hier immer hart verdrahtet ein bekanntes ResultSet zurückgegeben werden.

1.3 Nutzung von gemeinsamem Code

EJB 3.0 bietet nun einen Mechanismus, Code, der mehrfach verwendet wird (dieser Code wird auch als *Crosscutting Concern* bezeichnet), in eigene Methoden auszulagern. Dieser Code wird dann durch das Framework gesteuert aufgerufen und wird als *Interceptor* bezeichnet.

Interceptor

Beispielsweise wäre es sehr umständlich, im Code überall den Ein- und Austritt aus einer Methode mitzuloggen. Ein Interceptor könnte dies in ein paar Zeilen Code erledigen und würde lediglich über eine zusätzliche Annotation an beteiligte Klassen konfiguriert.

Neben der Logging-Funktionalität, die ja das »Hello World« der Interceptoren darstellt und die weiter unten noch weiter beschrieben wird, lassen sich über Interceptoren auch sehr schön andere technische Aspekte kapseln, um sich so in den Bean-Methoden auf die eigentliche Geschäftslogik zu konzentrieren. Die Funktion von Interceptoren ist aber nicht auf technische Aspekte beschränkt. Auch fachliche Funktionalität, wie beispielsweise die Überprüfung auf einen positiven Kontostand vor Ausführung einer fachlichen Methode, ließe sich damit realisieren.

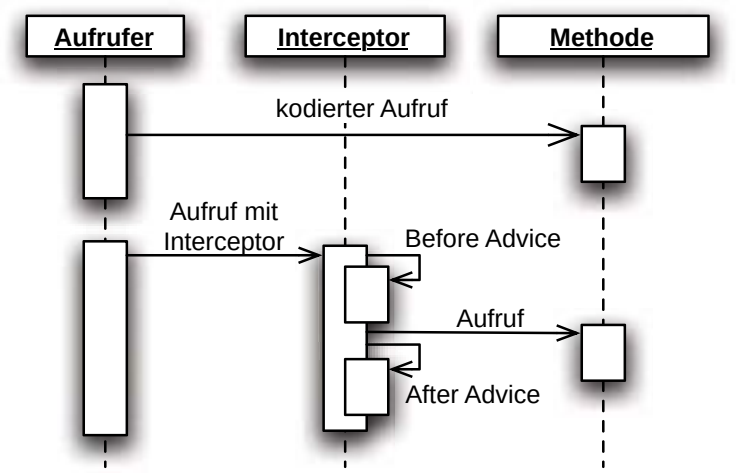


Abbildung 1.4 *Funktionsweise von Interceptoren*

In Abbildung 1.4 ist die Funktionsweise eines Interceptors aufgezeigt. Der Entwickler kodiert dabei einen ganz normalen Methodenaufruf (oberer Aufruf). In Wirklichkeit ruft der Container den Interceptor auf, der Aktionen ausführen kann (*Before Advice*), über `proceed()` wird dann die eigentliche Zielmethode aufgerufen. Danach kehrt der Aufruf in den Interceptor zurück, wo weitere Aktionen ausgeführt werden können (*After Advice*).

2 Ein Weblog als Beispielanwendung

In diesem Kapitel werfen wir einen Blick auf eine EJB-3.0-Beispielanwendung. Diese wird nicht komplett beschrieben, sondern soll einen Aufhänger für den eher referenzartigen zweiten Teil des Buches darstellen. Der Quellcode der Anwendung kann von der Begleitseite zum Buch im Internet heruntergeladen werden.

Als Beispiel soll ein *Weblog* dienen, das mehrere Blogs anbietet, in die Artikel eingestellt werden können. Ein Blog kann einen oder mehrere Nutzer haben, die einerseits nur Benachrichtigungen erhalten, wenn neue Artikel eingestellt werden. Andererseits gibt es Nutzer, die auch selbst Artikel schreiben können.

Damit ergeben sich unter anderem die folgenden Anwendungsfälle:

- Weblogs anlegen und löschen
- Benutzer anlegen und löschen
- Benutzer einem Weblog zuordnen
- Artikel in Weblog einstellen
- Weblogs auflisten
- Artikel im Weblog anzeigen
- Einzelne Artikel im Weblog lesen
- Integration in die JAAS-Autorisierung von Java EE
- Benachrichtigung von Nutzern über neue Artikel in einem Blog (hier nicht ausprogrammiert)

Zugriffe sollen dabei sowohl von Webbrowsern aus möglich sein als auch über Standard-Java-Clients (z.B. Kommandozeilen- oder Swing-Clients). Zusätzlich sollen Artikel auch über Webservices ausgelesen werden können.

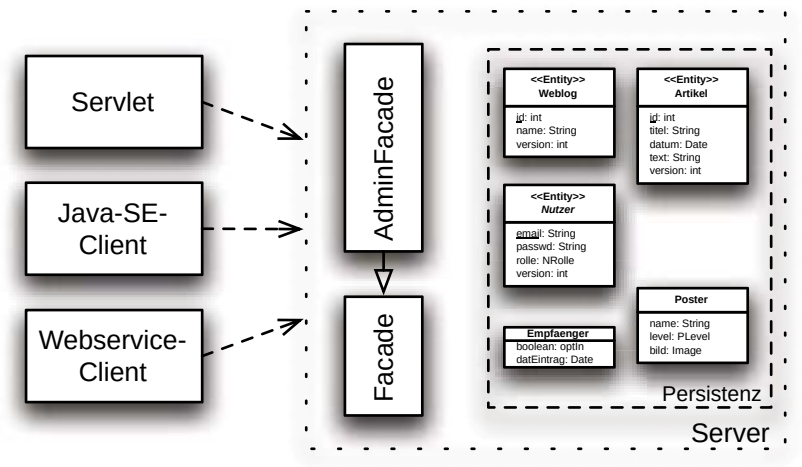
In diesem Kapitel ist wie gesagt nur ein Teil der Anwendung sowie eines Kommandozeilen-Clients abgedruckt. Der vollständige Code inklusive einer WebGUI steht auf der Begleitseite des Buches unter `http://bsd.de/e3fu/` zur Verfügung.

2.1 Gesamtarchitektur

Aufgrund der Anforderungen (und um beim Thema des Buches zu bleiben :-)) wird die Geschäftslogik über Enterprise JavaBeans realisiert. Servlets oder andere Arten von Clients übernehmen lediglich die Aufbereitung der Daten für die Anzeige.

Abbildung 2.1 zeigt die Gesamtarchitektur der Anwendung. Im Weiteren konzentrieren wir uns auf den mit *Server* markierten Bereich. Ein Java-SE-Client ist am Ende des Kapitels ebenfalls skizziert.

Abbildung 2.1
Gesamtarchitektur des Weblogs



Zentrale Zugangspunkte zur Logik auf dem Server sind zwei Fassaden, die durch Stateless Session Beans realisiert sind. Die Aufteilung der Session Beans folgt dabei einer Trennung der Aufgaben. Eine Fassade bietet Dienste für die Administration der Anwendung (beispielsweise Benutzer und Weblogs anlegen), während die andere für das Anlegen und Anzeigen von Artikeln genutzt wird. Anfragen von Clients werden mit Hilfe der Persistenzklassen an die Datenbank weitergeleitet. Hierfür werden sowohl vorbereitete Abfragen (Named Query), die den Findern in EJB 2.x ähneln, als auch Ad-hoc-Abfragen verwendet.

2.2 Geschäftsobjekte

Die Geschäftsobjekte geben die oben genannten Objekte und Beziehungen zwischen diesen Objekten wieder. Ein Weblog kann viele Artikel haben, die jeweils durch einen Poster geschrieben wurden. Gleichzeitig hat ein Weblog eine Menge von Nutzern, die entweder Benachrichti-

gungsmails erhalten können oder die selbst in der Lage sind, Artikel für dieses Weblog zu schreiben.

An dieser Stelle kann man sich auch als Erweiterung vorstellen, dass ein Weblog als privat gekennzeichnet wird und nur eine bestimmte Menge von Nutzern die Einträge des Weblogs lesen darf.

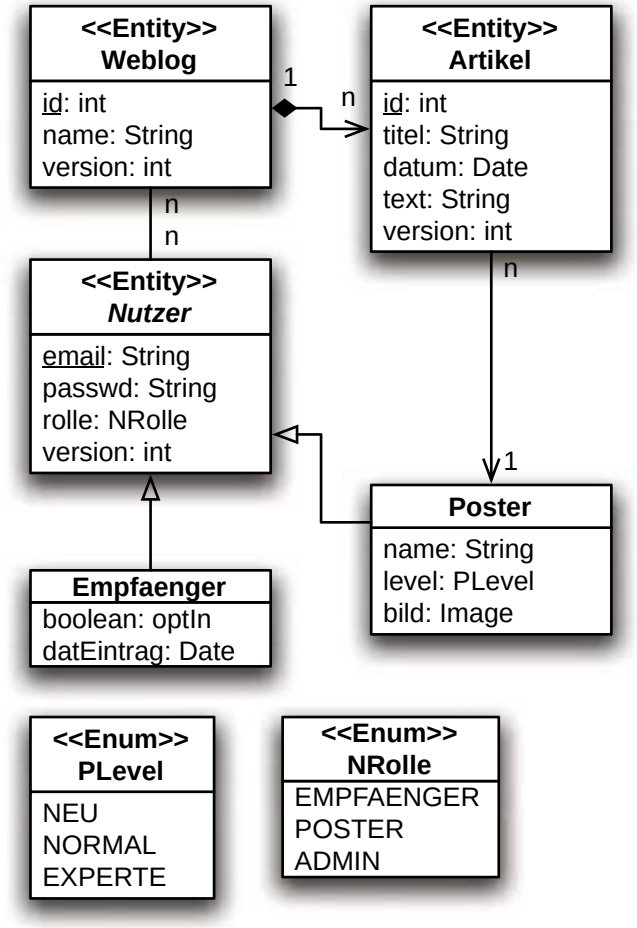


Abbildung 2.2
Geschäftsobjekte des Weblogs

In den beiden Klassen PLevel und NRolle sind typsichere Aufzählungen für einige verwendete Konstanten hinterlegt.

In den nachfolgenden Abschnitten werden die einzelnen Klassen nun vorgestellt, wobei nur die relevanten Teile des Codes abgedruckt sind. Der vollständige Code steht auf der Begleitseite zum Buch zum Download bereit.

2.2.1 Die Klasse Weblog

In dieser Klasse werden die einzelnen Weblogs verwaltet.

Listing 2.1 *Auszug aus Weblog.java*

```
@NamedQueries({
  @NamedQuery(name="WeblogNachName",
    query="SELECT w FROM Weblog AS w WHERE "+
      "w.name = :blog"),
  @NamedQuery(name="ArtikelInWeblog",
    query="SELECT COUNT(*) FROM Artikel AS a "+
      "WHERE a.weblog.name = :weblog")
})
@Entity
@Table(name="WL_WEBLOGS")
public class Weblog
{
```

Zunächst werden Abfragen in der Abfragesprache JP-QL definiert, die später im Code genutzt werden können. Zu übergebende Parameter werden als benannte Parameter angegeben, die durch einen Doppelpunkt eingeleitet werden (:blog, :weblog). Die Klasse wird über `@Entity` als Entity Bean gekennzeichnet, und über `@Table` wird der Tabellenname angegeben.

```
private Long id; // Primärschlüssel
private String name; // Name des Blogs
private Collection<Artikel> artikel =
    new ArrayList<Artikel>();
private Collection<Nutzer> nutzer =
    new ArrayList<Nutzer>();
private long version;

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

@SuppressWarnings("unused")
private void setId(Long id) { this.id = id; }

/**
 * Der Name des Blogs. Dieser muss eindeutig sein.
 */
@Column(unique=true)
public String getName() { return name; }
```

Der Primärschlüssel, *id*, wird vom Persistenz-Provider automatisch erzeugt. Wie dies genau geschieht, hängt dabei von der Datenbank ab (*GenerationType.AUTO*) und wird vom Provider automatisch ausgewählt.

Zusätzlich zum Primärschlüssel hat das Blog einen Namen, der ebenfalls eindeutig sein muss. Dieser hätte also auch als Primärschlüssel verwendet werden können. `@Column` hält diese Vorgabe fest, wird aber im Wirkbetrieb vom System nicht ausgewertet. Die Angabe dient lediglich als Zusatzinformation für die Generierung des Schemas.

```
/** Ein Weblog kann mehrere Nutzer haben. */
@JoinTable(name="WL_Weblog_Nutzer")
@ManyToMany
public Collection<Nutzer> getNutzer() {
  return nutzer;
}
```

Die Relation zu den Nutzern des Blogs wird über `@ManyToMany` gekennzeichnet. Über `@JoinTable` wird der vom System bestimmte Tabellenname der Join-Tabelle überschrieben. Diese Relation ist in diesem Fall unidirektional – in der Klasse Nutzer gibt es kein Feld, über das direkt die Weblogs eines Nutzers ermittelt werden können. Da die Collection in der Generics-Form angegeben ist, ist es nicht notwendig, die andere Seite der Relation innerhalb der `@ManyToMany`-Annotation extra anzugeben.

```
/** Ein Weblog kann viele Artikel enthalten.*/
@OneToMany(mappedBy="weblog",
           cascade=CascadeType.REMOVE)
public Collection<Artikel> getArtikel()  {
  return artikel;
}
```

Die Relation zu den Artikeln ist bidirektional. Da hier immer die N-Seite die führende Seite ist, muss das entsprechende Feld der anderen Seite der Relation über den Parameter *mappedBy* von `@OneToMany` angegeben werden. Über *cascade=CascadeType.REMOVE* wird sichergestellt, dass durch das Löschen eines Blogs automatisch auch die darin veröffentlichten Artikel gelöscht werden.

Zu guter Letzt benötigen wir für das standardmäßig durchgeführte *Optimistic Locking* noch eine Versionsspalte. Diese darf nur vom Persistenz-Provider verändert werden, weswegen der Setter private ist.

```
/** Versionierungsspalte für Optimistic Locking */
@Version
public long getVersion() { return version;  }

@SuppressWarnings("unused")
private void setVersion(long version) {
  this.version = version;
}
```

2.2.2 Die Klasse Artikel

Die Klasse Artikel speichert alle Informationen über einen einzelnen Artikel. Zusätzlich zur Angabe des Tabellennamens via `@Table` und der Definition der Klasse als Entity Bean via `@Entity` stehen an der Klasse noch zwei sogenannte *Native Queries*. Die erste ist eine `@NamedNativeQuery` in SQL (hier PostgreSQL-spezifisch) mit einem zusätzlichen Marker, der das Mapping des Ergebnisses der SQL-Abfrage auf ein JDBC-ResultSet bestimmt (`@SqlResultSetMapping`).

Listing 2.2 *Auszug aus der Klasse Artikel*

```
@NamedNativeQuery(name="EinExzerpt",
  resultSetMapping="toString",
  // Die Query ist spezifisch für Postgres
  query="SELECT substring(text for 100) AS start" +
" FROM WL_Artikel WHERE id = ?1")
@SqlResultSetMapping(name="toString",
  columns=@ColumnResult(name="start"))

@NamedQuery(name="ArtikelProJahr",
  query="SELECT COUNT(*) FROM Artikel AS a "
    +"WHERE a.weblog.name = :weblog "
    +"AND a.datum >= :start AND a.datum < :end"
)

@Entity
@Table(name="WL_Artikel")
public class Artikel implements Serializable
{
  private static final long serialVersionUID = 1L;
  private Long id; // Primärschlüssel
  private Date datum; // Datum des Postings
  private String titel;
  private Poster poster;
  private Weblog weblog;
  private String text; // Body-Text
  private long version;
```

Die zweite Query ist in JP-QL formuliert und über `@NamedQuery` definiert. Die Named Queries werden dann später in der Fassade über *EntityManager.createNamedQuery()* angesprochen.

```
/** Das Datum des Artikels */
@Temporal(TemporalType.DATE)
public Date getDatum() { return datum;  }
```

Das Datum, an dem der Artikel geschrieben wurde, wird über ein Feld vom Typ `java.util.Date` angegeben. Über `@Temporal` wird bestimmt, dass nur das Datum ohne Uhrzeit in der Datenbank gespeichert werden soll.

```
/** Text des Artikels */
@Lob
public String getText() { return text; }
```

Da der Artikeltext beliebig groß werden kann, wird über `@Lob` definiert, dass er über ein Locator-Objekt gespeichert werden soll. Weil das Feld ein String ist, erfolgt die Speicherung als CLOB.

Der Rest der Klasse besteht noch aus der Definition des Primärschlüssels, der Versionierungsspalte und der Relationen zu Poster und Weblog, wobei hier nur Letztere gezeigt wird.

```
/** Ein Weblog kann viele Artikel enthalten */
@ManyToOne
public Weblog getWeblog() { return weblog; }
```

Über `@ManyToOne` wird hier die N-Seite der Relation definiert. Das Feld *weblog* korrespondiert dabei mit dem *mappedBy*-Parameter von Weblog.getArtikel(), wie er oben gezeigt wurde.

2.2.3 Die Klasse Nutzer mit Subklassen

Alle Nutzer müssen eine eindeutige E-Mail-Adresse vorweisen. Diese wird in dieser Klassenhierarchie als Primärschlüssel verwendet, um die Verwendung von nicht generierten Schlüsseln zu zeigen.

Die Basisklasse *Nutzer* ist eine abstrakte Klasse, da sie selbst nicht instanziiert werden soll. Sie bietet die Basis für die konkreten Klassen *Poster* und *Empfänger*.

Listing 2.3
Auszug aus der Klasse Nutzer

```
@Entity
@Table(name="WL_NUTZER")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
public abstract class Nutzer implements Serializable {
```

Über `@Inheritance` wird die Abbildungsstrategie der Klassen auf Datenbanktabellen festgelegt. In diesem Fall werden alle drei Klassen innerhalb derselben Tabelle abgelegt.

```
/**
 * E-Mail des Benutzers. Diese
 * dient auch als Primärschlüssel und muss
 * eindeutig sein.
 */
@Id
@Column(length=64)
public String getEmail() { return email;  }

private void setEmail(String email) {
  this.email = email;
}
```

Wie gesagt, ist die E-Mail-Adresse der Primärschlüssel der Hierarchie und muss beim Anlegen eines Nutzers mitgegeben werden. Der Setter ist trotzdem als privat markiert, da die E-Mail-Adresse nach dem Anlegen eines Nutzers nicht mehr geändert werden darf.

```
/**
 * Rolle des Benutzers.
 * Diese wird in der DB als String
 * abgelegt.
 */
@Column(length=12)
@Enumerated(EnumType.STRING)
public NRolle getRolle() { return rolle; }
```

Jeder Nutzer im System bekommt eine Rolle aus dem Enum *NRolle* zugewiesen. Die Rollen werden dabei als Zeichenkette mit maximal 12 Zeichen in der Datenbank abgelegt. Dies geschieht, damit das JAAS-Subsystem des Applikationsservers die Rolle eines Nutzers zur Autorisierung heranziehen kann. Die Werte in der Spalte Rolle können dann direkt in `@RolesAllowed`-Annotationen angegeben werden, wie dies unten beim *AdminSessionBean* gezeigt wird.

2.2.4 Die Klasse Empfaenger

Die Klasse *Empfaenger* ist sehr einfach und definiert im Wesentlichen nur zwei zusätzliche Felder:

Listing 2.4 *Auszug aus der Klasse Empfaenger*

```
@Entity
public class Empfaenger extends Nutzer
  implements Serializable
{
  boolean optIn;
  Date datEintrag;
```

Die Klasse erweitert die Klasse *Nutzer* von oben. Durch die Markierung mit `@Entity` ist angedeutet, dass ihre Felder ebenfalls persistiert werden sollen.

2.2.5 Die Klasse Poster

Die Definition der Klasse *Poster* beginnt mit einer Reihe von Named Queries, die über `@NamedQueries` zusammengefasst sind.

Listing 2.5 *Auszug aus der Klasse Poster*

```
@NamedQueries({
  @NamedQuery(name="PosterListe",
    query="SELECT p FROM Poster p"),
  @NamedQuery(name="PosterNachName",
    query="SELECT p FROM Poster p WHERE p.name = :name")
})
@SecondaryTable(name="WL_PExtra")
@Entity
public class Poster extends Nutzer
    implements Serializable
{
```

Die Annotation `@SecondaryTable` definiert, dass Daten dieser Klasse innerhalb einer zusätzlichen Tabelle abgelegt werden sollen. Welche dies sind, muss dann am jeweiligen Feld angegeben werden.

```
/**
 * Erfahrungslevel des Nutzers
 */
@Enumerated(EnumType.ORDINAL)
public PLevel getLevel() { return level;  }
```

Der Level des Posters gibt an, wie sehr er im Umfang mit der Applikation geübt ist, und wird als Zahl in der Datenbank abgelegt.

```
/**
 * Der Geek Code des Posters
 * Liegt in einer separaten Tabelle und soll
 * nur bei Bedarf geladen werden.
 */
@Lob
@Column(table="WL_PExtra")
```

```
@Basic(fetch=FetchType.LAZY)
public String getGeekCode() { return geekCode; }
```

Das Feld *geekCode* soll als CLOB in der Datenbank gespeichert werden. Allerdings soll dies nicht in der Haupttabelle geschehen, sondern in der Tabelle *WL_PExtra*, die bereits auf Klassenebene via `@SecondaryTable` definiert wurde. Über `@Basic` wird dem System noch mitgeteilt, dass dieses Feld nur bei Bedarf geladen werden soll.

Eine Versionsspalte für das Optimistic Locking sowie ein Primärschlüssel wird in der Klasse *Poster* nicht benötigt, da diese ja bereits in der Superklasse *Nutzer* definiert wurden.

2.3 Geschäftslogik

Die Geschäftslogik besteht aus zwei Session-Bean-Klassen, deren Aufgaben sich in den Namen widerspiegeln:

- FacadeSessionBean: Allgemeine Methoden, die von allen Clientbenutzern aufgerufen werden können. Dies sind beispielsweise Methoden zur Anzeige eines Eintrags, dem Auflisten aller Blogs oder auch zum Hinzufügen von neuen Artikeln.
- AdminFacadeSessionBean: Methoden zur Administration des Weblogsystems. Hierunter fallen Methoden zum Anlegen neuer Benutzer und neuer Blogs oder dem Zuordnen von Nutzern zu Blogs. Alle Methoden dieser Klasse dürfen nur von Nutzern aufgerufen werden, welche in der Rolle *ADMIN* sind.

Beide Session Beans bestehen aus einer Schnittstellenklasse und einer Implementierungsklasse. Die Schnittstellenklasse ist ein einfaches Java Interface (POJI), das über `@Remote` als Remote-Schnittstelle gekennzeichnet ist.

Listing 2.6 *Kennzeichnung als Remote-Schnittstelle*

```
@Remote
public interface Facade
{
  public int anzahlArtikelInBlog(String blog);
  public Long artikelHinzufuegen(Artikel art,
    String posterId, String weblog);
  //...
}
```

Alle in der Schnittstellenklasse aufgeführten Methoden sind dann für Clients sicht- und aufrufbar.

2.3.1 Die Klasse FacadeSessionBean

Das Facade Session Bean ist ein Stateless Session Bean und wird über `@Stateless` an der Bean-Klasse entsprechend gekennzeichnet.

Listing 2.7
Auszug aus der Klasse FacadeSessionBean

```
@PermitAll
@Stateless
public class FacadeSessionBean implements Facade
{
  @PersistenceContext
  private EntityManager em;
```

Über `@PermitAll` wird der Zugriff auf alle Methoden der Klasse für alle Rollen freigegeben. Dies ist notwendig, da die Admin Facade die Rolle *ADMIN* verlangt und damit dann alle Methoden in der Anwendung auf die Rolle des Aufrufers überprüft. Sind Methoden dann nicht entsprechend markiert, können sie von niemandem aufgerufen werden.

Über `@PersistenceContext` wird ein Entity Manager durch den Container in das Feld *em* injiziert. Dadurch muss der Entity Manager nicht vom Anwendungscode im JNDI gesucht werden.

```
public int anzahlArtikelInBlog(String blog)
{
  Query q = em.createNamedQuery("ArtikelInWeblog");
  q.setParameter("weblog", blog);
  Object o = q.getSingleResult();
  return ((Long)o).intValue();
}
```

Die Methode *anzahlArtikelInBlog()* erzeugt eine Named Query, die ihrerseits in der Klasse Weblog definiert wurde (siehe Listing 2.1 auf Seite 12). Der Parameter *weblog* wird entsprechend befüllt und die Abfrage gestartet. Die Abfrage *ArtikelInWeblog* liefert hier kein Entity Bean, sondern ein Objekt vom Typ Long zurück.

```
public Long artikelHinzufuegen(Artikel art,
    Poster post, Weblog blog)
{
  // ... Check der Eingangsdaten

  // Beziehungen zwischen den Beans setzen
  art.setPoster(post);
  post.addArticle(art);
  art.setWeblog(blog);
  blog.getArtikel().add(art);
```

```
    // Persistieren
    em.persist(art); // Artikel neu anlegen

    return art.getId();
  }
```

Beim Hinzufügen eines neuen Artikels zu einem Blog müssen im Wesentlichen die beteiligten Beziehungen zwischen den beteiligten Objekten gesetzt werden. Da die Relationen bidirektional sind, muss dies jeweils an beiden beteiligten Objekten geschehen.

Danach wird der neue Artikel initial persistiert und die beiden anderen Entity Beans werden ebenfalls aktualisiert.

2.3.2 Die Klasse AdminFacadeSessionBean

Diese Klasse erweitert das FacadeSessionBean und kann so deren Methoden nutzen, ohne auf eine zusätzliche Instanz einer Facade zugreifen zu müssen. Die natürliche Vererbung von Java-Klassen ist also auch innerhalb von Session Beans möglich. Dies ist allerdings nicht explizit im Standard festgehalten und kann möglicherweise nicht portabel sein.

Listing 2.8 *Ausschnitt aus der Klasse AdminFacade-SessionBean*

```
@Stateless
@RolesAllowed("ADMIN")
public class AdminFacadeSessionBean
  extends FacadeSessionBean implements AdminFacade
{
  @PersistenceContext
  private EntityManager em;
```

Auch dieses Session Bean wird über `@Stateless` als Stateless Session Bean gekennzeichnet. Da die Methoden der Klasse nur von Benutzern in der Rolle *ADMIN* aufgerufen werden sollen, wird dies dem System über `@RolesAllowed` mitgeteilt.

```
public boolean posterZuWeblog(String poster,
    String weblog)
{
  Weblog wl = weblogNachName(weblog);
  Poster p = posterNachName(poster);
  wl.getNutzer().add(p);

  return true;
}
```

In der Methode *posterZuWeblog* werden zuerst das zu den übergebenen Parametern passende Weblog und der passende Poster ermittelt. Danach wird dem Weblog in der Liste der Nutzer dieser neue Poster

hinzugefügt. Am Ende der laufenden Transaktion aktualisiert das System dann das Weblog-Objekt.

2.4 Zusätzliche Features

Zusätzlich zur Grundfunktionalität sind noch zwei weitere Features in die Anwendung eingebaut, die nun beschrieben werden.

2.4.1 Logging der Aufrufzeiten für Methoden

Für das FacadeSessionBean ist ein *Interceptor* definiert. Das Konzept der Interceptoren ist in Abschnitt 1.3 beschrieben. Für alle Methodenaufrufe wird im Interceptor die Start- und Endzeit des Aufrufs festgehalten und die Differenz dann im Logfile ausgegeben. Der Quellcode des eigentlichen Interceptors ist in Kapitel 3 in Listing 3.14 auf Seite 48 abgedruckt.

Eingebunden wird der Interceptor auf Klassenebene wie folgt:

Listing 2.9
Interceptor einbinden

```
@Interceptors(TimingInterceptor.class)
public class FacadeSessionBean implements Facade
{
  ...
```

Damit wird der in der Klasse *TimingInterceptor* definierte Interceptor für alle Methoden des FacadeSessionBeans aktiviert.

Die Ausgaben des Interceptors werden über den Loglevel *DEBUG* ausgegeben. Je nach Konfiguration des Applikationsservers werden sie dadurch in ein Logfile des Servers geschrieben (hier für den Druck umbrochen):

```
2006-11-04 10:12:14,022 DEBUG [de.bsd.weblog.interceptors
  .TimingInterceptor] Call to holeArtikelNachId took 13ms
```

Mehr zu Interceptoren gibt es im Abschnitt 3.4 ab Seite 46.

2.4.2 Bereitstellung als Webservice

Einige der Methoden der Applikation werden als Webservice exportiert, um so beispielsweise von .NET-Clients angesprochen werden zu können. Da wir die Anwendung hier glücklicherweise auf der grünen Wiese erstellt haben, müssen wir kein existierendes WSDL berücksichtigen. Damit reicht es aus, die Klassen und Methoden entsprechend zu markieren. Der Server (im Fall von JBoss) erledigt den Rest für uns.

Auf der Klassenebene sieht die notwendige Annotation dann so aus:

Listing 2.10 *Export als Webservice*

```
// JBoss-spezifisch. Legt den Webservice + wsdl nach
// http://localhost:8080/weblogws/FacadeSessionBean?wsdl
@PortComponent(contextRoot="/weblogws")

// Exportieren der Klasse als Webservice.
@WebService(targetNamespace="http://bsd.de/weblog/")
@Stateless
public class FacadeSessionBean implements Facade {
```

Über `@WebService` wird die ganze Klasse als Webservice gekennzeichnet. Die JBoss-spezifische Annotation `@PortComponent` legt fest, dass die Context Root des Webservices */weblogws* lauten soll. Damit kann die WSDL-Datei von Clients von der URL

```
http://server:port/weblogws/FacadeSessionBean?wsdl
```

bezogen werden, wobei *server* und *port* die IP-Adresse und den HTTP-Port angeben.

Da wir nur ein paar der Methoden exportieren wollen, werden diese explizit über `@WebMethod` markiert:

```
@WebMethod
public int anzahlArtikelInBlog(String blog)
{
   ...
```

Mehr zu Webservices gibt es in Kapitel 7 ab Seite 165.

2.5 Zusammenbau und Start

Die Anwendung wird mit Hilfe des Java-Compilers übersetzt und dann mittels `jar` zusammengepackt. Da alle notwendigen Metainformationen für die eigentlichen Mappings etc. in den Annotationen enthalten sind, muss lediglich die Datei *persistence.xml* mit in das Verzeichnis *META-INF* des Jar-Archivs gepackt werden.

Zum Kompilieren werden noch einige Bibliotheken im Klassenpfad benötigt. Diese sind in der beiliegenden Datei *build.xml* in der Pfadangabe `server.path.ref` zusammengefasst:

- ejb3-persistence.jar: Klassen der JPA
- jboss-ejb3x.jar: EJB-3-Klassen ohne JPA
- javax.servlet.jar: Servlet-Klassen für die Webanwendung
- log4j.jar: Klassen des verwendeten Loggers
- jbossws.jar: Klassen für die Webservices. Dieses Jar ist aus der Datei *deploy/jbossws.sar* extrahiert.

Auf anderen Applikationsservern können natürlich die entsprechenden Jars dieser Server verwendet werden. Eine Anleitung zur Installation einer entsprechenden JBoss-Instanz gibt es auf der Begleitseite zum Buch.

Die wichtigsten Ziele in *build.xml* sind:

- all: Übersetzt die gesamte Serveranwendung und erstellt die Datei *weblog.ear*.
- clean: Löscht temporäre Dateien und Verzeichnisse.
- deploy: Kopiert das generierte Archiv, weblog.ear, auf den Server. Hierbei wird eine Konfiguration *weblog* erwartet und der JBoss-Server in */jboss* im Dateisystem. Diese beiden Einstellungen können aber oben in der Datei *build.xml* geändert werden.
- ejb: Kompiliert den EJB-Teil der Anwendung.

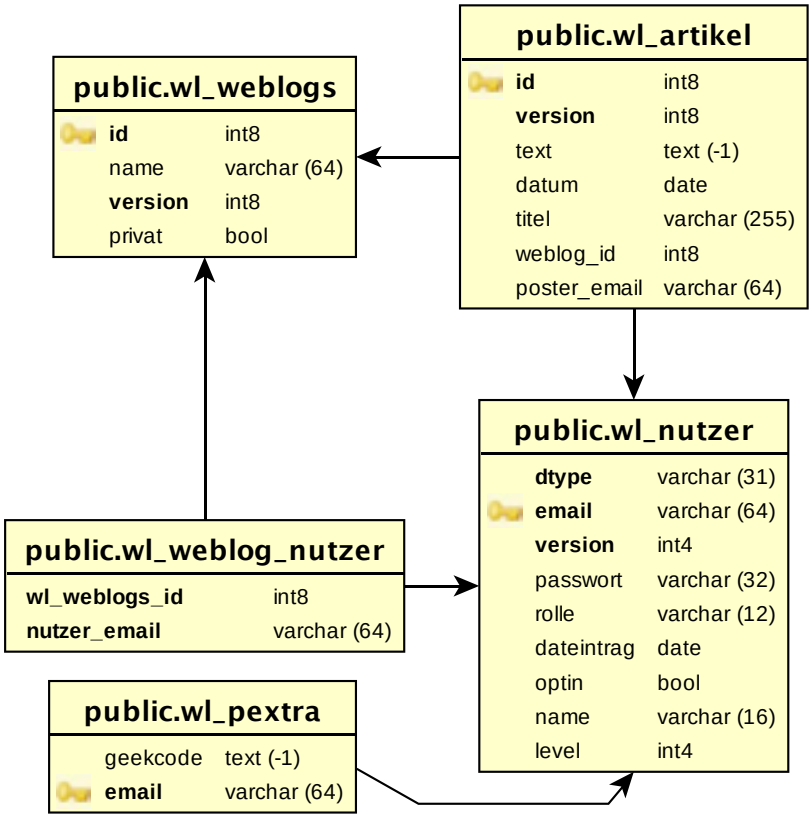


Abbildung 2.3 *Generierte Datenbanktabellen der Anwendung*

Nach dem Übersetzen und Einspielen der Anwendung, kann man den Applikationsserver starten. Falls der Server nicht in der Lage ist, beim Start der Anwendung die Tabellen selbst anzulegen, muss dies

noch von Hand oder über entsprechende Werkzeuge des Herstellers geschehen. Die JBoss-EJB3-Implementierung ist dazu in der Lage, wenn in der Datei *persistence.xml* die Zeile

```
<property name="hibernate.hbm2ddl.auto" value="create"/>
```

hinzugefügt wird, wie in Listing 4.48 auf Seite 137 gezeigt.

Startet man das obige Archiv (im JBoss-Server durch einfaches Kopieren der JAR-Datei in das Verzeichnis *deploy/*) und lässt sich die Tabellen vom Applikationsserver oder einem zusätzlichen Werkzeug generieren, erhält man die in Abbildung 2.3 gezeigte Struktur (die abgebildeten Datentypen gelten für die PostgreSQL-Datenbank).

Die gezeigten Datentypen sind spezifisch für PostgreSQL und werden von der Java Persistence API im Betrieb automatisch umgesetzt.

Schön sieht man auch, wie durch die Strategie der Abbildung der Klasse Nutzer mit Subklassen in eine Tabelle alle Felder in der Tabelle *WL_Nutzer* abgelegt werden. Dies wurde durch `@Inheritance` gesteuert, wie in Listing 2.3 gezeigt.

In der Tabelle *WL_Nutzer* wurde noch eine zusätzliche Spalte *dtype* vom System hinzugefügt. In dieser sog. *Diskriminator-Spalte* ist abgelegt, aus welcher Klasse ein Datensatz stammt.

2.6 Ein Kommandozeilen-Client

Dieser Abschnitt zeigt nun noch ausschnittsweise, wie man mit einem Kommandozeilen-Client auf das Weblog zugreifen kann. Der Code der Beispielanwendung auf der Begleitseite umfasst auch eine WebGUI, mit Hilfe derer man mit dem Browser auf die Geschäftslogik zugeifen kann.

Listing 2.11 *Auszug aus WeblogClient.java*

```
public class WeblogClient
{

  @EJB public static Facade fa;
  @EJB public static AdminFacade afa;

  public static void main(String[] args)
  {
```

Die Facade und AdminFacade werden hier via `@EJB` markiert. Je nach Implementierung des Client Containers kann das System die beiden EJBs injizieren. Ist dies nicht der Fall, dient die Annotation immer noch als gute Gedankenstütze beim Lesen des Quellcodes.

```
try {
  // JNDI Server wird über jndi.properties bestimmt.
  InitialContext ic = new InitialContext();

  if (fa==null) {
    System.out
       .println("Injection hat nicht funktioniert");
    fa = (Facade)ic
       .lookup("weblog/FacadeSessionBean/remote");
    afa = (AdminFacade)ic
       .lookup("weblog/AdminFacadeSessionBean" +
                   "/remote");
  }
```

Hat das Setzen der beiden Variablen durch den Container nicht funktioniert, können sie auf klassische Weise im JNDI gesucht werden. Man sieht hier auch, dass kein Home Interface mehr notwendig ist. Die Suche im JNDI liefert direkt die Remote-Schnittstelle der Beans zurück.

```
// ...
count = fa.anzahlArtikelInBlog("default");
```

Aufrufe von Servermethoden erfolgen dann wie gewohnt über die Schnittstellen der beiden Facaden.

2.7 Zusammenfassung

Dieses Beispiel zeigt in Ausschnitten den Quellcode einer mit EJB 3.0 realisierten Anwendung. Weitere Features wie der Export von Methoden als Webservice oder ein Interceptor zur Zeitmessung der Methodenaufrufe ist im Quellcode ebenfalls vorhanden und wird in den Kapiteln über die Geschäftslogik und Webservices noch beschrieben.

Durch die Nutzung von Annotationen und der Injektion von Ressourcen durch den EJB-Container ist der Quellcode sehr übersichtlich geworden. Die in vorherigen EJB-Versionen notwendigen Zusatzartefakte können hier komplett weggelassen werden. Dies vermeidet Fehler und hilft auch, deutlich schnellere Entwicklungszyklen zu erreichen.

3 Geschäftslogik mit Session und Message-Driven Beans

Die Geschäftslogik wird bei den Enterprise JavaBeans üblicherweise durch Session Beans und Message-Driven Beans bereitgestellt. Session Beans werden dabei vom Client direkt und synchron aufgerufen, während Message-Driven Beans über asynchrone Events aufgerufen werden. Dieses Verhalten ist zwischen EJB 2.x und EJB 3.0 gleich geblieben. Allerdings hat sich die Entwicklung unter anderem durch den Wegfall nicht zwingend notwendiger Artefakte sehr vereinfacht.

Dieses Kapitel stellt zunächst Session und Message-Driven Beans in EJB 3.0 vor, um danach Features wie Interceptoren Security und Transaktionen zu beschreiben, die für diese beide Arten der Enterprise JavaBeans zutreffen.

3.1 Session Beans

Session Beans werden zum Abarbeiten von Geschäftslogik auf dem Server verwendet. Ein Client (in derselben JVM oder aus einer anderen heraus) erhält einen Verweis auf ein serverseitiges Session Bean, dessen Methoden dann aufgerufen werden können. Der Client muss sich dabei nicht selbst um die (eventuell) notwendige Netzwerk- oder RMI-Verbindung kümmern – dies erledigt das Framework. Die Methodenaufrufe erfolgen dabei synchron. Solange also die aufgerufene Methode aktiv ist, blockiert der Aufrufer.

Session Beans sind in EJB 3.0 nur noch ganz gewöhnliche Java-Objekte (*Plain Old Java Object, POJO*), die mit einer Annotation `@Stateless` oder `@Stateful` versehen werden, um anzuzeigen, um welche Unterart von Session Bean es sich handelt. Diese Unterarten werden unten noch beschrieben. *POJO*

Zusätzlich zur Implementierungsklasse müssen die Geschäftsmethoden, die den Clients zur Verfügung gestellt werden, in einem Java-Interface (*Plain Old Java Interface, POJI*) angegeben werden. *POJI*

Die nächsten beiden Listings zeigen den Quellcode für ein einfaches lokal sichtbares Stateless Session Bean.

Listing 3.1
Beispiel für ein Stateless Session Bean ...

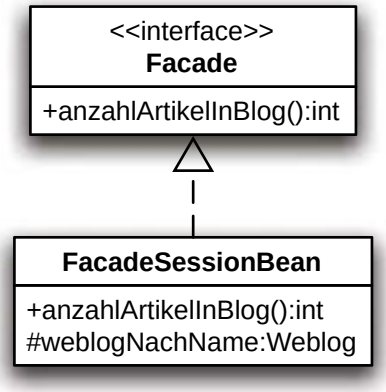
```
package de.bsd.weblog.ejb;

import javax.ejb.Stateless;

import de.bsd.weblog.interfaces.Facade;
import de.bsd.weblog.persistence.Weblog;

@Stateless
public class FacadeSessionBean implements Facade
{
  public int anzahlArtikelInBlog(String blog)
  { // ... }

  protected Weblog weblogNachName(String weblog)
  { // ... }
}
```



Listing 3.2
... und das zugehörige Business Interface

```
package de.bsd.weblog.interfaces;

@Remote
public interface Facade {
  public int anzahlArtikelInBlog(String blog);
  // ...
}
```

Man sieht, dass keine Callbacks wie `ejbCreate()` mehr implementiert werden müssen. Die Einstellung, dass es sich um ein *Stateless* Session Bean handelt, wird über die Annotation `@Stateless` angedeutet, und die lokale Sichtbarkeit ist einfach die Voreinstellung. Letztere beide Ein-

stellungen müssen in EJB 2.x über zusätzliche Einträge in *ejb-jar.xml* eingestellt werden.

3.1.1 Stateless Session Beans

Stateless Session Beans dienen insbesondere als Klammer für weitere Aktionen auf dem Server. Diese Aktionen können einfache Berechnungen oder auch Aufrufe weiterer EJBs sein. Stateless Session Beans können, wie der Name schon sagt, keinen Zustand zwischen zwei Client-Aufrufen halten. Dies bedeutet nicht, dass sie keine klassenweiten Felder haben können, sondern dass ein Client sich nicht darauf verlassen kann, dass er bei jedem Aufruf einer Methode des Session Beans dieselbe Instanz wie beim vorherigen Aufruf erhält. Stateless Session Beans werden in der Literatur oft durch *SLSB* abgekürzt.

SLSB

Stateless Session Beans werden durch die Annotation `@Stateless` an der Implementierungsklasse gekennzeichnet. Diese Annotation kennt drei Parameter, die alle optional sind:

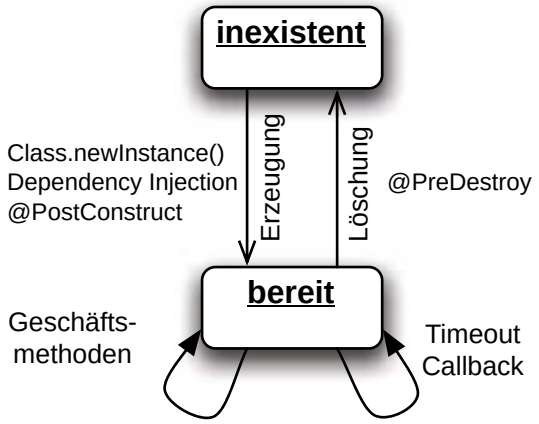
@Stateless

- name: Hiermit kann der Name des Session Beans explizit angegeben werden. Ist dieser Parameter nicht vorhanden, wird der unqualifizierte Name der Bean-Klasse verwendet. Unter diesem Namen ist das Session Bean dann in einem eventuell vorhandenen Deployment-Deskriptor referenzierbar, weswegen der Name immer innerhalb eines EJB-Archivs eindeutig sein muss.
- description: Eine Beschreibung für das Bean, die in externen Werkzeugen angezeigt werden kann.
- mappedName: Über dieses Attribut kann der Name des Session Beans auf einen anderen implementierungsspezifischen (JNDI-) Namen gemappt werden. Da Applikationsserver dieses Mapping nicht unterstützen müssen, ist dieses Attribut nicht portabel.

Lebenszyklus von Stateless Session Beans

Der Lebenszyklus von Stateless Session Beans ist relativ einfach und besteht nur aus zwei Zuständen.

Abbildung 3.1
Lebenszyklus von Stateless Session Beans



Im Zustand *inexistent* existiert das Bean noch nicht. Um in den Zustand *bereit* zu wechseln, instanziiert der Container das Bean mittels des Default-Konstruktors über den Aufruf von `Class.newInstance()`.

Nach der Instanziierung erfolgt die Dependency Injection, bei der der neuen Session-Bean-Instanz die Objekte zugewiesen werden, zu denen es Referenzen unterhält (siehe Abschnitt 6.1 ab Seite 153).

Stateless Session Beans können eine Methode haben, die vom Container nach der Instanziierung und Dependency Injection aufgerufen wird. Dies entspricht der `ejbCreate()`-Methode in EJB 2.x. In EJB 3.0 ist es möglich, eine beliebige Methode über `@PostConstruct` zu kennzeichnen. Ist eine *ejbCreate()*-Methode vorhanden, darf nur diese über `@PostConstruct` markiert werden. Die Callback-Methode wird spätestens dann aufgerufen, wenn ein Client das Session Bean anfordert, jedoch immer nachdem die Dependency Injection durchgeführt wurde.

In der Praxis wird es oft so sein, dass der Container bereits zur Deploy-Zeit eine Reihe von Beans erzeugt, diese in einen *Pool* legt und die *create*-Methode dabei aufruft. Das bedeutet, dass nicht immer eine direkte Kopplung zwischen dem initialen Aufruf eines SLSB durch einen Client und der Ausführung der durch `@PostConstruct` markierten Methode besteht. Ein aufrufender Client wird im Normalfall eine Instanz aus dem Pool erhalten, sofern noch eine zur Verfügung steht. Nur wenn alle Instanzen aus dem Pool in Benutzung sind, wird der Container neue Instanzen erzeugen und damit auch die mit `@PostConstruct` markierte Methode der neu erzeugten Instanz aufrufen.

Die mit `@PostConstruct` markierte Methode muss die Signatur

```
void methodenName()
```

haben und darf nur einmal pro Bean-Klasse vorkommen.
Der in Abbildung 3.1 gezeigte *Timeout Callback* bezieht sich auf den EJB-Timerservice, der weiter unten und in Abschnitt 3.5 beschrieben wird.

Löschen von SLSB

Ein SLSB kann, wie auch schon in der Vorgängerversion, eine *remove*-Methode haben, die aufgerufen wird, wenn der Container das Bean löscht (üblicherweise wenn die Applikation heruntergefahren wird, oder wenn der Container einen vorhandenen Pool wieder verkleinert, nachdem überzählige Instanzen nicht mehr benötigt werden). Diese Methode kann einen beliebigen Namen haben und wird mit `@PreDestroy` markiert. Ist eine Methode mit dem Namen `ejbRemove()` vorhanden, darf nur diese mit `@PreDestroy` versehen werden.

Ein Aufruf dieser *remove*-Methode durch einen Client führt üblicherweise nicht zur Löschung der Bean-Instanz, sondern ist lediglich ein No-op.

Callback für EJB Timer

Im Gegensatz zu den unten vorgestellten Stateful Session Beans können SLSB auch Methoden als Callback für den EJB Timer Service (siehe Abschnitt 3.5) zur Verfügung stellen. Diese werden dann bei Ablauf des Timers aufgerufen. Aufzurufende Methoden werden via `@Timeout` markiert, müssen `void` als Rückgabewert haben und einen Parameter von Typ `Timer` besitzen. Dies wird aber unten nochmals genauer beschrieben.

3.1.2 Stateful Session Beans

Im Gegensatz zu den gerade beschriebenen SLSB können Stateful Session Beans (*SFSB*) den Zustand zwischen Client-Aufrufen halten, da eine SFSB-Instanz einem Client direkt zugeordnet wird und so lange existiert, bis der Client diese Instanz explizit löscht (oder sie bei zu langer Inaktivität vom Container gelöscht wird).

@Stateful

Stateful Session Beans werden durch die Annotation `@Stateful` auf Klassenebene gekennzeichnet. Auch diese Annotation kennt dieselben drei Parameter, wie sie oben für `@Stateless` bereits beschrieben wurden.

Lebenszyklus von Stateful Session Beans

Der Lebenszyklus eines Stateful Session Beans ist etwas umfangreicher als der des zustandslosen Pendants, da SFSB auch auf Sekundärspeicher ausgelagert (passiviert) werden können.

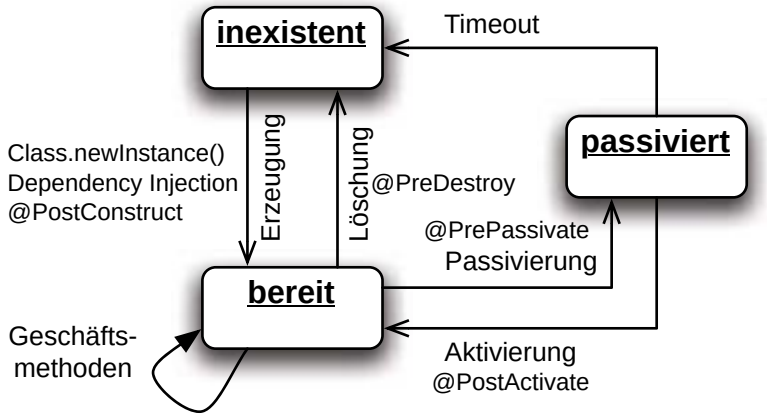


Abbildung 3.2 *Zustandsübergänge bei Stateful Session Beans*

Der Übergang zwischen *inexistent* und *bereit* erfolgt wie bei den Stateful Session Beans und wurde bereits auf Seite 30 beschrieben.

Aktivierung und Passivierung

Nachdem bei vielen gleichzeitigen Clients sich auch viele SFSB im Speicher befinden, kann der Container unbenutzte SFSB (also solche, auf die gerade kein Client zugreift) auf Sekundärspeicher (z.B. die Festplatte) auslagern, um benötigte Ressourcen wieder freizugeben.

Dieser Vorgang wird *Passivierung* genannt. Bevor die Passivierung durchgeführt wird, kann der Container eine Methode aufrufen, die mittels `@PrePassivate` gekennzeichnet wurde. In dieser Methode müssen nicht-serialisierbare Ressourcen oder solche, bei denen nicht garantiert werden kann, dass sie bei der Aktivierung wieder zur Verfügung stehen (wie z.B. Datenbankverbindungen), freigegeben werden.

@PrePassivate

Lagert der Container das Bean wieder ein (also die *Aktivierung*), wird eine durch `@PostActivate` markierte Methode aufgerufen, in der diese Ressourcen wieder geöffnet werden können. Ein erweiterter Persistenzkontext überlebt die Passivierung und Reaktivierung. Dieser Persistenzkontext wird ab Seite 116 beschrieben.

@PostActivate

In der Praxis wird man oftmals dieselben Initialisierungen nach der Instanziierung eines SFSB und nach der Aktivierung durchführen (z.B. Datenbankverbindungen öffnen). Hier lässt sich eine Methode auch mit

den beiden entsprechenden Annotationen versehen, so dass die Methode für beide Zwecke genutzt wird:

Listing 3.3 Mehrfachnutzung einer Callback-Methode

```
...
@PostConstruct
@PostActivate
public void initDbConn()
{
  Connection x = ...
}
```

Löschen von SFSB

Durch die Zuordnung eines SFSB zu einem Client muss dieser Client das Bean nach Ende der Benutzung wieder explizit löschen, damit der Container in der Lage ist, die verwendeten Ressourcen wieder freizugeben. Dies geschieht über den Aufruf einer durch `@Remove` gekennzeichneten Methode. Diese Annotation kennt nur einen Boolean-Parameter *retainIfException*, der angibt, ob das Löschen des SFSB fortgeführt werden soll, wenn die remove-Methode eine Exception wirft. Die Voreinstellung ist `false`, so dass das Bean trotzdem gelöscht wird.

@Remove

Beim Löschen des Beans wird vom Container eine mit `@PreDestroy` markierte Methode aufgerufen. Dieser Aufruf kann allerdings nicht in jedem Fall garantiert werden. Wenn das Bean beispielsweise passiviert wurde, wie im oberen Abschnitt beschrieben, kann der Container das passivierte Bean auch direkt löschen und dies nicht erst aktivieren, um es dann zu löschen. Dies ist in Abbildung 3.2 als *Timeout* markiert.

Create-Methode für EJB 2.x Clients

EJB 3.0 Clients können die Geschäftsmethoden des EJB 3.0 Session Beans direkt aufrufen, ohne dass vorher eine *create*-Methode im Home Interface aufgerufen werden müsste (letzteres ist ja weggefallen).

Soll ein EJB 3.0 Session Bean von EJB 2.x Clients aufgerufen werden, wird ein (Local)Home Interface benötigt. Dies wird in Abschnitt 3.1.5 beschrieben.

Stateful Session Beans können dabei in diesem EJB 2.x Home Interface mehrere Methoden für das Erzeugen des Beans haben. Um die Schnittstellen für EJB 3.0 SFSBs bereitzustellen, können diese Methoden mit `@Init` gekennzeichnet werden. Dabei ist zu beachten, dass der Rückgabetyp `void` ist und dass die Methodenparameter exakt mit denen der entsprechenden `create`*`MNAME`*`()`-Methode übereinstimmen. Ist nur eine solche Create-Methode im Home Interface vorhanden, benö-

@Init

tigt `@Init` keine weiteren Parameter. Ansonsten muss über den *value* der Name der entsprechenden Create-Methode angegeben werden.

Listing 3.4 *Verwendung von @Init ...*

```
/**
 * Das EJB 3.0 Session Bean, das von EJB-2.x-Clients
 * aus sichtbar sein soll
 */
@Stateful
@RemoteHome(ShoppingCartHome.class)
public class ShoppingCartBean
{
   @Init(value="createEmptyCart")
   public void neuerEinkaufswagen()
   {
     ...
   }

   @Init(value="createCartWithArticle")
   public void neuerEinkaufswagen(Article artikel)
   {
     ...
   }

   public void update() { ... }
}
```

Hier wurden also zwei create-Methoden mit unterschiedlicher Signatur definiert, die dann im Remote Home Interface gelistet werden. Dies wird unten gezeigt. Die Klasse des Remote Home Interfaces wird über `@RemoteHome` angezeigt. Diese Annotation ist in Abschnitt 3.1.5 weiter unten beschrieben.

Listing 3.5 *... das zugehörige Remote Home Interface ...*

```
/**
 * Remote Home des Session Beans
 */
public interface ShoppingCartHome extends EJBHome
{
  public ShoppingCart createEmptyCart()
     throws RemoteException, CreateException;

  public ShoppingCart createCartWithArticle(Article a)
     throws RemoteException, CreateException;
}
```

Zusätzlich zum Remote Home Interface wird auch noch das Remote-Objekt benötigt, über das die Geschäftsmethoden des EJB 3.0 Session Beans angesprochen werden können.

```
/**
 * Remote-Objekt des Session Beans
 */
public interface ShoppingCart extends EJBObject
{
  public update() throws RemoteException;
}
```

Listing 3.6
... und das Remote-Objekt

In Stateless Session Beans ist diese Auszeichnung der create-Methoden nicht notwendig, da es hier in EJB 2.x nur eine einzige Methode im Home Interface gibt, deren Aufruf einfach einen Verweis auf die Geschäftsschnittstelle zurückliefert.

3.1.3 Geschäftsschnittstellen

In EJB 3.0 ist die Notwendigkeit für Home Interfaces weggefallen. Clients können Session Beans nun direkt verwenden, ohne die *create()*-Methode aus dem Home Interface vorher aufzurufen. Aufrufbare Operationen sind dabei in der Geschäftsschnittstelle des Beans hinterlegt.

Diese kann nun, wie bereits erwähnt, ein einfaches Java-Interface sein. Je nachdem, ob die Schnittstelle lokal oder remote zur Verfügung stehen soll, wird dies über eine `@Local`-oder `@Remote`-Annotation angezeigt.

@Local
@Remote

Diese Annotationen können dabei sowohl an an der Implementierungsklasse als auch an der entsprechenden Schnittstellenklasse angebracht werden. Werden Annotationen sowohl an der Schnittstelle als auch an der Implementierungsklasse angebracht, müssen diese in beiden Fällen dieselbe Annotation sein.

Annotation an der Bean-Klasse

Wird die Annotation an der Bean-Klasse angebracht, kann über den Parameter *value* ein Array von `Class`-Objekten übergeben werden welche die entsprechenden Geschäftsschnittstellen darstellen. In diesem Fall muss die Bean-Klasse diese Schnittstellen nicht direkt implementieren.

```
public interface SomeInterface
{
  public void sayHello();
}

@Stateless
@Remote(SomeInterface.class)
```

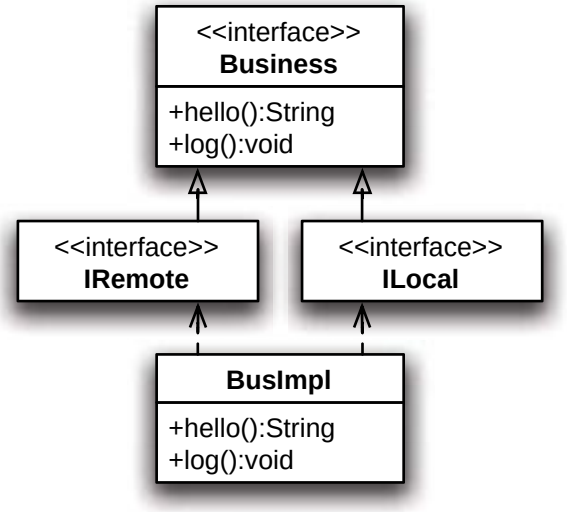
Listing 3.7
Session Bean ohne direkte Implementierung der Geschäftsschnittstelle

```
public class SomeBean
{
  public void sayHello() { ... }
}
```

Sichtbarkeit Local und Remote

Sollen beide Sichtbarkeiten für dieselbe Geschäftsschnittstelle verfügbar sein, kann man das Entwurfsmuster aus Abbildung 3.3 anwenden, da eine Schnittstelle nicht beide Sichtbarkeiten auf einmal implementieren kann.

Abbildung 3.3
Pattern zur Angabe der Komponentenschnittstellen



Die beiden in der Abbildung als *IRemote* und *ILocal* bezeichneten Interfaces sind dabei nur als Marker zu sehen, die außer der jeweiligen Annotation (`@Remote` bzw. `@Local`) leer sind und die gemeinsame Schnittstelle *Business* beerben. Die Klasse *BusinessBean* ist dabei die eigentliche Bean-Klasse.

Alternativ hierzu ist es auch möglich, die beiden Schnittstellen als Parameter der `@Local` und `@Remote` entsprechend anzugeben:

Listing 3.8
Angabe der Local und Remote Interfaces über Annotationen

```
import javax.ejb.Local;
import javax.ejb.Remote;
import javax.ejb.Stateless;

@Remote(RemoteBusinessInterface.class)
@Local(LocalBusinessInterface.class)
@Stateless
```

```
public class MyStatelessSessionBean
{
  ...
}
```

Standardsichtbarkeit: Local

Die Annotation `@Local` kann für ein nur lokal sichtbares Session Bean weggelassen werden, da Session Beans standardmäßig als Local gelten. Die Annotation wird nur dann benötigt, wenn mehr als eine Schnittstelle implementiert wird und der Container damit nicht in der Lage ist, das Business Interface zu ermitteln. Ausgenommen von dieser Regel sind die Schnittstellen `java.io.Serializable` und `java.io.Externalizable`, sowie die Schnittstellen aus dem Paket `javax.ejb`.

```
...
@Stateless
@Local(IFoo.class)
public class Foo implements IFoo, IBar, IBaz {
...
```

Allerdings muss die Implementierungsklasse in jedem Fall ein Business Interface implementieren. Eine alleinstehende Klasse reicht für ein Session Bean nicht aus.

Aufrufsemantik der Schnittstellen

Die Semantik der Parameterübergabe ist bei den beiden Client-Schnittstellen verschieden: Bei einem *Local* Interface werden Parameter als Referenz übergeben (*call by reference*). Hier wird also nur ein Zeiger auf die als Parameter genutzte Variable an die aufgerufene Methode übergeben. Dies ist möglich, weil die Methoden immer in derselben JVM laufen. Eine Modifikation eines übergebenen Parameters in der aufgerufenen Methode ist also auch in der aufrufenden Methode sichtbar.

Bei *Remote*-Schnittstellen werden die Parameter für den Aufruf kopiert (*call by value*). Eine Änderung eines übergebenen Parameters in der aufgerufenen Methode wirkt sich also nicht auf den Aufrufer aus.

3.1.4 Session Beans und Webservices

Session Beans können wie bereits in J2EE 1.4 sowohl einen Webservice exportieren als auch Client entfernter Webservices sein. Dies wird in Kapitel 7 ab Seite 165 genauer beschrieben. Deshalb hier nur ein kurzer Abriss dazu.

Export als Webservice

Methoden von Stateless Session Beans können wie auch schon in J2EE 1.4 als Webservice exportiert werden. Dieser Export hat zwei große Änderungen erfahren:

- Durch die Annotationen von JSR-181 wurde die Entwicklung deutlich vereinfacht.
- Es muss kein Service Endpoint Interface (SEI) mehr definiert werden. Geschieht dies trotzdem, werden alle dort gelisteten Methoden als Webservice exportiert, ungeachtet weiterer `@WebMethod`-Annotationen.

Prinzipiell ist es aber nun ganz einfach, ein Stateless Session Bean als Webservice zu exportieren. Das Bean wird via `@WebService` als Webservice markiert. Dadurch werden alle öffentlichen Methoden des Beans als Webservice-Operation exportiert. Alternativ können auch nur die Methoden mittels `@WebMethod` markiert werden, die als Webservice-Operation aufrufbar sein sollen. Mehr hierzu gibt es in Kapitel 7.

Als Webservice Clients

Session Beans sind ebenfalls seit J2EE 1.4 in der Lage, entfernte Webservices zu nutzen. Diese entfernten Webservices können dem Session Bean über die `@WebServiceRef`-Annotation bekannt gemacht werden. Die Verwendung ist nicht ganz so transparent wie bei EJB-Referenzen, die über `@EJB` injiziert werden. Es ist ebenfalls in Kapitel 7 beschrieben.

3.1.5 Home-Schnittstellen für EJB 2.x Clients

In EJB 2.x ist für die Ansprache eines Session oder Entity Beans eine Home-Schnittstelle notwendig. Diese ist in EJB 3.0 weggefallen. Entity Beans können sogar gar nicht mehr von Remote angesprochen werden (mehr dazu in Kapitel 4). Soll ein EJB 3.0 Session Bean von einem EJB 2.x Client angesprochen werden, muss für diesen Client eine Home-Schnittstelle bereitgestellt werden.

@RemoteHome
@LocalHome

Diese Schnittstellen können an einem Session Bean über die Annotationen `@RemoteHome` oder `@LocalHome` angegeben werden. Die Annotationen kennen beide den Parameter *value*, der die Klasse der entsprechenden Schnittstelle angibt.

Listing 3.9 *Definition einer EJB-2.1-LocalHome-Schnittstelle*

```
public interface MyHome extends javax.ejb.EJBLocalHome
{
  ... // wie in EJB-2.x
}
```

```
@Stateless
@LocalHome(MyHome.class)
public class MyBean implements MyBeanIF
{
   ...
}
```

Listing 3.4 auf Seite 34 zeigt ein weiteres Beispiel für die Verwendung.

3.1.6 Aufrufe aus EJB 3.0 Clients heraus

Clients wie Servlets oder Java EE Application Clients können Session Beans in EJB 3.0 »einfach nutzen«, indem sie sich via Dependency Injection eine Referenz vom Client Container geben lassen. Dabei ist es unerheblich, ob der Client ein anderes Session Bean, ein Servlet oder gar ein Application Client in einer anderen JVM ist. Im Java Application Client müssen die Felder, in die injiziert wird, `static` sein, was in Servlets oder EJBs nicht der Fall sein darf. Siehe Abschnitt 6.3.

Listing 3.10
Aufruf eines Session Beans aus einem Servlet heraus

```
...
public class MyClient extends HttpServlet
{
  @EJB
  MyEjb ejb; // MyEJB wird injiziert

  public void init()
  {
    ejb.foo(); // Aufruf ohne vorheriges create
  }
}
```

Der Client Container stellt dabei sicher, dass die Variable *ejb* vor ihrer ersten Verwendung entsprechend belegt ist. Dafür kann es natürlich weiterhin notwendig sein, die Adresse des JNDI-Servers in der Datei *jndi.properties* dem Client bekannt zu geben.

Clients, die nicht in die oben genannten Kategorien fallen, können weiterhin wie gewohnt Ressourcen im JNDI des Server suchen.

3.1.7 Unterschiede zu Session Beans in EJB 2.x

Dieser Abschnitt listet noch einmal die Unterschiede der Session Beans zwischen den EJB-Versionen 2.x und 3.0 auf.

- Wegfall der Deployment-Deskriptoren: Annotationen im Quellcode ersetzen die Deployment-Deskriptoren. Letztere können

aber weiterhin genutzt werden, um Einstellungen einzelner Annotationen zu überschreiben.

- Wegfall der Home Interfaces. EJB 3.0 Clients können direkt die gewünschten Geschäftsmethoden aufrufen.
- Es ist nicht mehr unbedingt notwendig, die Java-Schnittstelle `javax.ejb.SessionBean` und damit die Callback Interfaces `ejb*()` zu implementieren.
- Wegfall von Methoden, die nichts bewirken (`create()` und `remove()` sind bei Stateless Session Beans ein No-op).
- Es gibt nun die Möglichkeit, Interceptoren für Aufrufe von Geschäftsmethoden zu definieren. Interceptoren werden in Abschnitt 3.4 ab Seite 46 beschrieben.
- Die Geschäftsschnittstelle eines Session Beans ist nun ein Plain Old Java Interface. Es muss weder EJBHome noch EJBObject implementiert werden.

3.2 Message-Driven Beans

Message-Driven Beans (MDB) werden im Gegensatz zu Session Beans nicht direkt von einem Client aufgerufen. Aufrufe erfolgen nach einem asynchronen Ereignis durch den Container. Dabei wird immer eine vorher festgelegte Methode (meist `onMessage()`) aufgerufen, welche die eigentliche Arbeit verrichtet.

Auch die Message-Driven Beans sind mittlerweile nur noch POJOs. Im Gegensatz zu den Session Beans müssen sie allerdings weiterhin ein technisches Interface, `MessageListener`, beispielsweise aus dem Paket `javax.jms`, implementieren (direkt über eine `implements`-Beziehung oder durch Angabe in der Annotation `@MessageDriven`), können dafür aber auch auf die Angabe eines Business Interfaces verzichten, da sie ja nie direkt von einem Client aufgerufen werden.

Die Tatsache, dass es sich um ein Message-Driven Bean handelt, wird über die Annotation `@MessageDriven` dem Laufzeitsystem mitgeteilt.

Listing 3.11 *Beispiel für ein Message-Driven Bean*

```
import javax.ejb.ActivationConfigProperty;
import javax.ejb.MessageDriven;
import javax.jms.Message;
import javax.jms.MessageListener;
```

```
@MessageDriven(activationConfig=
{
  @ActivationConfigProperty(
    propertyName="destinationType",
    propertyValue="javax.jms.Topic"),
  @ActivationConfigProperty(
    propertyName="destination",
    propertyValue="queue/aTopic"),
  @ActivationConfigProperty(
    propertyName="subscriptionDurability",
    propertyValue="Durable")
})
public class MyMDBean implements MessageListener
{
  ...
  public void onMessage(Message msg)
  {
   ...
  }
}
```

@MessageDriven

Die Parameter von `@MessageDriven` bestimmen unter anderem, auf welchen Kanälen ein MDB auf Events wartet und wie mit empfangenen Nachrichten verfahren werden soll. Die Parameter sind im Einzelnen:

- name: Gibt den Namen des MDBs an. Ist er nicht vorhanden, wird der unqualifizierte Klassenname verwendet. Der Name muss innerhalb eines ejb-jars eindeutig sein.
- description: Eine Kurzbeschreibung dieses Message-Driven Beans.
- messageListenerInterface: Gibt die Klasse des implementierten MessageListener (JMS, JCA-inbound etc.) an. Diese Klasse muss angegeben werden, wenn die Bean-Klasse entweder keine MessageListener-Schnittstelle implementiert oder mehr als eine Java-Schnittstelle implementiert. Die Marker-Schnittstellen `java.io.Serializable` und `java.io.Externalizable` fallen nicht unter diese Zählung, genauso wie Schnittstellen aus dem Paket `javax.ejb`.
- activationConfig: Dieser Parameter definiert Einstellungen, die für die Aktivierung des Message-Driven Beans zuständig sind, und spezifiziert ein Array von `@ActivationConfigProperty`-Elementen. Diese werden gleich noch näher beschrieben.
- mappedName: Dieser String-Parameter gibt einen herstellerspezifischen Namen an, auf den das MDB gemappt werden soll. Dies kann zum Beispiel der globale JNDI-Name einer Queue oder ei-

nes Topics sein. Da Server diesen Parameter nicht unterstützen müssen, ist seine Verwendung nicht portabel.

@ActivationConfigProperty

Die Parameter der `@ActivationConfigProperty`-Elemente bestehen aus Paaren von String-Werten, *propertyName* und *propertyValue*, und spezifizieren, wie sich das MDB bei eingehenden Nachrichten verhalten soll.

Wird das Message-Driven Bean über JMS bedient und implementiert die Schnittstelle `javax.jms.MessageListener`, sind mögliche Werte für den *propertyName* nachfolgend gelistet. Werte mit einen (*) sind Voreinstellungen, falls die entsprechende Eigenschaft nicht angegeben ist.

- acknowledgeMode: Bei Verwendung von Container-Managed Transactions (CMT) bestätigt der Container eingegangene und zugestellte Nachrichten selbständig beim *commit* der Transaktion. Bei Bean-Managed Transactions (BMT) müsste dies eigentlich das MDB selbst erledigen.
  Da dies aber nicht innerhalb der Transaktion möglich ist, erledigt dies der Container. Der Bestätigungsmodus im Fall von Bean-Managed Transactions kann dabei diese beiden Werte annehmen:
  - Auto-acknowledge(*): Nachrichten werden vom Container bestätigt. Es ist sichergestellt, dass eine Nachricht genau einmal (erfolgreich) zugestellt wird.
  - Dups-ok-acknowledge: Nachrichten werden vom Container bestätigt. Dabei wird kein so hoher Aufwand getrieben, um Duplikate in der Zustellung zu vermeiden. Diese Duplikate können zum Beispiel bei einem Crash des Containers auftreten. Das MDB muss selbst auf diese Duplikate reagieren.
- destination: Der JNDI-Name der Destination (Queue, Topic), von der Nachrichten empfangen werden sollen
- destinationType: Der Typ der Destination. Für JMS sind die möglichen Werte `javax.jms.Queue` und `javax.jms.Topic`.
- messageSelector: Ein Ausdruck, über den sich JMS-Nachrichten filtern lassen. Dabei werden Felder im Nachrichtenkopf vom Container ausgewertet. Die Auswertung erfolgt dabei über den *messageSelector*. Dieser ist in einer Untermenge des SQL-92-Standards formuliert und wird in Kapitel 3.8 von [7] genau beschrieben. Ist das Ergebnis des Ausdrucks wahr, wird die Nachricht an das MDB weitergeleitet.
- subscriptionDurability: Diese Eigenschaft legt fest, was mit Nachrichten geschehen soll, die eintreffen, wenn das MDB nicht im Zustand *bereit* ist (s.u.).

- NonDurable(*): Ist das Message-Driven Bean nicht *bereit*, wird die Nachricht nicht zugestellt und auch nicht für eine spätere Zustellung gespeichert.
- Durable: Der JMS-Provider speichert Nachrichten an das MDB zwischen und stellt sie zu, wenn das Bean *bereit* ist.

Der Lebenszyklus von Message-Driven Beans wird im nächsten Abschnitt beschrieben.

Für andere Arten von Message Listenern gibt es andere Propertys, auf die hier nicht näher eingegangen wird.

3.2.1 Lebenszyklus von Message-Driven Beans

Der Lebenszyklus eines Message-Driven Beans ähnelt stark dem eines Stateless Session Beans.

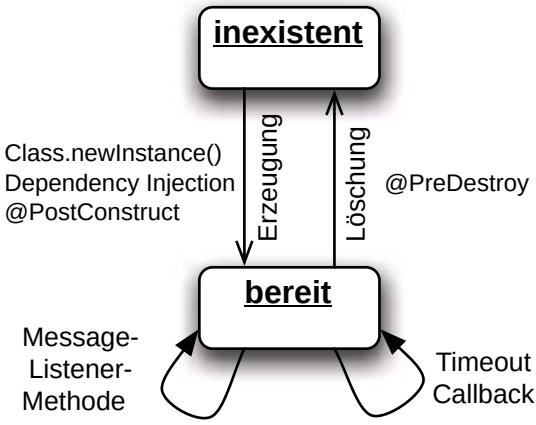


Abbildung 3.4
Lebenszyklus von Message-Driven Beans

Da der Container das Message-Driven Bean über den Aufruf der Methode `Class.newInstance()` instanziiert, muss das Bean einen öffentlichen Default-Konstruktor besitzen. Nach der Instanziierung erfolgt die Dependency Injection, bei der der neu erzeugten Instanz die Objekte zugewiesen werden, zu denen sie Referenzen unterhält. Danach folgt der Aufruf von durch `@PostConstruct` markierten Methoden. Wenn dies geschehen ist, geht das MDB in den Zustand *bereit* und kann Nachrichten empfangen. Diese werden an die Message-Listener-Methode, wie zum Beispiel `onMessage()`, vom Container zugestellt.

Die gezeigten *Timeout* Callbacks kommen vom EJB Timer Service, der in Abschnitt 3.5 näher beschrieben wird.

3.2.2 Callbacks für MDBs

Message-Driven Beans kennen drei Callbacks, die vom Container aufgerufen werden können.

@PostConstruct

- `@PostConstruct`: Eine so markierte Methode wird nach der Instanziierung der Bean-Klasse und dem Injizieren von Ressourcen aufgerufen. Diese Methode entspricht den *ejbCreate()*-Methoden in EJB 2.x. Üblicherweise wird ein Bean unabhängig von eingehenden Nachrichten instanziiert, so dass hier kein direkter Zusammenhang zu einer von einem Client versandten Nachricht zu sehen ist.

@PreDestroy

- `@PreDestroy`: Bevor der Container das MDB zerstört, wird dieser Callback aufgerufen. Dieser Aufruf ist allerdings nicht garantiert und kann zum Beispiel dann nicht erfolgen, wenn der Container crasht oder wenn eine System-Exception geworfen wird. System-Exceptions werden in Abschnitt 3.6 ab Seite 57 beschrieben.
- `@Timeout`: Hiermit kann eine Methode markiert werden, die aufgerufen wird, wenn ein EJB-Timer abgelaufen ist. Für mehr Information siehe Abschnitt 3.5.

3.2.3 Unterschiede zu Message-Driven Beans in EJB 2.x

Dieser Abschnitt fasst nochmals die Unterschiede bei den Message-Driven Beans zwischen EJB 3.0 und EJB 2.x zusammen.

- Die Schnittstelle `javax.ejb.MessageDrivenBean` muss nicht mehr implementiert werden.
- Message-Driven Beans werden über die Annotation `@MessageDriven` gekennzeichnet.
- Eigenschaften über die Aktivierung des MDBs durch eingehende Nachrichten können im Feld `activationConfig` der Annotation `@MessageDriven` angegeben werden.
- Das Message-Driven Bean muss weiterhin einen Message Listener implementieren. Dies kann über eine `implements`-Beziehung geschehen oder alternativ durch Angabe der implementierten Schnittstelle innerhalb des Parameters `activationConfig` der Annotation `@MessageDriven`.

3.3 Der EJB-Kontext

Wie in EJB 2.x verfügen Session Beans und Message-Driven Beans in EJB 3.0 ebenfalls über einen EJB-Kontex, der in den Klassen `EJBContext` beziehungsweise `SessionContext` und `MessageDrivenContext` zur Verfügung steht. Dieser Kontext kann nun ebenfalls injiziert werden.

Für EJB 2.1 Entity Beans gibt es auch weiterhin einen EntityContext. Dieser ist allerdings für EJB 3.0 Entity Beans nicht mehr vorhanden.

Listing 3.12 *Injektion des SessionContexts*

```
import javax.annotation.Resource;
import javax.ejb.SessionContext;
import javax.ejb.Stateful;

@Stateful
public class MySessionBean implements SomeIf
{
  @Resource SessionContext ctx;

  public void machWas()
  {
    // ...
    ctx.setRollbackOnly();
  }
```

Zusätzlich zu den aus EJB 2.x bekannten Methoden des EJB-Kontexts sind einige neue Methoden hinzugekommen:

- `lookup()`: Diese Methode dient als Shortcut für das Aufsuchen von Objekten im Enterprise Naming Context (ENC). Dieser ENC ist im JNDI unter `java:/comp/env` zu finden; die Verwendung wird in Abschnitt 6.2.4 ab Seite 159 beschrieben. Einträge im ENC können auch via `@Resource` injiziert werden. Dies ist in Abschnitt 6.2.5 beschrieben. Die Methode hat die Signatur

  ```
  Object lookup(String name)
  ```

  Der übergebene *name* wird dabei relativ zum ENC gesucht.

  ```
  public machWasAnderes()
  {
    Object o = ctx.lookup("ejb/SomeSessionBean");
  }
  ```

Diese Suche über den SessionContext ist also äquivalent zu einer Suche nach *java:comp/env/ejb/SomeSessionBean* innerhalb eines (Initial)Contexts im JNDI.

- `getBusinessObject()` (nur SessionContext): Mit dieser Methode kann eine Referenz auf das gerade ausgeführte EJB erfragt werden. Der Rückgabewert ist quasi das Äquivalent zum `this`-Pointer aus Java SE. Die Signatur der Methode ist

```
<T> getBusinessObject(Class<T> bif) throws
        IllegalStateException
```

  Das übergebene *bif* beschreibt dabei eine der Geschäftsschnittstellen des Session Beans, die dann an ein anderes Bean weitergereicht werden kann. Wird der Methode eine für das aufrufende Session Bean ungültige Geschäftsschnittstelle übergeben, wird eine `IllegalStateException` geworfen.

- `getInvokedBusinessInterface()` (nur SessionContext): Diese Methode liefert die Geschäftsschnittstelle zurück, durch die das Session Bean aufgerufen wurde. Die Signatur ist ganz einfach

```
Class getInvokedBusinessInterface()
```

  Über diese Methode lässt sich dann herausfinden, ob das Session Bean über seine Local-, Remote- oder Webservice-Schnittstelle aufgerufen wurde.

3.4 Interceptoren

Ein komplett neues Feature von EJB 3.0 sind Interceptoren. Diese Interceptoren können genutzt werden, um Querschnittsaspekte von Programmen aus dem eigentlichen Code herauszuziehen und »zentral« für die Verwendung bereitzustellen. Die generelle Funktionsweise wurde bereits in Abschnitt 1.3 gezeigt.

Interceptoren können für Session Beans und Message-Driven Beans angewandt werden.

Die Java-EE-Spezifikation spricht dabei von zwei Arten von Interceptoren:

- Interceptoren für Geschäftsmethoden. Diese dienen, wie beschrieben, dazu, Aufrufe von Methoden abzufangen, um beispielsweise Parameter zu loggen.
- Lebenszyklus-Callbacks. Diese Interceptoren beschreiben die neue Art und Weise, um Methoden für die Callbacks des Con-

tainers (also das Äquivalent zu `ejbCreate()` etc.) bereitzustellen. Einige dieser Callbacks wurden bereits bei den Session Beans und Message-Driven Beans vorgestellt.

Beginnen wir mit der Betrachtung der Interceptoren für Geschäftsmethoden.

3.4.1 Interceptoren für Geschäftsmethoden

Abbildung 1.4 auf Seite 6 illustriert die Funktionsweise eines Interceptors. Der Entwickler kodiert seine Methodenaufrufe wie bisher. Zur Laufzeit ruft der Container den Interceptor auf, der Aktionen ausführen kann (*Before Advice*), über `proceed()` wird dann die eigentliche Zielmethode aufgerufen. Danach kehrt der Aufruf in den Interceptor zurück, wo weitere Aktionen ausgeführt werden können (*After Advice*). Es ist durchaus möglich, dass mehr als ein Interceptor für eine Methode konfiguriert ist. Dann werden zunächst die Before Advices der Reihe nach aufgerufen, dann die Ziel-Methode und dann in der umgekehrten Reihenfolge die After Advices. Im Gegensatz zu den von JBoss bekannten Interceptoren sind Interceptoren in EJB 3.0 nur für die Serverseite definiert.

Interceptoren sind zustandslos – sie können also keine Information zwischen zwei Methodenaufrufen austauschen. Allerdings ist es möglich, wenn mehr als ein Interceptor konfiguriert ist, Informationen zwischen diesen Interceptoren auszutauschen. Ein Interceptor darf aber durchaus lokale Variablen haben, um Zustandsinformationen vom *Before Advice* zum *AfterAdvice* transferieren zu können.

Interceptoren können dabei innerhalb der Bean-Klasse selbt oder in einer eigenen Klasse hinterlegt werden. Sind sie in der Bean-Klasse hinterlegt, reicht es, in dieser eine `@AroundInvoke`-Annotation an eine Methode anzubringen, um diese Methode als Interceptor zu kennzeichnen. Soll der Interceptor für mehrere Beans verwendet werden, kann er in einer eigenen Klasse abgelegt und via `@Interceptors`-Markierung an der Klasse oder an einzelnen Klassenmethoden verwendet werden. Für die Verwendung an einzelnen Methoden der Klasse muss der Interceptor immer in einer von der Bean-Klasse separaten Klasse liegen.

Listing 3.13
Beispiel für die Verwendung eines klassenweiten Interceptors aus einer Interceptorklasse

```
...
import javax.interceptor.Interceptors;

@Interceptors(TimingInterceptor.class)
@Stateless
@Interceptors(TimingInterceptor.class)
public class FacadeSessionBean implements Facade
```

```
{
  // ..
  public Collection<Weblog> listeAlleBlogs()
  { // ... }
```

@Interceptors

In Listing 3.13 wurde ein Interceptor *TimingInterceptor* über die `@Interceptors`-Annotation für die Klasse *FacadeSessionBean* deklariert. Der *TimingInterceptor* wird im nächsten Abschnitt beschrieben.

Wären innerhalb von `@Interceptors` mehr als ein Interceptor angegeben, wie beispielsweise

```
@Interceptors(TimingInterceptor.class, I2.class),
```

würde die Reihenfolge der Interceptoren in der Deklaration die Reihenfolge des Aufrufs angeben. Ein Aufruf der Methode *listeAlleBlogs()* würde also erst durch den Interceptor *TimingInterceptor* und dann durch *I2* geleitet werden, bevor *listeAlleBlogs()* aufgerufen wird.
Interceptoren können sowohl auf Klassen- als auch auf Methodenebene deklariert werden, indem `@Interceptors` am entsprechenden Element angebracht wird. Interceptoren auf Methodenebene werden dabei als Letzte in der Interceptorenkette angeordnet.

Ein einfacher Interceptor

Als Nächstes wollen wir uns nun anschauen, wie ein solcher Interceptor aussehen könnte. Hierzu nehmen wir das »HelloWorld« der Interceptoren, den Timing-Interceptor.

Listing 3.14 *Beispiel für einen EJB-3.0-Interceptor*

```
package de.bsd.weblog.interceptors;

import javax.interceptor.AroundInvoke;
import javax.interceptor.InvocationContext;

import org.apache.log4j.Logger;

/**
 * Ein EJB-3.0-Timing-Interceptor. Dieser misst
 * die Dauer der Aufrufe von EJB-Geschäftsmethoden.
 */
public class TimingInterceptor {

  Logger log = Logger.getLogger("TimingInterceptor");

  @AroundInvoke
  public Object timing(InvocationContext ctx)
    throws Exception
```

```
    {
      long t1 = System.currentTimeMillis();
      long now;
      String meth = "";
      try {
        // Aufgerufene Methode extrahieren
        meth = ctx.getMethod().getName();

        // Nächste Methode aufrufen
        return ctx.proceed();
      }
      finally {
        now = System.currentTimeMillis();
        if (log.isDebugEnabled()) {
          log.debug("Call to " + meth + " took "
            + (now-t1) + "ms");
        }
      }
    }
  }
```

@AroundInvoke

Über `@AroundInvoke` wird die Methode ausgezeichnet, welche die aktuelle Interceptor-Funktion implementiert. Diese darf natürlich keine Geschäftsmethode sein, da dies in einer Endlosrekursion enden würde. Die Interceptor-Methode bekommt ein `InvocationContext`-Objekt übergeben, aus dem die aufgerufene Methode, ihre Parameter oder der EJB-Kontext ausgelesen werden können. Schließlich enthält das Objekt die Methode `proceed()`, über die der nächste Interceptor oder die endgültige Geschäftsmethode aufgerufen wird. Die Interceptor-Methode an sich darf dabei weder `static` noch `final` sein.

Für Interceptor-Klassen gilt, dass sie einen `public` Default-Konstruktor besitzen müssen.

Möchte man einen Interceptor nur innerhalb einer Klasse nutzen, kann man auch eine Methode über `@AroundInvoke` kennzeichnen. Diese wird dann in die Liste der durch `@Interceptors` deklarierten Interceptoren an der letzten Stelle hinzugefügt (allerdings noch vor den Interceptoren, die direkt an einer Methode spezifiziert wurden).

Innerhalb der Interceptor-Methode können alle Arten von Java-Entitäten aufgerufen werden. Dies beinhaltet EJBs, JMS, JDBC oder auch JNDI. Der Aufruf erfolgt dabei mit den Security- und Transaktionskontexten der aufgerufenen Geschäftsmethode.

Technische Aspekte verstecken

Über Interceptoren lassen sich, wie bereits erwähnt, sehr schön technische Aspekte kapseln, wie etwa das Holen einer Datenbankverbindung vor dem Eintritt in eine Methode und das (sichere) Schließen der Verbindung etc. beim Austritt. Hierzu könnte man die Datenbankverbindung als Feld in der Klasse ablegen. Bei Aufruf einer Geschäftsmethode wird vorher der Interceptor aufgerufen, der das Feld füllt. Beim Verlassen der Geschäftsmethode kehrt der Aufruf in den Interceptor zurück, welcher dann die Verbindung wieder schließt.

3.4.2 Verwendung von Klassen-Interceptoren verhindern

@ExcludeClass-Interceptors

In manchen Fällen kann es notwendig sein, die Ausführung der klassenweiten Interceptoren für eine Geschäftsmethode zu unterbinden. Dies lässt sich über `@ExcludeClassInterceptors` an der entsprechenden Methode erreichen, wie dies in Listing 3.15 gezeigt wird.

Listing 3.15 *Verwendung von Interceptoren verhindern*

```
@Interceptors(MyInterceptors.class)
public class foo {

  // hier keine Klassen-Interceptoren
  @ExcludeClassInterceptors
  public void bar()
  { ...}

  // hier keine Klassen-Interceptoren, aber
  // die aus OtherInterceptors.class
  @ExcludeClassInterceptors
  @Interceptors(OtherInterceptors.class}
  public String toString()
  { ... }

  // Hier keine Default-Interceptoren, aber
  // die klassenweiten
  @ExcludeDefaultInterceptors
  public int getCount()
  { ... }
```

3.4.3 Default-Interceptoren

Über einen Eintrag in *ejb-jar.xml* können auch Interceptoren definiert werden, die für alle Session- und Message-Driven Beans innerhalb des jar-Archivs gelten, in dem *ejb-jar.xml* enthalten ist.

Diese werden vor allen anderen Interceptoren in der Aufrufkette verwendet. Über die `@ExcludeDefaultInterceptors`-Annotation kann (wie bei `@ExcludeClassInterceptors`) die Ausführung der Default-Interceptoren für die markierte Methode verhindert werden. Dies ist ebenfalls in Listing 3.15 im vorherigen Abschnitt gezeigt.

@ExcludeDefault-Interceptors

3.4.4 Interceptoren für Lebenszyklus-Callbacks

Lebenszyklusmethoden wie `@PostConstruct`, die für mehr als eine Klasse genutzt werden sollen, können in eine eigene Klasse ausgelagert werden, welche dann in der eigentlichen Bean-Klasse referenziert wird. In diesem Fall wird die Callback-Methode, die ja normalerweise keine Parameter übergeben bekommt, zu einem Interceptor. Dies sieht dann wie im Beispiel in Listing 3.16 aus.

Listing 3.16 *Container-Callbacks in eigener Klasse*

```
import javax.ejb.PostActivate;
import javax.interceptor.InvocationContext;

public class MyCallbacks {

  @PostActivate
  void intercept(InvocationContext ctx)
  {
    ...
    ctx.proceed();
  }
}
```

Diese Klasse wird dann wie gewohnt über `@Interceptors` referenziert, wie z.B. in Listing 3.13 gezeigt.

Die zu verwendenden Methoden bekommen also die Annotation des gewünschten Callbacks und müssen in der Form

`void`*`name`*`(InvocationContext)`

vorliegen. Sie dürfen dabei weder `final` noch `static` sein. Auch ist es nicht erlaubt, mehr als einen Callback eines Typs pro Klasse zu haben. Die Liste der Callbacks ist nochmals in Tabelle 3.1 zusammengefasst.

3.4.5 InvocationContext

Interceptoren bekommen immer einen `InvocationContext` übergeben, dessen `proceed()`-Methode sie aufrufen müssen, damit der Container den nächsten Interceptor oder die eigentliche Geschäftsmethode aufrufen kann. Dies wurde weiter oben schon gezeigt.

Callback	Paket	Bean	Beschreibung
`@PostConstruct`	javax.annotation	SB, MDB	Dieser Callback wird nach der Instanziierung, aber vor dem ersten Aufruf einer Geschäftsmethode aufgerufen. Damit ist es möglich, Ressourcen für die Instanz zu beschaffen. Es ist durch den Container sichergestellt, dass die Dependency Injection (siehe Abschnitt 6.1) bereits stattgefunden hat.
`@PreDestroy`	javax.annotation	SB,MDB	Hiermit wird der Instanz signalisiert,dass sie vom Container zerstört werden wird. Die Instanz bekommt damit also eine Chance, Ressourcen kontrolliert freizugeben.
`@PostActivate`	javax.ejb	SFSB	Wie in Abschnitt 3.1.2 beschrieben, können Stateful Session Beans auf einen Sekundärspeicher passiviert werden. Dieser Callback wird nach dem Reaktivieren des Beans aufgerufen.
`@PrePassivate`	javax.ejb	SFSB	Dieser Callback stellt das Pendant zu PostActivate dar und wird direkt vor dem Passivieren aufgerufen.
`@Timeout`	javax.ejb	SB, MDB	Dieser Callback wird immer dann aufgerufen, wenn ein Timeout eines Timers stattgefunden hat. Siehe auch Abschnitt 3.5.

Tabelle 3.1
Liste der Callbacks

Dieses `InvocationContext`-Objekt bietet noch einige andere Methoden:

- `Object getTarget()`: Diese Methode liefert die aufzurufende Instanz der Bean-Klasse.
- `Method getMethod()`: Diese Methode liefert die aufgerufene Methode der Bean-Klasse. Im Falle von Lebenszyklus-Interceptoren (wie im vorherigen Abschnitt gezeigt) liefert die Methode `null` zurück.
- `Object[] getParameters()`: Diese Methode liefert die Aufrufparameter für die aufgerufene Methode zurück. Der Interceptor kann die Parameter damit auslesen und modifizieren. Wurden die Parameter bereits über `setParameters()` geändert, werden diese modifizierten Parameter zurückgeliefert.
- `void setParameters(Object[])`: Mit dieser Methode kann der Interceptor die Aufrufparameter für die aufgerufene Zielmethode setzen. Die Typen der Parameter müssen dabei mit denen für die

aufzurufende Methode übereinstimmen – ansonsten wird eine `IllegalArgumentException` geworfen.

- `Map getContextData()`: In dieser Map werden Informationen gehalten, die zwischen Interceptoren innerhalb einer Aufrufkette ausgetauscht werden können.

3.4.6 Interceptoren, Callbacks und Vererbung

In den bisherigen Beispielen traten alle Klassen ohne Vererbung auf. Was geschieht aber, wenn eine Klasse, die einen Interceptor verwendet, von einer anderen Klasse erbt, die ihrerseits einen anderen Interceptor verwendet?

Wie bereits geschrieben, werden als Allererstes die *Default*-Interceptoren angewendet. Dies gilt sowohl für Callbacks als auch für die Business-Interceptoren.

Danach muss man zwischen Callbacks und Interceptoren für Geschäftsmethoden unterscheiden, wobei sie sich im Großen und Ganzen recht ähnlich verhalten.

Callbacks

Sind an der Bean-Klasse Callbacks über `@Interceptors` angebracht, werden diese als Erstes aufgerufen, wobei die Reihenfolge der Spezifikation innerhalb der `@Interceptors`-Annotation eingehalten wird. Hat eine Interceptorklasse Superklassen, werden deren (entsprechende) Callbacks vor den Callbacks der genannten Klasse aufgerufen.

Nach der Ausführung aller Callbacks auf Klassenebene werden die Callbacks auf Methodenebene ausgeführt. Dabei werden auch wieder die Callbacks der Superklasse(n) zuerst ausgeführt.

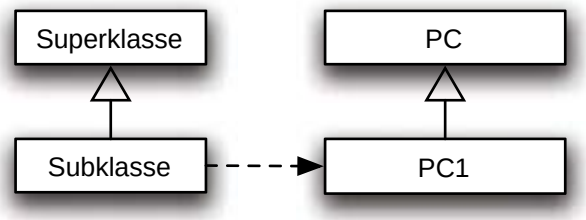


Abbildung 3.5 *Beispielszenario für Callbacks*

Angenommen, die beiden Klassen *Superklasse* und *Subklasse* in Abbildung 3.5 hätten je eine lokale mit `@PostConstruct` markierte Methode. Des Weiteren wäre auf Klassenebene bei *Subklasse* die Klasse *PC1* als Interceptor angegeben. Beide Klassen *PC* und *PC1* würden als

Callback-Interceptor fungieren. Dann ergäbe sich folgende Reihenfolge der Callback-Aufrufe:

- PC
- PC1
- Superklasse
- Subklasse

Wurde eine Methode im Callback-Interceptor (also eine Methode, die z.B. mit `@PostConstruct` markiert wurde und in die Bean-Klasse via `@Interceptors` eingebunden wird) von einer anderen Methode überschrieben, wird der Callback nicht ausgeführt – und zwar unabhängig davon, ob die überschreibende Methode selbst ein Callback ist oder nicht. Wenn also im Beispiel die Callback-Methode in *PC1* die Methode *PC* überschreiben würde, wäre die Reihenfolge

- PC1
- Superklasse
- Subklasse

Interceptoren an Geschäftsmethoden

Sind an der Bean-Klasse Interceptoren über `@Interceptors` angebracht, werden diese als Erstes aufgerufen, wobei die Reihenfolge der Spezifikation innerhalb der `@Interceptors`-Annotation eingehalten wird. Hat eine Interceptorklasse Superklassen, werden deren mit `@AroundInvoke` gekennzeichnete Methoden vor den `@AroundInvoke`-Methoden der genannten Klasse aufgerufen.

Nach der Ausführung der Interceptoren auf Klassenebene werden die auf Methodenebene definierten Interceptoren ausgeführt.

- Zuerst werden die an der Methode selbst definierten Interceptoren aufgerufen.
- Falls vorhanden, werden danach die durch `@AroundInvoke` gekennzeichneten Methoden der Superklassen ausgeführt, wobei mit der Superklasse begonnen wird, die in der Vererbungshierarchie am nächsten bei *Object* ist.
- Zu guter Letzt wird eine in der Bean-Klasse vorhandene `@AroundInvoke`-Methode aufgerufen.

Wird eine mit `@AroundInvoke` gekennzeichnete Methode überschrieben, wird der Interceptor nicht ausgeführt – und zwar unabhängig davon, ob die überschreibende Methode selbst ein Interceptor ist oder nicht.

Wird ein Interceptor an einer Geschäftsmethode verwendet, die als Webservice exportiert wird, und hat dieser Webservice Message Handler definiert, werden die Message Handler vor dem Interceptor aufgerufen. Die Message Handler werden in Abschnitt 7.4 beschrieben.

3.5 EJB Timer Service

Seit J2EE 1.4 bietet das System einen Timerservice, über den sich Applikationen zu bestimmten Zeitpunkten benachrichtigen lassen können. Die Idee hinter diesen Benachrichtigungen sind Jobs, die beispielsweise jede Nacht ablaufen sollen (»Batch-Jobs«), oder Benachrichtigungen an die Anwendung, wenn seit Beginn eines Geschäftsvorfalls eine bestimmte Zeitspanne überschritten wurde. Wenn auch die Zeitspanne in der Einheit Millisekunden eingestellt wird, sind hier große Zeiträume wie Minuten, Stunden oder Tage gemeint. Als Kurzzeit-Timer für wenige Millisekunden ist der Timerservice ungeeignet.

Diese Benachrichtigungen können für Stateless Session Beans und Message-Driven Beans erfolgen. EJB 2.1 Entity Beans können ebenfalls benachrichtigt werden, da diese komplett durch den Container verwaltet werden; für EJB 3.0 Entity Beans ist dies allerdings nicht mehr möglich. Benachrichtigungen werden an die Bean-Klasse gesendet, in der der Timer aufgesetzt wurde. Beim Timeout wird die Methode in der Klasse aufgerufen, die durch die Annotation `@Timeout` markiert wurde. Alternativ kann die Klasse auch das Interface `TimedObject` über die Methode `ejbTimeout()` implementieren.

@Timeout

Wird eine Methode über `@Timeout` markiert, muss diese Methode die Signatur `void` *`name`*`(Timer timer)` haben, darf weder `static` noch `final` sein und darf keine Application Exception (siehe auch Abschnitt 3.6) werfen.

Listing 3.17 *Definition einer bei einem Timeout aufzurufenden Methode*

```
import javax.ejb.Timer
import javax.ejb.Timeout
...
  @Timeout
  public void foo(Timer timer)
  {

  }
...
```

Zu beachten ist, dass der Container bei einem Timeout eine beliebige Instanz des EJBs benachrichtigt. Es ist also nicht möglich, z.B. einen Timer zu setzen, dann eine langwierige Berechnung zu starten und zu hoffen, dass diese bei einem Timeout eventuell abgebrochen wird. Hier

wird es eher im Gegenteil so sein, dass der Container das Bean, in dem die Berechnung läuft, als beschäftigt ansieht und ein anderes Bean, das im Pool ist, benachrichtigt.

3.5.1 Timer starten

Der Timerservice kann über eine Suche im JNDI oder über

```
TimerService EJBContext.getTimerService()
```

ermittelt werden.

Dieses Interface bietet dann einige Methoden zum Erzeugen eines Timers und zum Auflisten der vorhandenen Timer. Die Methoden zum Erzeugen können alle ein serialisierbares Objekt aufnehmen, das dann bei Ablauf des Timers an die aufgerufene Timeout-Methode übergeben wird.

Listing 3.18 *Beispiel für das Aufsetzen eines Timers, der nach 15 min feuern soll*

```
import javax.ejb.Stateless;
import javax.ejb.Timer;
import javax.ejb.TimerService;
import javax.ejb.Timeout;

@Stateless
public class MySessionBean implements SomeInterface
{
  @Resource SessionContext ctx;

  public void setUpTimer()
  {
    TimerService ts = ctx.getTimerService();
    long min15 = 15*60*1000; // 15 Minuten

    Timer tim = ts.createTimer(min15,
       "Info über den Timer");
  }

  @Timeout
  public void timeout(Timer timer)
  {
     String info = timer.getInfo();
  }
}
```

Zu beachten ist, dass für die Timeout-Werte die Einheit Millisekunden ist, die Timer aber deutlich grobgranularer arbeiten. Es kann auch nicht garantiert werden, dass die Timeout-Methode exakt zum angegebenen Moment aufgerufen wird.

Ein laufender Timer kann über die Methode `Timer.cancel()` vor Ablauf gelöscht werden.

Da die mit `@Timeout` markierte Methode nach Ablauf des Timers vom EJB-Container aufgerufen wird, läuft die Methode in einem unspezifizierten Security Context. Das bedeutet, dass ein Aufruf der Methode `EJBContext.getCallerPrincipal()` immer den unauthentifizierten Benutzer zurückliefert.

Timer werden vom System persistiert, so dass sie Reboots des Containers überleben. Somit müssen Timer nicht bei jedem Start der Applikation neu aufgesetzt werden. Die Applikation sollte im Gegenteil vor dem Start eines Timers über `TimerService.getTimers()` sich die aktiven Timer geben lassen, um zu schauen, ob der gewünschte Timer bereits aktiv ist.

3.5.2 Timer und Transaktionen

Läuft die Methode, die den Timer startet, in einer Transaktion und wird diese Transaktion zurückgerollt, wird auch der Timer nicht erzeugt. Möchte man den Timer auf jeden Fall erzeugen, sollte die Methode zur Erzeugung in einer eigenen Transaktion laufen (z.B. durch Auszeichnung der Methode mit `@TransactionAttribute` und einem Parameter REQUIRES_NEW). Siehe auch Abschnitt 3.7.

Dasselbe gilt auch für das Beenden eines Timers: Wird die Transaktion, in welcher der Timer beendet werden soll, zurückgerollt, wird der Timer nicht beendet.

Die Methode, die beim Timeout Callback aufgerufen wird, hat üblicherweise die Transaktionsattribute REQUIRED oder REQUIRES_NEW und wird damit in einer Transaktion aufgerufen. Da die Methode vom Container und nicht von einem Client aus aufgerufen wird, erzeugt auch REQUIRED auf jeden Fall eine neue Transaktion, in welcher der Callback dann läuft.

3.6 Exceptions

Bei den Exceptions gibt es im neuen Standard auch einige kleinere Änderungen. Generell wird zwischen *Application Exceptions*, also Ausnahmen in der Geschäftslogik, und *System Exceptions* als Ausnahmen in der technischen Infrastruktur unterschieden.

3.6.1 Application Exceptions

Eine *Application Exception* ist eine Exception, welche durch die Geschäftslogik einer Anwendung geworfen wird. Mit ihr werden fachli-

che Fehler an den Aufrufer gemeldet. Mit ihnen sollten keine Fehler des Systems gemeldet werden (hierzu gibt es die unten beschriebenen System Exceptions).

Application Exceptions fallen in die beiden Klassen:

- *checked*: Subklassen von `java.lang.Exception`, außer solcher von `java.lang.RuntimeException`. Checked Exceptions dürfen auch keine Subklasse von `java.rmi.RemoteException` sein.
- *unchecked*: Subklassen von `java.lang.RuntimeException`

@ApplicationException

Application Exceptions sorgen nicht automatisch für ein Rollback der laufenden Transaktion. Um dies zu erreichen, muss der Exception-Klasse die `@ApplicationException`-Annotation mitgegeben werden, bei der das Attribut *rollback* auf *true* gesetzt wurde. Damit »unchecked« Exceptions als Application Exception angesehen werden, müssen sie auf jeden Fall mit `@ApplicationException` markiert werden.

Listing 3.19 *Definition einer Application Exception mit Rollback*

```
import javax.ejb.ApplicationException;

@ApplicationException(rollback=true)
public class TestException extends Exception {
   ...
}
```

Alternativ kann hier natürlich weiterhin vor dem Werfen der Exception `EJBContext.setRollbackOnly()` aufgerufen werden.

Application Exceptions und EJB 2.x Clients

Für EJB 3.0 Session Beans, die für EJB 2.x Clients bereitgestellt werden (siehe Abschnitt 3.1.5), müssen im Home Interface auch die entsprechenden EJB 2.x Exceptions in der `throws`-Klausel angegeben werden. Dies sind insbesondere:

- javax.ejb.CreateException
- javax.ejb.RemoveException
- javax.ejb.FinderException

Dies wurde auch bereits in Listing 3.4 auf Seite 34 gezeigt.

3.6.2 System Exceptions

System Exceptions signalisieren Fehler in der Infrastruktur. Diese Exceptions sind `java.rmi.RemoteExceptions` oder »unchecked« Exceptions, die keine Application Exceptions sind, die also nicht via `@ApplicationException` markiert wurden. Beispiele hierfür sind Fehler beim Lookup

einer Ressource im JNDI, Fehler beim Holen einer Datenbankverbindung oder auch Fehler, die aus der JVM herrühren.

Wenn das Bean auf solche Fehler trifft, die es selbst nicht behandeln kann oder möchte, kann es diese an den Container weiterleiten oder eine `javax.ejb.EJBException` werfen. Der Vorteil von `EJBExceptions` ist, dass diese eine Unterklasse der `RuntimeException` ist und damit nicht in den `throws`-Klauseln der Methoden gelistet werden muss.

Speziell die `RemoteException` oder eine davon abgeleitete Exception sollten nun nicht mehr geworfen werden. Die `RemoteException` ist ein Relikt aus den EJB-1.x-Zeiten und wurde bereits in EJB 2.x abgekündigt. Dies gilt auch für das Remote Interface oder Webservices.

Erhält der Container eine System Exception, sorgt er auf jeden Fall für ein Rollback einer laufenden Transaktion. Ebenso wird sichergestellt, dass keine weiteren Methoden der Instanz aufgerufen werden, in der die System Exception geworfen wurde.

3.7 Transaktionen

Transaktionen dienen dazu, eine Operation, die sich über mehrere Schritte oder Komponenten erstreckt, entweder ganz oder gar nicht auszuführen. Transaktionen im Java-EE-Sinne sind nicht nur Datenbanktransaktionen, sondern können auch Verbindungen zu Legacy-Systemen oder über JMS etc. umfassen. Mit den Standardeinstellungen (bzw. wenn nichts anderes angegeben wird) wird jeder Aufruf von einem Client heraus in einer großen Transaktion abgearbeitet. Ressourcen, die innerhalb der Transaktion benötigt werden, werden dabei für den Schreibzugriff gesperrt. Dies erfolgt selbst dann, wenn dies nicht nötig wäre, weil die Ressource im Sinne der Geschäftslogik nur zum Lesen ist.

Transaktionen können auch in EJB 3.0 in den Varianten Container-managed und Bean-managed erfolgen. Im Gegensatz zu den EJB 2.1 Beans beschränkt sich diese Einstellung allerdings auf Session und Message-Driven Beans; die EJB 3.0 Entity Beans laufen immer innerhalb des Transaktionskontexts des Aufrufers. Ansonsten gilt auch hier die Vereinfachung über die Verwendung von Annotationen zur Markierung des transaktionalen Verhaltens einer Klasse oder Methode.

Ob die Transaktion vom Container oder von der Bean-Klasse selbst gemanagt werden soll, kann über die Annotation `@TransactionManagement` an der Bean-Klasse spezifiziert werden. Wird die Annotation nicht gegeben, erfolgt ein Management durch den Container.

@TransactionManagement

TransactionManagementType

Der Parameter *value* der Annotation ist vom Typ `TransactionManagementType` und kann die folgenden beiden Werte annehmen:

- `BEAN`: Management durch die Applikation selbst (*Bean-Managed Transaction, BMT*).
- `CONTAINER`: Transaktionsmanagement durch den Container (*Container-Managed Transaction, CMT*). Dies ist die Voreinstellung, wenn nichts anderes angegeben wurde.

Listing 3.21 auf Seite 64 zeigt unter anderem die Einstellung für Bean-Managed Transactions in einem Session Bean.

Die Verwendung der beiden Arten des Transaktionsmanagements wird in den nächsten Abschnitten näher erläutert.

3.7.1 Container-Managed Transactions

Mittels `@TransactionAttribute` wird im Fall der Container-Managed Transactions, das transaktionale Verhalten der Methode oder Klasse angegeben. Die Attribute sind dieselben wie in früheren Versionen des Standards. Es wurde lediglich die aus den Deployment-Deskriptoren bekannte Schreibweise mit den Binnenmajuskeln zugunsten der Java-Konvention, dass Konstanten groß geschrieben werden, angepasst.

@TransactionAttribute TransactionAttributeType

`@TransactionAttribute` hat nur einen Parameter *value*, der vom Typ `TransactionAttributeType` ist und die folgenden Werte annehmen kann:

- `MANDATORY`: Wird die Methode ohne Transaktionskontext aufgerufen, wirft der Container eine `TransactionRequired` Exception.
- `NEVER`:: Das Bean unterstützt keine Transaktionen; wenn es in einer Transaktion aufgerufen wird, wird vom Container eine Exception geworfen.
- `NOT_SUPPORTED`: Wird die Methode in einem Transaktionskontext aufgerufen, unterbricht der Container den Kontext und ruft die Methode ohne Kontext auf. Im Deployment-Deskriptor wird diese Einstellung mit NotSupported bezeichnet.

Default-Verhalten: REQUIRED

- `REQUIRED`: Beinhaltet der Aufruf des Beans bereits einen Transaktionskontext, wird das Bean in diesem ausgeführt. Erfolgt der Aufruf ohne Transaktionskontext, wird vom Container ein neuer Kontext erzeugt. Dies ist der Standard für CMT, wenn keine andere Einstellung angegeben ist.
- `REQUIRES_NEW`: Wenn das Bean aufgerufen wird, wird ein eventuell vorhandener Kontext unterbrochen und ein neuer Kontext erzeugt, in welchem das Bean dann aufgerufen wird. Nach dem Aufruf wird der unterbrochene Transaktionskontext fortgesetzt

– und zwar unabhängig davon, ob die eingeschobene Transaktion erfolgreich war oder nicht. Die Schreibweise im Deployment-Deskriptor ist RequiresNew.

- `SUPPORTS`: Die Methode unterstützt Transaktionen, kann aber auch ohne Transaktionskontext korrekt funktionieren.

Die Angabe von `@TransactionAttribute` auf Klassenebene bewirkt, dass alle Methoden mit dieser Einstellung laufen. Wird die Annotation auf Methodenebene angegeben, wird eine etwaige Markierung auf Klassenebene überschrieben.

Container-managed Transactions dürfen auch weiterhin keine `UserTransaction` verwenden.

Verwendung von REQUIRED und REQUIRES_NEW

Wir wollen nun noch kurz zwei Szenarien anschauen, mit denen die Verwendung der Flags REQUIRED und REQUIRES_NEW erläutert werden soll. Von einer Session-Bean-Methode, für die über das Flag REQUIRED auf jeden Fall eine Transaktion geöffnet wird, werden drei weitere Methoden aufgerufen, wobei eine dieser Methoden einen Audit-Log über durchgeführte Aktionen in die Datenbank schreiben soll. Abbildung 3.6 zeigt dieses Szenario als Sequenzdiagramm.

Betrachten wir zunächst den Fall, bei dem alle drei aufgerufenen Methoden ebenfalls mit REQUIRED markiert wurden (sei es implizit über die Default-Einstellung oder explizit über `@TransactionAttribute(REQUIRED)`).

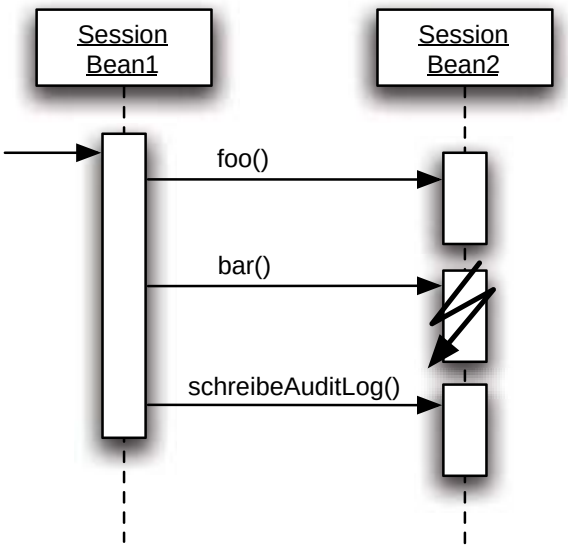


Abbildung 3.6
Szenario, bei dem alle Methoden mit REQUIRED gekennzeichnet sind

Läuft die Verarbeitung korrekt durch, wird das Audit-Log geschrieben und der Aufruf kehrt erfolgreich zum Aufrufer zurück. Tritt allerdings, wie in Abbildung 3.6 gezeigt, eine Exception auf und wird die Transaktion auf Zurückrollen gesetzt, wird auch der Audit-Log nicht geschrieben, da die umklammerte Datenbanktransaktion ebenfalls zurückgerollt wird. Dies ist sicherlich nicht gewünscht. Im Gegenteil wird man speziell diese Ausnahmesituation im Logfile finden wollen.

Im zweiten Szenario wird die Methode *schreibeAuditLog()* mit dem Transaktionsflag REQUIRES_NEW versehen. Damit wird die äußere Transaktion angehalten und *schreibeAuditLog()* läuft in einer eigenen neuen Transaktion, die beendet (committed) wird, wenn die Methode *schreibeAuditLog()* beendet ist. Damit kann das Audit-Log erfolgreich geschrieben werden, obwohl die äußere Transaktion zurückgerollt wird.

3.7.2 CMT und Vererbung in EJB 3.0

In EJB-Versionen vor 3.0 mussten die Transaktionsattribute für Container-Managed Transactions immer im Deployment-Deskriptor angegeben werden. In EJB 3.0 können diese Attribute nun über Annotationen direkt an den betroffenen Methoden gesetzt werden, was zur Folge hat, dass hier die Vererbung beziehungsweise das Überschreiben von so markierten Methoden beachtet werden muss. Es gelten die folgenden Regeln:

- Wird an einer Klasse keine Annotation `@TransactionAttribute` angegeben, werden alle nicht markierten Methoden mit dem Attribut REQUIRED betrieben, da dies die Voreinstellung für CMT ist.
- Transaktionsattribute an Methoden einer Superklasse werden in die Subklassen »mitgenommen«.
- Transaktionsattribute an Methoden in der Subklasse überschreiben Attribute der Superklasse.

Zum Beispiel seien die beiden Klassen Ober und Unter wie in Listing 3.20 angeordnet:

Listing 3.20 *Beispiel für die Vererbung von Transaktionsattributen*

```
@TransactionAttribute(SUPPORTS)
public class Ober
{
   public void foo() {...}

   public void bar() {...}
}
```

```
@Stateless
public class Unter extends Ober implements SomeIf
{
   public void foo() {...}

   @TransactionAttribute(REQUIRES_NEW)
   public void baz() {...}

   public void bam() {...}
}
```

Die Transaktionseinstellungen für die einzelnen Methoden sind dann:

- REQUIRED für *Unter.foo*, da die Klasse *Unter* mit REQUIRED implizit markiert wurde, für *Unter.foo* keine weitere Annotation angegeben wurde und das Attribut auf Klassenebene durch die Auswirkung auf die Methode die Einstellung von *Ober.foo* überschreibt.
- SUPPORTS für *Unter.bar*, da das Attribut SUPPORTS aus *Ober* mitgenommen wird und in *Unter* nicht redefiniert wurde.
- REQUIRES_NEW für *Unter.baz*, da diese Methode direkt mit einem Transaktionsattribut gekennzeichnet wurde, das die Einstellungen auf Klassenebene überschreibt.
- REQUIRED für *Unter.bam*, da die Klasse *Unter* mit REQUIRED implizit markiert wurde und für *Unter.bam* keine weitere Annotation angegeben wurde.

3.7.3 Verwendung von CMT-Attributen in EJB 3.0 Beans

Obwohl es sechs verschiedene Transaktionsattribute gibt, können nicht alle für die EJBs in Version 3.0 verwendet werden. Die nachfolgende Auflistung zeigt die Restriktionen.

- Message-Driven Beans: Für die *Message-Listener*-Methode der Message-Driven Beans können nur die beiden Einstellungen `REQUIRED` und `NOT_SUPPORTED` verwendet werden.
- Timeout-Callback: Hier sind nur `REQUIRED`, `REQUIRES_NEW` und `NOT_SUPPORTED` erlaubt.
- Session Beans mit SessionSynchronization: Implementiert ein Session Bean das Interface `SessionSynchronization`, sind nur noch `REQUIRED`, `REQUIRES_NEW` und `MANDATORY` als Transaktionseinstellungen erlaubt.

3.7.4 Bean-Managed Transactions

Möchte ein Bean nicht an der Transaktionsverwaltung durch den Container (CMT) teilnehmen, sondern die Transaktionen selbst verwalten (BMT), kann es dies dem Container auf Klassenebene via `@TransactionManagement(BEAN)` mitteilen und dann `UserTransactions` nutzen. Diese lassen sich neben einem Lookup im JNDI nun auch über `@Resource` injizieren, wie dies in Listing 3.21 gezeigt wird.

Listing 3.21 *Injektion der UserTransaction*

```
import static javax.ejb.TransactionManagementType.BEAN;
import javax.ejb.Stateful;

@TransactionManagement(BEAN)
@Stateful
public class MyClass implements MyClassIF
{
  @Resource
  javax.transaction.UserTransaction ut;

  public void machWas()
  {
    ut.begin();
    // ...
  }
}
```

Anwendungsfälle sind beispielsweise Stateful Session Beans, bei denen der Aufruf einer Methode die Transaktion starten soll und der Aufruf einer anderen Methode die Transaktion beenden soll.

Die Injektion der `UserTransaction` kann auch in Servlets und EJB-Clients genutzt werden.

3.8 Security

Die Java Enterprise Edition definiert ein rollenbasiertes Sicherheitsmodell. Ist dies angeschaltet, bekommen nur Benutzer Zugriff auf Methoden der Enterprise JavaBeans, die in einer der benötigen Gruppen sind. Alternativ können Methoden auch so gekennzeichnet werden, dass alle Benutzer ungeachtet ihrer Rollenzugehörigkeit diese Methoden aufrufen können, oder ihnen der Zugriff verwehrt wird. Ist für eine Methode nichts anderes über Annotationen oder den Deployment-Deskriptor angegeben, wird für sie keine Überprüfung auf das Vorhandensein einer spezifischen Rolle durchgeführt.

Die Security Settings sind prinzipiell wie in EJB 2.1 geblieben und können nun zusätzlich durch Annotationen deklariert werden. Die

nachfolgend aufgelisteten Annotationen aus JSR-250 stehen zur Verfügung und können für Session Beans und Message-Driven Beans verwendet werden. Da die Entity Beans in EJB 3.0 vom Client aus nicht mehr direkt aufgerufen werden können (siehe Kapitel 4), sind diese Annotationen dort nicht notwendig.

*javax.annotation.security.**

Die nachfolgenden Annotationen sind alle in JSR-250 definiert und befinden sich im Paket `javax.annotation.security`.

@DenyAll

- `@DenyAll`: Diese Annotation ist nur auf Methodenebene anwendbar. Über sie wird keiner Rolle erlaubt, die bezeichnete(n) Methode(n) aufzurufen.

@PermitAll

- `@PermitAll`: Diese Annotation ist das Gegenstück zu `@DenyAll`: Alle Rollen dürfen die bezeichnete Klasse oder Methode aufrufen. Wird die Annotation auf Klassenebene gegeben, gilt sie für alle Methoden der Klasse.

@RolesAllowed

- `@RolesAllowed`: Hierüber wird eine Liste von Rollen angegeben, welche die annotierte Methode oder Klasse aufrufen dürfen. Der Parameter ist dabei ein Array von Strings, welche die erlaubten Rollen bezeichnen. Auch hier gilt, dass die Anwendung auf Klassenebene sich auf alle Methoden der Klasse auswirkt.

@RunAs

- `@RunAs`: Legt die Rolle fest, mit der das Bean ausgeführt wird. Dies erfolgt ungeachtet der aufrufenden Rolle.

```
@RolesAllowed("administrator")
@RunAs("user")
public class MyClass {
 // ...
}
```

Methoden der Klasse *MyClass* benötigen also die Rolle *administrator*, damit sie ausgeführt werden können, werden dann aber in der Rolle *user* ausgeführt. Dies beinhaltet Aufrufe zu anderen Beans, die dann ebenfalls in der Rolle *user* ausgeführt werden.

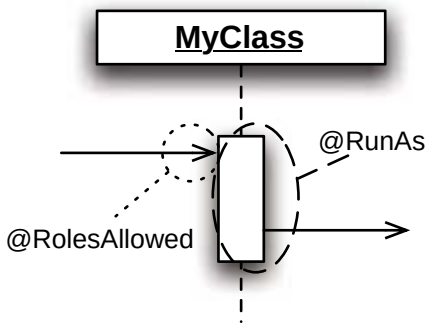


Abbildung 3.7 *Verwendung von RunAs und RollesAllowed*

@DeclareRoles

- `@DeclareRoles`: Diese Annotation auf Klassenebene dient der Auflistung der im Bean verwendeten Rollen. Dies sind üblicherweise Rollen, welche in Aufrufen der Art `isCallerInRole()` verwendet werden. Die in der Annotation angegebenen Rollen sind Referenzen (pro Komponente), die dann im Deployment-Deskriptor auf existierende Rollen im System gemappt werden können. Erfolgt kein solches explizites Mapping, geht der Container davon aus, dass es im System eine Rolle mit dem in `@DeclareRoles` angegebenen Namen gibt.

Die Annotationen `@PermitAll`, `@DenyAll` und `@RolesAllowed` dürfen dabei nie gleichzeitig an einer Methode oder Klasse angebracht werden. Annotationen, die an einer Methode angebracht sind, haben Vorrang vor einer anders lautenden Annotation auf Klassenebene:

```
@PermitAll
public class UnsecureClass {

  @DenyAll
  public void toString() {
    // ...
  }
}
```

Obwohl die gesamte Klasse für alle Aufrufer zugänglich ist, darf kein Benutzer die Methode *toString()* der Klasse aufrufen.

Ein ähnliches Verhalten gilt auch für die Vererbung. Sind die erlaubten Rollen in einer Unterklasse genauer spezifiziert als in der Superklasse, gelten die Einstellungen der Subklasse.

Wie auch innerhalb der Deployment-Deskriptoren überschreibt eine Annotation an einer Methode die Angabe für die gesamte Klasse.

3.9 Deployment-Deskriptor und Paketierung

Für EJB 3.0 wurde auch der Deployment-Deskriptor *ejb-jar.xml* angepasst und erweitert. Dieser ist nun im XML-Schema

```
http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd
```

definert.

Da so gut wie alle Einstellungen über Annotationen gemacht werden können (die Angabe applikationsweiter Default-Interceptoren ist die einzige Ausnahme für Session- und Message-Driven Beans), wird dieser Deployment-Deskriptor hier nicht weiter vertieft. Wer Genaueres wissen möchte, kann dies beispielsweise in [10] nachlesen.

EJBs werden auch in Version 3.0 in JAR-Archive gepackt. Diese Archive können sowohl EJBs der Version 3.0 als auch von früheren Versionen enthalten. Der Deployment-Deskriptor *ejb-jar.xml* wird weiterhin im Verzeichnis *META-INF* abgelegt.

Im Archiv werden dabei sowohl die Bean-Klassen als auch die Geschäftsschnittstellen und die Interceptoren abgelegt. Dies kann sowohl direkt durch Kopieren der Klassen in das Archiv geschehen, als auch indirekt über eine Referenz auf andere Archive duch entsprechende *Class-Path*-Einträge in der Manifest-Datei.

4 Entity Beans als Geschäftsobjekte

Auch Entity Beans sind in EJB 3.0 normale Java-Klassen, die kein spezielles Interface mehr implementieren, sondern als POJO ausgeführt sind. Wer in der Vergangenheit bereits mit Hibernate oder JDO gearbeitet hat, dem werden die neuen Entity Beans sehr bekannt vorkommen. Dies kommt unter anderem daher, dass die Entwickler der genannten Systeme und APIs (und auch Toplink) einen großen Einfluss im Standardisierungsprozess hatten.

Es gibt Diskussionen in der Community, ob die Bezeichnung POJO, Plain Old Java Object, für diese Entity Beans zutrifft, da sie doch mit Annotationen versehen werden müssen. Es wird ebenfalls ein spezielles Primärschlüsselfeld benötigt, was einfache Java-Objekte auch nicht benötigen. Ich bleibe hier allerdings bei »POJO«, da die neuen Entity Beans gegenüber denen von EJB 2.x doch sehr leichtgewichtig sind.

Da die EJB 3.0 Entity Beans auch außerhalb des Java-EE-Umfeldes eingesetzt werden können, werden sie auch als *Java Persistence API*, JPA, bezeichnet. Damit ist es möglich, dasselbe Datenmodell sowohl im Serverbackend als auch in angeschlossenen Anwendungen zu nutzen – lediglich der Datenbestand ist ein anderer.

JPA

Dieses Kapitel und das nächste über die Abfragesprache, Java Persistence Query Language (im Folgenden als JP-QL bezeichnet), führen nur in die Thematik ein und stellen die standardisierten Features vor. Da der unterliegende Persistenz-Provider austauschbar ist, sollte auf jeden Fall die Dokumentation des Herstellers hinzugezogen werden.

JP-QL

Ein großer Unterschied zu den EJB 2.x Entity Beans ist sicher auch, dass Entity Beans in EJB 3.0 nun nicht mehr von anderen JVMs aus angesprochen werden können, was früher durchaus möglich war. Da hier zu viel Schindluder getrieben wurde und auch zusätzliche Klassen wie *ValueObjects* (DTO-Pattern, siehe [22]) notwendig wurden, ist dies nun nicht mehr möglich. Als Belohnung können die Entity Beans nun direkt als *detached object* an Clients übergeben werden. Hierfür muss das Entity Bean die Java-Schnittstelle `Serializable` implementieren.

Das Beispiel in Listing 4.1 soll eine sehr einfache Entity Bean zeigen.

Listing 4.1 *Beispiel einer Entity Bean*

```
import java.io.Serializable;

import javax.persistence.Entity;
import javax.persistence.GeneratorType;
import javax.persistence.Id;
import javax.persistence.Table;

@Table( name = "adb_email")
@Entity
public class Email implements Serializable {

  long id;
  String name;

  protected Email() { }

  @Id
  public long getId() { return id; }
  private void setId(long sid) { id = sid; }

  public String getName() { return name; }
  public void setName(String n) { name = n; }
}
```

Damit die Klasse als Entity Bean angesehen wird, muss sie die Annotation `@Entity` tragen. Über `@Table` kann angegeben werden, in welcher Tabelle die Felder abgespeichert werden sollen. Ist sie nicht angegeben, wird der Klassenname genommen. Schließlich bezeichnet `@Id` das Primärschlüsselfeld.

Der Rest des Kapitels ist nun in zwei große Teile gegliedert:

- Beschreibung der Entity Beans und ihrer Optionen. Hierzu gehören auch Vererbung und das zugehörige Mapping in Tabellen sowie Relationen zu anderen Entity Beans.
- Zugriff auf Entity Beans. Dieser Zugriff erfolgt nun über den sogenannten *Entity Manager*. Die Abfragesprache JP-QL wird separat in Kapitel 5 ab Seite 141 beschrieben.

JP-QL in Kapitel 5

4.1 Die Basics der Entity Beans

Entity Beans sind, wie schon geschrieben, nun einfache POJOs mit diesen Einschränkungen:

- Weder die Klasse selbst noch eine ihrer Methoden darf `final` sein.
- Die Entity-Bean-Klasse darf keine eingebettete Klasse, Interface oder Enum sein.
- Die Klasse muss einen öffentlichen parameterlosen Konstruktor haben, welcher durchaus vom Compiler bereitgestellt werden kann. Weitere Konstruktoren sind zusätzlich möglich.
- Felder, welche persistiert werden sollen, dürfen von Clients nur über Getter und Setter bzw. Geschäftsmethoden an der Klasse angesprochen werden. Aus diesem Grund sollten Felder nicht `public` sein. Das Persistenzsystem kann auf diese Felder auch zugreifen, wenn sie `private` sind.

Felder, welche die Datenbanktabellen abbilden, werden wie beschrieben entweder über Getter und Setter repräsentiert oder liegen nur als Felder vor. Der Unterschied der Verwendung entspricht in etwa dem bei der Dependency Injection (siehe Abschnitt 6.1). Welche Art des Setzens der Felder vom Container verwendet wird, wird durch den Ort der entsprechenden Annotationen vorgegeben. Dabei ist innerhalb einer Hierarchie von Entity Beans ein Mix der Zugriffsschemata nicht erlaubt.

Wird feldbasierter Zugriff gewählt, werden diese Felder von der Persistence Engine direkt befüllt, selbst wenn Getter und Setter vorhanden sind. Dabei werden alle Felder persistiert, welche weder nicht `transient` sind oder durch `@Transient` markiert wurden.

Wird der Zugriff über Getter und Setter gewählt, müssen die Annotationen für das Mapping an den Gettern angebracht werden. Getter und Setter müssen dabei `public` oder `protected` sein. Es werden alle Propertys berücksichtigt, deren Getter nicht durch `@Transient` markiert wurden.

Innerhalb dieser Zugriffsmethoden darf zusätzlicher Code, wie etwa für Validierungen, ausgeführt werden. Allerdings wird vom Container keine Reihenfolge für den Aufruf der Zugriffsmethoden garantiert, so dass in diesem Validierungscode darauf kein Bezug genommen werden darf.

Wie im Beispiel in Listing 4.1 zu sehen war, muss eine Entity-Bean-Klasse die Annotation `@Entity` tragen. Diese Annotation kennt nur das Attribut *name*. Dieses dient dazu, das Entity Bean in Abfragen ansprechen zu können. Ist das Attribut nicht angegeben, wird der unqualifizierte Klassenname verwendet. Da der Name in JP-QL-Abfragen verwendet wird, darf er kein dort reserviertes Schlüsselwort sein. Folglich darf also auch der Klassenname selbst kein dort definiertes Schlüsselwort sein. Diese Schlüsselwörter sind ab Seite 141 im Abschnitt 5.1.1 gelistet.

@Entity

4.1.1 Tabellennamen

Der Name der Datenbanktabelle, in der ein Entity Bean gespeichert wird, kann über die `@Table`-Annotation an der Klasse angegeben werden. Ist die Annotation nicht vorhanden, werden die entsprechenden Voreinstellungen der Annotation implizit verwendet. Dies betrifft insbesondere den Tabellennamen.

@Table

Die Parameter von `@Table` sind:

- name: Dieser Parameter vom Typ `String` gibt den Tabellennamen an. Ist dieser nicht angegeben, wird der unqualifizierte Klassenname des Entity Beans verwendet.
- catalog: Dieser `String`-Parameter gibt einen zu verwendenden Datenbankkatalog an. Ist das Attribut nicht gesetzt, wird der Standardkatalog verwendet.
- schema: Über diesen Parameter kann ein zu verwendendes Schema angegeben werden. Die Voreinstellung ist hier das Standardschema des Benutzers.
- uniqueConstraints: Eine Reihe von Eindeutigkeits-Bedingungen, welche über die Beschränkungen der Felder und Relationen hinausgehen. Dieses Attribut ist nur von Bedeutung, wenn der Persistenz-Provider die Tabellen beim ersten Start der Anwendung selbst anlegen soll und diese Constraints dann direkt mit anlegt. Alternativ kann diese Aufgabe auch durch Werkzeuge erfolgen, welche die Klassendateien nach den entsprechenden Annotationen durchsuchen. Die Werte dieses Attributs können eine oder mehrere `@UniqueConstraints` sein, die ihrerseits eine Reihe von Strings haben können, welche die Namen der Spalten angeben, deren Werte eindeutig sein sollen.

  @UniqueConstraint

  Soll nur eine einzige Spalte gekennzeichnet werden, lässt sich dies besser über `@Column` (siehe Abschnitt 4.1.3) erledigen.

4.1.2 Ablage in mehreren Tabellen

Sollen die Felder des Entity Beans in mehreren Tabellen abgelegt werden, kann eine weitere Tabelle über `@SecondaryTable` auf Klassenebene angegeben werden. Zusätzlich ist es allerdings noch notwendig, die Abbildung der einzelnen Felder auf die Tabellen über die weiter unten beschriebene `@Column`-Annotation zu definieren.

@SecondaryTable

Die meisten Attribute dieser Annotation sind bereits von `@Table` her bekannt:

- name: Dieser Parameter vom Typ `String` gibt den Tabellennamen an. Ist dieser nicht angegeben, wird der unqualifizierte Klassenname des Entity Beans verwendet.
- catalog: Dieser `String`-Parameter gibt einen zu verwendenden Datenbankkatalog an. Ist das Attribut nicht gesetzt, wird der Standardkatalog verwendet.
- schema: Über diesen Parameter kann ein zu verwendendes Schema angegeben werden. Die Voreinstellung ist hier das Standardschema des Benutzers.
- pkJoinColumns: Ein oder mehrere `@PrimaryKeyJoinColumn`-Einträge (siehe Abschnitt 4.3.4), welche für den Join dieser Tabelle mit der primären Tabelle verwendet werden. Ist das Attribut nicht gegeben, werden automatisch Spalten mit demselben Namen und Typ wie die Primärschlüsselspalte(n) der primären Tabelle verwendet.
- uniqueConstraints: Eine Reihe von ergänzenden Bedingungen zur Eindeutigkeit von Werten in der Spalte. Diese wurden bereits im letzten Abschnitt näher erläutert.

@SecondaryTables

Da eine Annotation nicht mehrfach an einem Element vorkommen darf, können mehrere zusätzliche Tabellen über `@SecondaryTables` an der Klasse zusammengefasst werden. Das Argument der Annotation ist dabei eine Liste von `@SecondaryTable`-Einträgen:

```
@Entity
@SecondaryTables({
   @SecondaryTable(name="second"),
   @SecondaryTable(name="third")
 })
@Table(name="first")
public class VieleTabellen
```

Hiermit würde also definiert, dass die Daten der Entity *VieleTabellen* auf die Tabellen »first«, »second« und »third« aufgeteilt werden. Wie dies genau geschieht, muss über `@Column` definiert werden.

4.1.3 Zu persistierende Felder

@Transient

Alle Felder oder Propertys, die in einem Entity Bean angegeben sind, werden automatisch persistiert, wenn es nicht über die `@Transient`-Annotation anders angegeben wurde bzw. das Feld als `transient` gekennzeichnet wurde.

Im Normalfall bestimmt der Container die Abbildung zwischen Java-Typ, JDBC-Typ und SQL-Typ der jeweiligen Datenbank. Dieses Mapping bestimmt der Persistenz-Provider anhand einer expliziten

Konfiguration oder der Metadaten, die über den JDBC-Treiber ermittelt wurden. Details hierzu sind produktspezifisch.

Im Rahmen dieser Umsetzung wird auch der Spaltenname aus dem Feldnamen abgeleitet, und die Felder landen in der primären Tabelle, selbst wenn mehr als eine Tabelle für die Entity Bean vorgegeben wurde.

Wenn diese Annahmen nicht zutreffen, können diese gezielt über einige Annotationen geändert werden. Diese werden nun vorgestellt.

@Basic

Diese Annotation ist in den meisten Fällen nicht notwendig, da ja alle Felder, die nicht via `@Transient` markiert wurden und die weder `static` oder `transient` sind, persistiert werden.

@Basic

Über `@Basic` kann zusätzlich angegeben werden, zu welchem Zeitpunkt das Feld geladen werden soll, und ob es `null` sein darf. Dies geschieht über diese beiden Parameter:

FetchType
Default: EAGER

- fetch: Dieser Parameter ist vom Typ `FetchType` und kann die Werte `EAGER` (Voreinstellung) und `LAZY` annehmen. Beim ersten Wert wird das Feld sofort beim Laden des Objekts aus der Datenbank gelesen, während bei `LAZY` dies erst beim Zugriff auf das Feld geschieht.

  Im Normalfall wird man die meisten oder alle Felder eines Objekts sofort laden wollen. In manchen Fällen kann es aber sinnvoll sein, ein Feld erst zu laden, wenn es benötigt wird. Zum Beispiel kann man aus einer Tabelle, in der Daten zu einer Person mit ihrem Bild gespeichert sind, das Bild erst dann laden, wenn es angezeigt werden soll, während es in einer Liste mit Suchtreffern nicht benötigt wird.

  Obwohl Lazy Loading auf den ersten Blick sehr vorteilhaft scheint, hat es auch Schattenseiten:

  - Das Nachladen der übersprungenen Objekte benötigt einen erneuten Datenbankzugriff. Es muss also genau geklärt werden, wann sich dies lohnt.
  - Wird ein Entity Bean beispielsweise an eine JSP-Seite weitergegeben und damit *detached* (siehe Abschnitt 4.7.4 ab Seite 118), kann das vorher nicht geladene Feld nicht direkt nachgeladen werden.

  Diese Problematik wird in Abschnitt 4.6.7 weiter vertieft (dort für Relationen, die Problematik ist aber dieselbe).
- optional: Dieser boolesche Parameter gibt an, ob das Feld `null` sein darf oder nicht. Für die primitiven Datentypen (`int` etc.) wird dieser Parameter ignoriert. Auf das Mapping an sich hat

dieser Parameter keinen Einfluss, sondern wird lediglich zur Generierung von Schemata genutzt. Dies kann beispielsweise über entsprechende externe Werkzeuge geschehen, welche die Annotationen aus den Klassen auslesen.

Beide Parameter dienen lediglich als Hinweis an die Persistenz-Engine. Die Engine muss die Hinweise nicht befolgen. Speziell im Fall des Zugriffszeitpunkts kann die Engine diesen Hinweis auch nur dann ausnutzen, wenn der Zugriff auf die Daten über die Propertys erfolgt.

Listing 4.2
Verwendung der Basic Annotation

```
import javax.persistence.Basic;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.Id;

@Entity
public class BasicEntity {
  long id;
  char[] langerWert;

  @Id
  public long getId() { return id; }
  private void setId(long sid) { id = sid; }

  @Basic(fetch=FetchType.LAZY)
  public char[] getLangerWert()
  { return langerWert ; }

  ...
}
```

@Column

Wird vom Entwickler nichts anderes angegeben, verwendet der Container eine ganze Reihe an Voreinstellungen für die Abbildung eines Java-Feldes in eine Datenbankspalte. Zum Beispiel wird der Feldname als Spaltenname übernommen. Dies muss nicht in allen Fällen richtig sein – speziell wenn die Datenbanktabellen schon existieren, gibt es auch Regeln, an die sich das O/R-Mapping halten muss. Über die Annotation `@Column` kann diese Umsetzung geändert und beeinflusst werden. Diese Annotation kennt eine ganze Reihe von Parametern, die alle optional sind:

@Column

- name: Der Name der Spalte. Ist der Parameter nicht vorhanden, wird der Feldname verwendet.
- length: Die Breite der Spalte für Strings. Als Voreinstellung wird hier der Wert 255 genommen. Änderungen an diesem Wert wirken sich nur auf die DDL aus. Der Persistenz-Provider muss keine Prüfung der übergebenen Daten auf Einhaltung der Länge durchführen. Dies könnte allerdings von der Anwendung über einen `@PrePersist`- oder `@PreUpdate`-Callback durchgeführt werden. Siehe hierzu Abschnitt 4.7.7 auf Seite 127.
- unique: Dieser boolesche Wert gibt an, ob die Werte in dieser Spalte eindeutig sein müssen. In der Voreinstellung ist dies nicht der Fall. Dieses Attribut ist eine Vereinfachung gegenüber `@UniqueConstraint` auf Tabellenebene.
- nullable: Dieser boolesche Wert gibt an, ob in der Datenbankspalte `null`-Werte enthalten sein dürfen oder nicht. In der Voreinstellung sind `null`- Werte erlaubt.
- insertable: Mit diesem Parameter wird angegeben, ob Werte in diese Spalte über insert-Statements geschrieben werden dürfen. Die Voreinstellung ist `true`.
- updatable: Über diesen Parameter wird das Verhalten bei SQL-Update gesteuert. Ist er auf `true` (Voreinstellung) gesetzt, wird die Spalte bei der Aktualisierung berücksichtigt.
- columnDefinition: Dieser String-Wert kann ein DDL-Fragment enthalten, das beim Anlegen der Tabelle für diese Spalte genutzt werden kann. Dieses Anlegen kann über externe Tools oder auch durch den Persistenz-Provider erfolgen. Da der String direkt ein SQL-DDL-Fragment ist, kann er unter Umständen nicht zwischen verschiedenen Datenbanken portabel sein.
- table: Der Name der Tabelle, in welche die Spalte geschrieben wird. Üblicherweise ist dies die Tabelle, die via `@Table` definiert wurde und muss nicht extra angegeben werden. Soll die Spalte in einer anderen Tabelle landen, die über `@SecondaryTable` definiert wurde, kann diese Tabelle hier angegeben werden.
- precision: Dieser Parameter wird nur für Dezimalzahlen genutzt und gibt die Gesamtanzahl der Dezimalstellen an.
- scale: Auch dieser Parameter wird nur für Dezimalzahlen genutzt und gibt die Anzahl der Nachkommastellen (für positive Werte von scale) an.

Für *scale* und *precision* liefert die JavaDoc-Seite für *java.math.BigDecimal* eine ausführliche Erläuterung. Stehen *updateable* und *insertable* jeweils auf `false`, wird aus der so bezeichneten Spalte nur gelesen.

```
@Column(length=16, unique=true)
public String getName() { return name; }

@Lob
@Column(table="WL_PExtra", columnDefinition="CLOB")
@Basic(fetch=FetchType.LAZY)
public String getGeekCode() { return geekCode; }
```

Listing 4.3 *Beispiele für die Verwendung von @Column*

@Enumerated

Java SE 5.0 bietet nun einen eigenen Aufzählungstyp, mit dem typsichere Aufzählungen realisiert werden können. Um Werte dieser Aufzählung persistieren zu können, gibt es die `@Enumerated`-Annotation. Diese Annotation kennt den Parameter *value*, der die Art der Persistierung angibt. Dieser Parameter ist vom Typ `EnumType`, der diese beiden Werte kennt:

@Enumerated

EnumType

- `ORDINAL`: Der Wert wird als Zahl beginnend mit 0 gespeichert. Dies ist der Normalfall.
- `STRING`: Der Wert wird als String gespeichert. Hierfür wird der Wert als String-Form gespeichert.

Das folgende Beispiel zeigt die Verwendung.

```
package de.bsd.weblog.persistence;

public enum PLevel {
       NEU, NORMAL, EXPERTE
}

@Entity
public class Poster ...
{
  // ...
  @Enumerated(EnumType.ORDINAL)
  public PLevel getLevel() {
    return level;
  }
}
```

Listing 4.4 *Beispiel für die Persistierung von Enums*

Hat ein Poster den Level *NEU*, wird in die Spalte *PLevel* der Wert 0 geschrieben. Würde die Speicherung als String erfolgen (via `@Enumerated(STRING)`), würde der String »NEU« in die Spalte geschrieben. Hierfür gibt es ein Beispiel in der Klasse Nutzer im Weblog.

@Lob

@Lob

Generische Objekte oder Strings, die eine gewisse Länge überschreiten, werden üblicherweise als *Large Object*, LOB, in der Datenbank gespeichert. Über `@Lob` kann für ein Feld angegeben werden, dass es als LOB gespeichert werden soll. Die Annotation selbst kennt keine Parameter. Welcher Typ von LOB verwendet wird, hängt von der unterliegenden Datenbank ab, wobei normalerweise Strings und Zeichenfelder als CLOB und alle anderen Typen als BLOB gespeichert werden. Dies lässt sich aber über `@Column` ändern, wie es in Listing 4.3 gezeigt wurde.

Listing 4.5 *Kennzeichnung eines Feldes als LOB*

```
@Entity
public class EntityWithLOB {
  @Id int id;

  @Lob
  String langerString;
}
```

`@Lob` wird oft im Zusammenhang mit `@Basic` verwendet, um ein Lazy Loading für das Large Object zu definieren.

@Temporal

@Temporal

Die Präzision von Datums- und Zeitwerten (`java.util.Calendar` bzw. `java.util.Date`) ist für die Speicherung in der Datenbank nicht vordefiniert. Für Spalten, die einen solchen Wert repräsentieren, muss die entsprechende Präzision via `@Temporal`-Annotation eingestellt werden.

TemporalType

Hier gibt es drei Alternativen, die alle in der Aufzählung vom Typ `javax.persistence.TemporalType` enthalten sind und der Annotation als Parameter mitgegeben werden:

- DATE: Dieser Wert gibt an, dass nur das Datum gespeichert werden soll (entspricht `java.sql.Date`).
- TIME: Dieser Wert gibt an, dass nur die Uhrzeit gespeichert werden soll (entspricht `java.sql.Time`).
- TIMESTAMP: Dieser Wert gibt an, dass Datum und Uhrzeit gespeichert werden sollen, und ist auch die Voreinstellung (entspricht `java.sql.Timestamp`).

Listing 4.6 *Genauigkeit von Datums- und Zeitwerten setzen*

```
...
@Temporal(DATE)
java.sql.Date startDatum;
...
```

@Version

Soll in einem Entity Bean *Optimistic Locking* zum Einsatz kommen (siehe Abschnitt 4.7.8), benötigt der Entity Manager die Möglichkeit, beim Speichern von Änderungen zu prüfen, ob sich der Zustand in der Datenbank mittlerweile geändert hat. Um dies zu bewerkstelligen, muss die `@Version`-Annotation an einem Feld angebracht werden, um dieses Feld als Versionsspalte zu kennzeichnen. Dieses Feld darf von der Applikation selbst nicht verändert werden und dient lediglich dazu, dem Persistenz-Provider die richtige Spalte in der Datenbank anzugeben. Ist eine Entity Bean über mehrere Tabellen verteilt, muss die Versionsspalte in der primären Tabelle abgelegt werden.

@Version

Die `@Version`-Annotation hat selbst keine Parameter. Weitere Eigenschaften der Versionsspalte müssen über andere Annotationen, wie beispielsweise `@Column`, angegeben werden.

Listing 4.7
Anlegen einer Versionsspalte für Optimistic Locking

```
@Entity
public class AnEntity {
  @Id int theId;

  @Version
  @Column(name="o_version")
  private int version;
}
```

Da die Version eines Entity Beans bei jeder Veränderung hochgezählt wird (Ausnahme hierzu sind Bulk Updates, siehe 4.8.3 und 5.3), dürfen nur einige wenige Java-Typen für die Versionsspalte zum Einsatz kommen: `int`, `Integer`, `short`, `Short`, `Timestamp`, `long`, `Long`. Dies sollte allerdings in der Praxis auch nie zu einem Problem führen.

4.2 Primärschlüssel

In JPA können Primärschlüssel sich über eine oder mehrere Datenbankspalten und damit Felder in der Entity-Bean-Klasse erstrecken. Betrachten wir hier zunächst die »einfachen« Primärschlüssel.

Als Datentypen sind die primitiven Datentypen (`int`, `float`, ...), ihre Objekt-Pendants (`Integer`, `Float`, ...), `String` und die beiden Datumstypen `java.util.Date` und `java.sql.Date` erlaubt. Es ist zwar erlaubt, Typen wie `float` für Primärschlüssel zu verwenden, dies erweist sich aber in der Praxis wegen der Ungenauigkeit der Darstellung oftmals als problematisch. Für Primärschlüssel, die sich über mehrere Spalten erstrecken, kommen extra definierte Klassen zum Einsatz. Dies wird weiter unten beschrieben.

Der Primärschlüssel muss im Falle der Vererbung immer in der Wurzel der Vererbungshierarchie definiert werden. Alternativ kann er auch an einer *Mapped Superclass* der Hierarchie angebracht werden.

Primärschlüsselspalten werden via `@Id` gekennzeichnet, und die Werte dürfen nach der Erzeugung nicht mehr verändert werden. Deshalb ist es eine gute Idee, den zugehörigen Setter als `private` zu deklarieren.

Wird keine zusätzliche `@GeneratedValue`-Annotation zur automatischen Erzeugung von Primärschlüsseln angebracht (s.u.), muss sich die Applikation selbst um die Erzeugung der Primärschlüssel kümmern.

4.2.1 Bemerkung zum Objektvergleich

Dieser Abschnitt beschreibt die Unterschiede der Objektgleichheit zwischen der Java- und der Datenbank-Welt.

In Java wird bei Vergleichen zwischen Objekten zwischen den beiden Operatoren `==` und `equals()` unterschieden.

`==` Der Operator `==` liefert dann `true` zurück, wenn die beiden verglichenen Objekte an derselben Stelle im Speicher stehen. Die beiden Objekte sind *identisch*.

Oftmals möchte man allerdings zwei Objekte vergleichen, die nicht an der gleichen Speicherstelle stehen und wissen, ob sie gleich sind.

`equals()` Hierzu gibt es die Methode `equals()`. Diese Methode stammt aus der Klasse `java.lang.Object`, wo sie über die Identität abgebildet wird.

Im Fall der Persistenzabbildung möchte man die Gleichheit im Normalfall so abbilden, dass zwei Instanzen einer Klasse dann gleich sind, wenn sie denselben Wert für den Primärschlüssel tragen. In diesem Fall muss die Methode `equals()` vom Benutzer überschrieben werden. Möchte man diese Objekte in einer `HashMap` speichern, muss die Methode `hashCode()` ebenfalls überschrieben werden. Es ist sowieso gute Praxis, diese Methode zu überschreiben, wenn `equals()` überschrieben wird.

Fallstricke

Das Erstellen von `equals()` und `hashCode()` ist allerdings etwas trickreich: Generierte Primärschlüssel verletzen die Unveränderlichkeit von `hashCode()`, falls sie zur Berechnung herangezogen werden, da sie erst beim Persistieren generiert werden; der Hashcode ändert sich also. Dasselbe Problem existiert für Relationen, die zur Ermittlung des Hashcodes herangezogen werden. Ändert sich die Relation, stimmt der errechnete Wert nicht mehr mit dem vorherigen überein. Bei der Nutzung von Collections besteht zusätzlich die Gefahr, dass für die Berechnung

die Collection geladen wird, selbst wenn die Relation für das Lazy Loading markiert wurde, so dass am Ende unter Umständen ein komplettes Objektgeflecht aus der Datenbank geladen wird.

Gut ist es also, `equals()` und `hashCode()` mit der Hilfe von unveränderlichen Nutzdaten zu implementieren. In einer Entity, welche einen Benutzer mit Passwort modelliert, wäre der Loginname ein guter Kandidat. Das Passwort hingegen ist kein Kandidat, da der Benutzer in der Lage sein sollte, sein Passwort zu ändern. Generell sind Werte, die in Spalten stehen, welche als *unique* markiert wurden, für die Implementierung von `equals()` und `hashCode()` gute Kandidaten.

4.2.2 Einfacher Primärschlüssel

Für einen einfachen Primärschlüssel reicht es aus, das zu verwendende Feld oder den zu verwendenden Getter mit `@Id` zu markieren. Diese Annotation kennt keine weiteren Parameter.

@Id

Listing 4.8 *Einfacher Primärschlüssel*

```
@Entity
public class Hund {
  String name;

  @Id
  public String getName() {
    return name;
  }
  private void setName(String pk) {
    name = pk;
  }
  ...
}
```

Da `@Id` die einzige Pflicht-Annotation in einer Entity-Hierarchie ist, bestimmt der Ort dieser Annotation (Feld oder Getter), die Art des Zugriffs auf die Entity-Daten durch den Entity Manager. Siehe auch Abschnitt 4.1.

4.2.3 Zusammengesetzter Primärschlüssel

Zusammengesetzte Primärschlüssel lassen sich auf zweierlei Arten kennzeichnen, die nachfolgend beschrieben werden. Der Unterschied besteht darin, dass im einen Fall beliebige Felder der Entity Bean als Primärschlüssel genommen werden können. Im andern Fall kann eine eingebettete Klasse als Primärschlüssel dienen.

In jedem Fall muss die Primärschlüsselklasse serialisierbar sein, einen öffentlichen Default-Konstruktor besitzen und die Methoden `equals()` sowie `hashCode()` implementieren.

Beliebige Felder als Primärschlüssel

@IdClass

Bei zusammengesetzten Primärschlüsseln muss zusätzlich zu den einzelnen Spalten noch eine Klassen-Annotation `@IdClass` angegeben werden, in welcher der Name der Primärschlüsselklasse angegeben ist. In der `@IdClass` ist es durchaus zulässig, nur eine einzige Spalte für diesen zusammengesetzten Primärschlüssel zu verwenden.

Listing 4.9
Beispiel für @IdClass

```
@Entity
@IdClass(class = MyPK.class)
public class Zusammengesetzt {
  @Id int arg1;
  @Id int arg2;
  ...
}

public class MyPK implements Serializable{
  int arg1;
  int arg2;

  boolean equals(Object other) { ... }
  int hashCode() { ... }
}
```

Die Klasse des zusammengesetzten Primärschlüssels muss dabei serialisierbar sein und sowohl `equals()` als auch `hashCode()` so überschreiben, dass es bei unterschiedlichen Primärschlüsseln zu keinen Kollisionen kommt. Außerdem müssen die Namen und Typen in der Entity-Bean-Klasse und der Primärschlüsselklasse übereinstimmen.

Da bei der Abbildung der zusammengesetzten Primärschlüssel über `@IdClass` die Felder des Schlüssels im Entity Bean vorhanden sind, kann auf diese Felder in Abfrage direkt zugegriffen werden. Dies ist besonders interessant, wenn man existierende EJB-2.x-Finder als Abfrage in JP-QL ohne Änderung weiterverwenden möchte.

Eingebettete Klasse als Primärschlüssel

@EmbeddedId

Ist bereits eine Klasse für den Primärschlüssel als eingebettete Klasse[1] definiert (siehe auch 4.4), kann dieser Typ über `@EmbeddedId` auch als Primärschlüssel angegeben werden.

[1] Hibernate spricht an dieser Stelle von Komponenten.

Listing 4.10 Beispiel für die Verwendung von @EmbeddedId

```
@Entity
public class Zusammengesetzt {
  @EmbeddedId
  MyPK thePK;
  ...
}

public class MyPK {
  int arg1;
  int arg2;

  boolean equals(Object other) { ... }
  int hashCode() { ... }
}
```

Der Zugriff auf den Primärschlüssel erfolgt hier also auf Ebene der zusammengesetzten Klasse und nicht auf Ebene der einzelnen Elemente.

4.2.4 Erzeugen von Primärschlüsseln durch den Container

In sehr vielen Umgebungen ist es wünschenswert, Primärschlüssel nicht durch die Applikation, sondern über andere Mechanismen erzeugen zu lassen. Dies kann zum einen sein, dass man sich keine Gedanken über die Eindeutigkeit machen möchte. Oder aber auch, dass man hier einfach die Möglichkeiten der verwendeten Datenbank nutzen möchte. Zu beachten ist, dass bei generierten Schlüsseln lediglich ganzzahlige Typen portabel sind.

@GeneratedValue

Über die Annotation `@GeneratedValue` an einem `@Id`-Element kann die Art der (automatischen) Erzeugung der Primärschlüssel angegeben werden. Diese `@GeneratedValue`-Markierung kennt dabei zwei Parameter, *strategy* und *generator*.

GenerationType

Die möglichen Werte für den optionalen Parameter *strategy* sind vom Typ `GenerationType` und sind:

- AUTO: Dieser Wert besagt, dass der Persistenz-Provider sich die Strategie für die Erzeugung der Primärschlüssel frei wählen kann. Dies ist die Voreinstellung.
- IDENTITY: Hier wird eine *Identity*-Spalte in der Datenbank für die Erzeugung der Primärschlüssel gewählt.
- SEQUENCE: Mit diesem Wert wird eine Sequenzspalte in der Datenbank gewählt.
- TABLE: Dieser Wert weist den Container an, eine Datenbanktabelle für die Erzeugung von eindeutigen Primärschlüsseln zu verwenden.

Die Auswahl von AUTO garantiert die beste Portabilität zwischen unterschiedlichen Datenbanktypen, da der Container sich um die Auswahl des richtigen Generators kümmert. Ist klar, welche Datenbank immer zum Einsatz kommen wird, können die anderen Typen unter Umständen eine bessere Performance bieten, da sich ihre Parameter anpassen lassen. Zuvor jedoch erstmal ein Beispiel für die Verwendung von `@Id` zusammen mit `@GeneratedValue`:

Listing 4.11 *Verwendung von @GeneratedValue*

```
@Entity
public class MyEntity {
  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  Long idColumn;
}
```

Über den String-Parameter *generator* kann der mit der gewählten Strategie zu verwendende Primärschlüsselgenerator benannt werden. Standardmäßig ist dieser Wert leer. Der hier angegebene Name des Generators bezieht sich dabei auf einen Generator, der über `@SequenceGenerator` (für `GenerationType.SEQUENCE`) beziehungsweise `@TableGenerator` (für `GenerationType.TABLE`) definiert wurde.

Tabelle 4.1 zeigt einige Datenbanken mit den von ihnen genutzten Generatoren. Letztlich wird man im Normalfall den Generator vom Persistenzsystem auswählen lassen, um die Anwendung zwischen Datenbanken portabel zu halten.

Tabelle 4.1 *Einige Datenbanken und ihre Generatoren*

Datenbank	Generator
DB/2	IDENTITY, SEQUENCE
Hsqldb	IDENTITY, SEQUENCE (versionsabhängig)
MySQL	IDENTITY
Oracle	SEQUENCE
PostgreSQL	IDENTITY, SEQUENCE
Sybase	IDENTITY

Der Sequenz-Generator

@SequenceGenerator

Die Parameter für die Generierung der Primärschlüssel über einen Sequenz-Generator können via `@SequenceGenerator` bestimmt werden. Diese Annotation kennt vier Parameter, wobei nur *name* erforderlich ist.

- allocationSize: Wie viele Werte sollen vom Generator auf einmal gelesen werden. Der Default steht hier auf 50. Ein größerer Wert verringert die Anzahl der Round-Trips zur Datenbank, hinterlässt aber eventuell Löcher in der Zählung.
- initialValue: Der Startwert für die Generierung. Dieser ist üblicherweise auf 1 gesetzt.
- name: Der Name des Generators. Dieser muss angegeben werden und dient zur Referenzierung in `@GeneratedValue`-Annotationen.
- sequenceName: Der Name des Datenbanksequenz-Generators, der die Primärschlüssel liefern soll. Als Standard wird ein Name vom Persistenz-Provider gewählt.

Listing 4.12
Verwendung des Sequenz-Generators

```
@Entity
@SequenceGenerator(name="mySeq", initialValue="1000")
public class EntityWithSeq {
  @Id
  @GeneratedValue(strategy=SEQUENCE, generator="mySeq")
  private int id;
  ...
}
```

Der Tabellen-Generator

@TableGenerator

Soll für die Erzeugung des Primärschlüssels eine eigene unterliegende Datenbanktabelle verwendet werden, kann diese Tabelle mit ihren Eigenschaften via `@TableGenerator`-Annotation definiert werden. Diese Annotation kennt die folgenden Parameter, die alle bis auf *name* optional sind:

- allocationSize: Die Anzahl der Werte, die vom Generator auf einmal gelesen werden sollen. Der Default steht hier auf 50. Ein größerer Wert verringert die Anzahl der Round-Trips zur Datenbank.
- catalog: Der zu verwendende Datenbankkatalog.
- initialValue: Der Startwert für die Generierung. Dieser ist standardmäßig auf 0 gesetzt.
- name: Der Name des Generators. Dieser muss angegeben werden und dient zur Referenzierung in `@GeneratedValue`-Annotationen.
- pkColumnName: Der Name der Primärschlüssel-Spalte dieser Generatortabelle.
- pkColumnValue: Ein Wert, um diese generierten Primärschlüssel von anderen Schlüsseln von anderen Generatoren unterscheiden zu können.

- schema: Das Datenbankschema, in dem die Tabelle für den Generator zu liegen kommt.
- table: Der Name der Tabelle.
- uniqueConstraints: Eindeutigkeitsbedingungen, welche bei der Erzeugung der Tabelle mit angelegt werden sollen. Der Typ ist ein Feld mit `@UniqueConstraint`-Werten, die in Abschnitt 4.1.1 beschrieben sind.
- valueColumnName: Der Name der Spalte, in welcher der letzte generierte Wert abgelegt wird.

4.3 Vererbung und abstrakte Klassen

Im Gegensatz zur starken Anlehnung an das relationale Modell in EJB 2.x wird in JPA die »natürliche« Objektvererbung berücksichtigt. Die Werte einer Datenbanktabelle können dabei durchaus aus mehreren Klassen kommen, die voneinander erben. Entitys dürfen dabei sowohl konkrete als auch abstrakte Klassen sein. Sie können von beiden Arten von Klassen erben und dürfen auch von beiden Arten von Klassen beerbt werden, wobei diese in beiden Fällen nicht notwendigerweise selbst Entity Beans sein müssen.

Entity Beans unterstützen ebenfalls polymorphe Assoziationen und Abfragen.

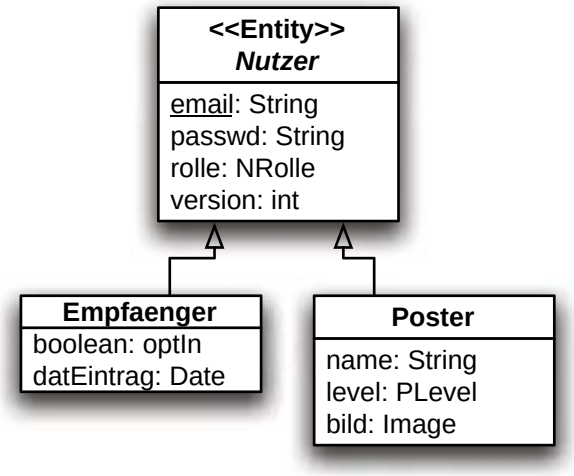


Abbildung 4.1
Vererbung von Entity Beans in EJB 3.0

In Abbildung 4.1 sieht man, wie die beiden Klassen *Empfaenger* und *Poster* von der abstrakten Klasse *Nutzer* erben. In Abfragen können durchaus Objekte von Typ *Nutzer* zurückgegeben werden, allerdings

können keine Objekte vom Typ *Nutzer* instanziiert werden. Wird dies gewünscht, kann die Klasse einfach konkret gemacht werden.

Über `@Inheritance` kann die Strategie für das Mapping zwischen einer Hierarchie von Entity Beans und den zugehörigen Tabellen festgelegt werden. Dabei können alle drei Klassen in einer Tabelle oder in verschiedenen Tabellen abgebildet werden. `@Inheritance` kennt nur das Attribut *strategy* vom Typ `InheritanceType`, das folgende drei Werte annehmen kann. Diese Werte beschreiben die möglichen Mappings der Objektvererbung zu den relationalen Tabellen und werden nachfolgend erläutert:

@Inheritance
InheritanceType

- SINGLE_TABLE (Standard)
- TABLE_PER_CLASS)
- JOINED)

`@Inheritance` muss nur an der Wurzel der Vererbungshierarchie angegeben werden.

Zusätzlich ist es noch möglich, eine Klasse in eine andere Klasse einzubetten. Dies hat zwar auf den ersten Blick nichts mit Vererbung zu tun. In der Praxis ist es aber oft so, dass man Objekte für technische Daten modelliert (Zeitraum der Gültigkeit, letzter Bearbeiter etc.) und sich dann entscheiden muss, ob diese Daten in der Vererbungshierarchie als Superklasse eingeführt werden. Alternativ dazu können diese technischen Daten auch in einer eigenen Klasse hinterlegt werden, werden aber in den Tabellen der »natürlichen« Vererbungshierarchie mit aufgenommen. Siehe auch Abschnitt 4.4.

Abfragen in JP-QL können dabei auf jede der Klassen durchgeführt werden. Abfragen gegen Klasse *Nutzer* können dabei durchaus Objekte vom Typ *Empfaenger* oder *Poster* liefern (polymorphe Abfragen). Damit spielt das Mapping zwischen Klassen und Tabellen bei der Vererbung eine große Rolle für die Performanz einer Anwendung.

4.3.1 Eine Tabelle pro Hierarchie: SINGLE_TABLE

Hier werden alle Attribute der Basis- und aller Subklassen in einer Tabelle abgebildet. Zusätzlich zu den Attributen wird noch eine sog. *Discriminator*-Spalte benötigt, die es erlaubt, die einzelnen Ausgangsklassen in den Tabellenzeilen auseinanderzuhalten. Diese Spalte kann vom System automatisch bereitgestellt werden.

Alternativ kann sie via `@DiscriminatorColumn` angegeben werden. Ist diese Annotation nicht gegeben, wird standardmäßig eine Spalte namens *DTYPE* vom Typ *STRING* verwendet, wobei die Spalte 31 Zeichen breit ist. `@DiscriminatorColumn` kennt diese Attribute:

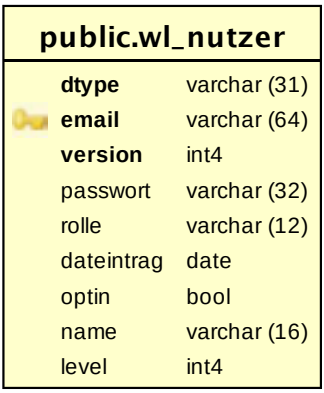
@DiscriminatorColumn

DiscriminatorType

- name: Der Name der Diskriminatorspalte. Vorbelegt ist *DTYPE*.
- discriminatorType: Dieser Wert vom Typ `DiscriminatorType` kann `STRING`, `CHAR` oder `INTEGER` sein und gibt den Spaltentyp an.
- columnDefinition: Ein DDL-Fragment zum Anlegen der Spalte.
- length: Die Spaltenbreite, falls der discriminatorType *STRING* ist.

Für die Klassenhierarchie aus Abbildung 4.1 wären also die Attribute aller drei Klassen in einer Tabelle abgelegt. Zusätzlich kommt noch der (automatisch generierte) Diskriminator *dtype* hinzu.

Abbildung 4.2
Eine Tabelle pro Hierarchie



Der Code für die Basisklasse *Nutzer* würde so aussehen:

Listing 4.13
Beispielcode für SINGLE_TABLE

```
@Entity
@Table(name="WL_NUTZER")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
public abstract class Nutzer implements Serializable {
  // ..
```

Für die Subklassen sieht es dann so aus (*Empfaenger* ist analog zu *Poster* und wird hier nicht extra gezeigt):

Listing 4.14
Beispiel für SINGLE_TABLE in den Subklassen

```
@Entity
public class Poster extends Nutzer
    implements Serializable
{
```

@DiscriminatorValue

Über die optionale Annotation `@DiscriminatorValue` könnte der Wert des Diskriminators für die jeweilige Klasse innerhalb der Hierarchie explizit angegeben werden. Der Parameter *value* ist dabei vom Typ String und gibt den Wert in der Spalte an. Ist die Annotation nicht präsent und ist der Typ des Diskriminators `STRING` oder nicht vorhanden, wird der unqualifizierte Klassenname verwendet. Letztlich bedeutet dies, dass es

in allen Klassen ausreicht, die `@Entity`-Annotation anzugeben. Im Beispiel werden die Werte *Poster* und *Empfaenger* in die Spalte *dtype* geschrieben. Der Wert *Nutzer* kommt nicht vor, da die Klasse *Nutzer* abstrakt ist.

Diese Vererbungsstrategie bietet eine gute Unterstützung für Abfragen, die Objekte aus den Subklassen zurückliefern, da eine Abfrage auf eine Klasse nur einen Datenbankzugriff benötigt. Allerdings müssen die Spalten, die aus den Subklassen kommen, nullbar sein, da ja nicht immer alle Felder bzw. Spalten der Subklassen gefüllt werden. So umgesetzte Hierarchien entsprechen auch nicht der gewünschten dritten Normalform.

Dafür reicht ein Select aus, um ein Objekt zu lesen, was in einer sehr guten Performanz resultiert.

4.3.2 Eine Tabelle pro konkreter Klasse: TABLE_PER_CLASS

Hierbei wird pro konkreter Klasse im Vererbungsbaum eine Tabelle angelegt, in der alle Attribute dieser Klasse sowie alle Attribute der Superklassen abgelegt werden.

Diese Strategie hat den großen Nachteil, dass JP-QL-Abfragen, die über mehrere Subklassen gehen, Unions in SQL oder mehrere SQL-Abfragen benötigen, was nachteilig für die Performance ist.

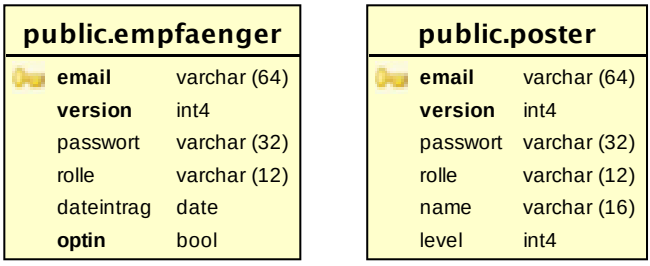


Abbildung 4.3 *Zusammengefügte Subklassen*

Der Code für diese Vererbungsstrategie kann für die Wurzelklasse der Hierarchie wie in Listing 4.15 aussehen.

```
@Entity
@Table(name="WL_NUTZER")
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public abstract class Nutzer implements Serializable {
  // ..
```

Listing 4.15 *Beispiel für TABLE_PER_CLASS*

In den Subklassen kann `@Inheritance` auch hier wieder weggelassen werden, wie dies bereits in Listing 4.14 gezeigt wurde.

Da an den beiden Subklassen *Empfaenger* und *Poster* keine `@Table`-Annotation den Tabellennamen explizit vorgibt, wird dieser Name aus den Klassennamen gebildet. Aus diesem Grund fehlt in Listing 4.15 auch das Präfix *wl_* an den Tabellennamen.

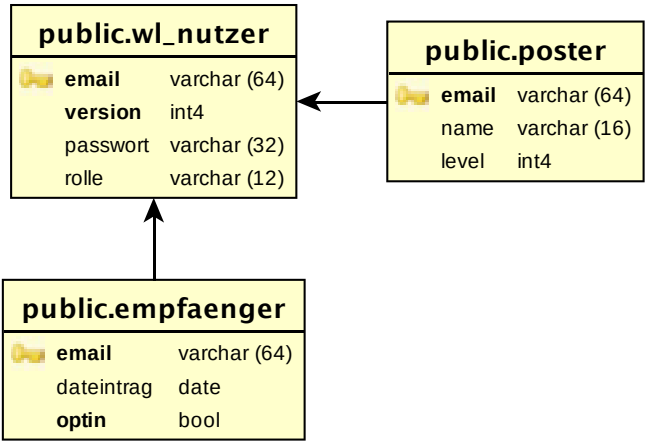
Nachteil dieses Vorgehens ist, dass die Attribute der Basisklasse redundant als Spalten in den Tabellen der Subklassen angelegt werden. Vorteilhaft ist dagegen, dass eine nichtpolymorphe Abfrage auf eine Klasse nur einen Zugriff benötigt. Polymorphe Abfragen müssen allerdings unter Umständen alle so angelegten Tabellen durchsuchen (im Beispiel nur zwei, bei etwas größeren Hierarchien steigt der Aufwand allerdings deutlich).

4.3.3 Eine Tabelle pro Subklasse: JOINED

Bei dieser Strategie hält eine Tabelle die Attribute der Basisklasse. Pro Subklasse gibt es dann eine weitere Tabelle, welche nur die Attribute dieser Subklasse hält.

Im Beispiel gibt es also drei Tabellen, wobei die Basistabelle für *Nutzer* die Attribute dieser Klasse enthält. Die beiden anderen Tabellen enthalten den Primärschlüssel der Tabelle *WL_Nutzer* sowie die Felder der Klassen *Empfaenger* und *Poster*.

Abbildung 4.4
Eine Tabelle pro Klasse



Der Code für die Klasse *Nutzer* sieht bei dieser Strategie so aus:

Listing 4.16
Setup der Basisklasse für die JOINED-Strategie

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
@Table( name = "WL_Nutzer")
public abstract class Nutzer implements Serializable {
  // ..
```

Für die beiden Subklassen sieht der Code wieder wie in Listing 4.14 gezeigt aus. Da auch hier keine `@Table`-Markierung an den Klassen angebracht wurde, werden die Namen der beiden Tabellen für *Poster* und *Empfaenger* aus dem Klassennamen gebildet.

Der Vorteil dieser Abbildungsstrategie ist, dass hier die gewünschte dritte Normalform erreicht wird. Die Performanz ist nicht ganz so hoch wie bei `SINGLE_TABLE`, dafür können alle Tabellen mit `not null`-Bedingungen belegt werden.

Für das Mapping von Relationen mit dieser Strategie kann die Annotation `@PrimaryKeyJoinColumn` verwendet werden, die im nächsten Abschnitt beschrieben wird. Diese wird ebenfalls benötigt, wenn der Primärschlüssel der Subklasse ein anderer ist als der der Superklasse (z.B. aus bestehenden Schemata).

4.3.4 PrimaryKeyJoinColumn

Normalerweise wird bei der *Joined Subclass*-Vererbungsstrategie davon ausgegangen, dass die Fremdschlüsselspalten der Subklasse dieselben Namen haben wie die Primärschlüsselspalten der (primären) Tabelle der Superklasse. Trifft dies nicht zu, kann die andere Anordnung über `@PrimaryKeyJoinColumn` konfiguriert werden, wobei die Annotation drei optionale Parameter hat:

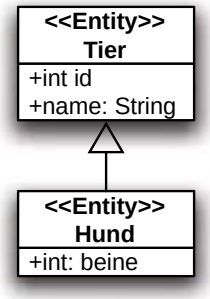
@PrimaryKeyJoinColumn

- name: Name der Primärschlüsselspalte der aktuellen Tabelle. Ist dieser Wert nicht gegeben, wird der Name der Primärschlüsselspalte der Superklasse verwendet.
- columnDefinition: DDL-Fragment, das zum Erzeugen dieser Spalte herangezogen werden soll.
- referencedColumnName: Der Name der Primärschlüsselspalte der Tabelle, auf die der Join durchgeführt werden soll. Ist dieser Wert nicht gegeben, wird der Name der Primärschlüsselspalte der Superklasse verwendet.

Ein Beispiel verdeutlicht die Verwendung (siehe Listing 4.17).

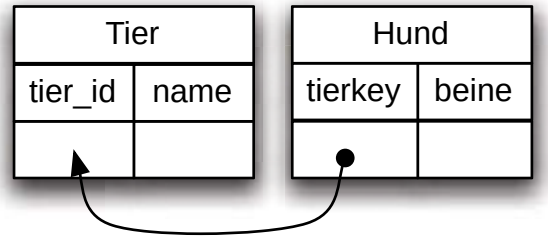
Listing 4.17 *Beispielklassen für PrimaryJoinColumn*

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class Tier
{
  @Id
  @Column(name="tier_id")
  int id;
  String name;
}
```




```
@Entity
@PrimaryKeyJoinColumn(name="tierKey",
    referencedColumnName="tier_id")
public class Hund extends Tier
{
  int anzahlBeine;
}
```

In der Subklasse *Hund* wird also über `@PrimaryKeyJoinColumn` definiert, dass der implizit für diese Tabelle generierte Primärschlüssel den Namen *tierKey* haben soll und dass dieser als Fremdschlüssel auf die Spalte *tier_id* der Tabelle *Tier* zeigen soll. Letztere Spalte bildet den Primärschlüssel dieser Klasse ab, der durch das Feld *id* definiert wurde. Damit ergibt sich folgendes Bild in der Datenbank:



Weitere Anwendungsgebiete dieser Annotation sind:

- Join einer zweiten Tabelle zur Primärtabelle einer Entity
- Join der beiden Tabellen in einer `@OneToOne`-Beziehung

@PrimaryKeyJoinColumns

Da auch hier die Annotation wieder nur einmal an einer Klasse stehen darf, können mehrere `@PrimaryKeyJoinColumn` über `@PrimaryKeyJoinColumns` zusammengefasst werden.

4.3.5 Einbetten von gemeinsamen Daten über eine Mapped Superclass

@MappedSuperclass

In einigen Fällen ist es notwendig, gemeinsame technische Attribute oder Felder der Geschäftslogik in einer Superklasse für alle zu persistierenden Objekte bereitzustellen, ohne dass diese Superklasse selbst in einer (eigenen) Tabelle gespeichert wird. Eine solche Klasse wird über `@MappedSuperclass` gekennzeichnet. Diese Annotation auf Klassenebene hat keine Attribute. Klassen, die als `@MappedSuperclass` gekennzeichnet wurden, dürfen nicht als Argumente für Operationen des Entity Managers oder der Schnittstelle `Query` verwendet werden und können auch nicht das Ziel einer persistenten Beziehung von anderen Entitys aus

sein. Sie kann aber Beziehungen haben, die von ihr ausgehen. Dies ist auch der große Unterschied zu einer simplen abstrakten Superklasse.

Mappings für Felder erfolgen genau wie bei den »normalen« Entity-Klassen auch und können genauso über `@AttributeOverride` in den Subklassen überschrieben werden. Für die Assoziationen zu anderen Klassen können die Mappings in den erbenden Klassen über `@AssociationOverride` angepasst werden. Diese beiden Mechanismen werden in Abschnitt 4.5 ab Seite 94 beschrieben.

4.4 Eingebettete Objekte

Oftmals hat man Informationen in zwei Klassen, die gemeinsam persistiert werden sollen. Dies kann über eine Eins-zu-eins-Beziehung erfolgen, wie sie weiter unten (Abschnitt 4.6.2) gezeigt wird. Diese Beziehung bedeutet aber immer einen Join auf der Datenbank. Als Beispiel soll eine Person mit einer Adresse herhalten, wie sie in Abbildung 4.6 auf Seite 98 gezeigt wird.

Hier ist es oftmals besser, die Adressdaten mit in die Tabelle für die Klasse Person aufzunehmen. Auf der Java-Seite sind es weiterhin zwei Klassen, die aber dann in eine Tabelle abgebildet werden. Die so eingebetteten Felder werden immer zusammen mit denen der aufnehmenden Klasse geladen, ein Lazy Loading ist nicht vorgesehen.

Für diese Funktionalität sind in beiden Klassen zusätzliche Annotationen notwendig. Die Klasse, welche eingebettet werden soll, erhält eine `@Embeddable`-Markierung auf Klassenebene. Diese Annotation hat keine Parameter.

@Embeddable

Wenngleich es von der Spezifikation nicht explizit gefordert wird, ist es trotzdem gut, die einzubettende Klasse die Schnittstelle *Serializable* implementieren zu lassen, da die aufnehmende Klasse ja eventuell an Clients weitergereicht wird und damit auch alle ihre Felder serialisierbar sein müssen.

In der aufnehmenden Klasse wird eine `@Embedded`-Annotation an einem Getter oder Feld mit dem entsprechenden Typ angebracht, um zu kennzeichnen, dass dieser Typ nicht als Objekt, sondern eingebettet verwendet werden soll. Auch diese Annotation hat keine Parameter.

@Embedded

Da eingebettete Klassen teilweise an verschiedenen Stellen verwendet werden, besteht die Möglichkeit, über `@AttributeOverrides` die Definitionen für die eingebetteten Spalten von der Hauptklasse aus zu überschreiben, um so beispielsweise die Spaltennamen entsprechend zu ändern. Dies wird im nächsten Abschnitt näher beschrieben.

Listing 4.18 *Ein Beispiel für eingebettete Klassen*

```
// Diese Klasse wird eingebettet
@Embeddable
public class Zeitspanne implements Serializable {
  @Temporal(DATE) Date start;
  @Temporal(DATE) Date end;
}

// Aufnehmendes Entity Bean
@Entity
public class Mitgliedschaft {
  ...
  @Embedded
  Zeitspanne zeitRaum;
  ...
}
```

Für die eingebettete Komponente sind nur die Typ-Annotationen aus Abschnitt 4.1.3 erlaubt. Des Weiteren müssen Implementierungen keine eingebetteten Objekte unterstützen, die sich über mehrere Tabellen erstrecken. Genauso ist aktuell nur eine Ebene von eingebetteten Klassen durch die JPA unterstützt.

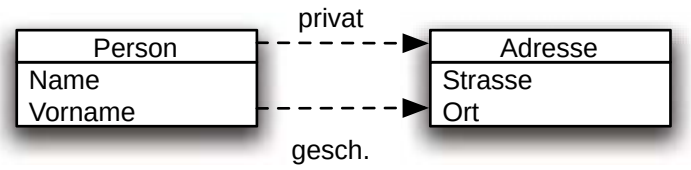
4.5 Überschreiben von Persistenzinformationen

Speziell wenn man Klassen in andere Klassen einbettet, kommt es vor, dass die Spaltennamen der eingebetteten Klasse mit denen der aufnehmenden Klasse kollidieren.

4.5.1 Überschreiben von Attributen

In Abbildung 4.5 wäre es beispielsweise so, dass das Objekt *Adresse* als Privat- und als Geschäftsadresse zweimal eingebettet wird, so dass es zwei Spalten *Ort* geben würde. Man könnte diesen Konflikt lösen, indem man statt der generischen Klasse *Adresse* auch spezielle Klassen für die beiden Arten der Adresse anlegen würde, welche jeweils eindeutige Namen für die Attribute bzw. Spalten hätten.

Abbildung 4.5 *Überschreiben von Attributinformationen*



Einen besseren Weg, um diesen Konflikt zu lösen, ist über `@AttributeOverride` ein Remapping der Felder durchzuführen. Die Annotation kennt diese beiden Pflichtattribute:

@AttributeOverride

- ❏ name: Der Name des Feldes in der einzubettenden Klasse, der umgemappt werden soll.
- ❏ column: Die Spalte, in welche das Feld abgebildet werden soll. Diese wird über `@Column` beschrieben (siehe Abschnitt 4.1.3).

Listing 4.19
Beispiel für das Überschreiben von Persistenzinformationen

```
@Embeddable
public class Adresse implements Serializable{
  String strasse;
  String ort;
}

@Entity
public class Person {
  // ...
  @Embedded
  @AttributeOverride(name="strasse",
    column=@Column(name="p_strasse"))
  Adresse privatAdresse;
}
```

Da auch hier `@AttributeOverride` nur einmal an einem Element stehen darf, können mehrere Felder über `@AttributeOverrides` zusammengefasst werden. Das Attribut der Annotation beinhaltet dann eine Liste von `@AttributeOverride`-Annotationen.

@AttributeOverrides

Listing 4.20
Vollständiges Remapping für Abbildung 4.5

```
... // wie oben

@Entity
public class Person {
  // ...
  @Embedded
  @AttributeOverrides({
    @AttributeOverride(name="strasse",
      column=@Column(name="p_strasse")),
    @AttributeOverride(name="ort",
      column=@Column(name="p_ort"))
  })
  Adresse privatAdresse;

  @Embedded
  @AttributeOverrides({
```

```
    @AttributeOverride(name="strasse",
      column=@Column(name="g_strasse")),
    @AttributeOverride(name="ort",
      column=@Column(name="g_ort"))
  })
  Adresse geschaeftsAdresse;
}
```

4.5.2 Überschreiben von Assoziationen

Analog zum Remapping von einzelnen Feldern kann es auch notwendig sein, n:1- und 1:1-Assoziationen umzusetzen. Beispielsweise kann in einer MappedSuperclass eine Relation gegeben sein, bei der dann in der konkreten Entity Bean die eigentliche Join Column (die Fremdschlüsselspalte) überschrieben werden muss. Dies kann über die Annotation `@AssociationOverride` geschehen, welche zwei Parameter kennt, die beide angegeben werden müssen:

@AssociationOverride

- name: Der Name des Feldes oder der Property für die Relation, die überschrieben werden soll.
- joinColumns: Die Spalten, welche für die Abbildung der Beziehung verwendet werden sollen. Der Parameter ist dabei eine Liste von `@JoinColumn`-Attributen. Diese werden in Abschnitt 4.6.9 beschrieben.

Ein Beispiel für die Verwendung könnte so aussehen, wobei Relationen im nächsten Abschnitt beschrieben werden:

Listing 4.21 *Beispiel für das Überschreiben von Assoziationsinformationen*

```
@MappedSuperclass
public class MySuper {

  @OneToOne
  Adresse addr;
}

@Entity
@AssociationOverride(name="addr",
   joinColumns=@JoinColumn(name="a_id")
public class Person extends MySuper {
   ...
}
```

Hier definiert also die Superklasse eine Beziehung zu einer Adresse. Diese wird in der Subklasse *Person* konkretisiert, indem über `@AssociationOverride` festgelegt wird, dass das Feld *addr* mit der Tabelle für die

Adresse durch eine Fremdschlüsselspalte namens *a_id* in dieser Tabelle verknüpft wird.

Auch hier gilt wieder, dass nur eine `@AssociationOverride`-Annotation an einem Element stehen darf. Falls mehr als ein Mapping überschrieben werden soll, kann dies via `@AssociationOverrides` geschehen, wobei das Argument dieser Annotation eine Liste von `@AssociationOverride`-Tags ist.

@AssociationOverrides

4.6 Relationen

Natürlich bietet auch die Java-Persistenz-API Unterstützung für Assoziationen zwischen zwei Klassen – und damit auch für Relationen zwischen zwei Tabellen. Hierbei werden vier Typen unterschieden, die jeweils gerichtet oder ungerichtet sein können.

Die n-Seiten von Relationen werden in den Entity Beans als *Collections* ausgedrückt. Hierbei sind die folgenden vier Typen (und ihre Generics-Varianten) zugelassen. Propertys oder Felder müssen von deren Typ sein, auch wenn das Feld dann mit einem konkreten Typ initialisiert wird.

- `java.util.Collection`
- `java.util.Map`
- `java.util.List`
- `java.util.Set`

Die Generics-Varianten der *Collections* haben den Vorteil in der Nutzung, dass man weniger zusätzliche Informationen für das Mapping der Beziehung bereitstellen muss, die entsprechenden Typinformationen im Quellcode und damit in den generierten Klassen vorhanden sind. Siehe hierzu `java.lang.reflect.GenericDeclaration`.

Speziell bei Listen darf nicht davon ausgegangen werden, dass die Reihenfolge der Listenelemente über Persistenzkontexte (siehe Abschnitt 4.7) eingehalten wird. Um dies sicherzustellen, sollte bei Abfragen eine `@OrderBy`-Klausel mitgegeben werden (siehe Abschnitt 4.6.8 auf Seite 107).

Zu beachten ist auch, dass Relationen uni- oder bidirektional sein können. Dies wird einfach dadurch erreicht, dass nur die »Verbindung« zur anderen Klasse navigierbar ist, die durch eine Annotation gekennzeichnet ist. In der Praxis sind noch einige weitere Dinge zu beachten, die bei den jeweiligen Relationen entsprechend angegeben sind.

Unidirektionale Relationen haben nur ein *führendes* Ende, während bidirektionale Relationen auch ein *inverses* Ende besitzen, welches einen *mappedBy*-Parameter bekommt, der das führende Ende angibt.

Wir wollen uns diese Relationen nun am Beispiel von zwei miteinander verbundenen Klassen ansehen.

Beispielklassen für die Relationen

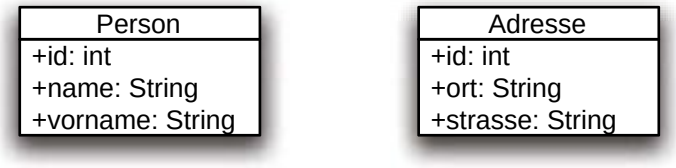


Abbildung 4.6 *Beispielklassen für die Relationen*

Diese beiden Klassen, *Person* und *Adresse*, dienen zur Erläuterung der Relation und bestehen immer aus dem folgenden Rumpf:

Listing 4.22 *Die Klassen* Person *und* Adresse

```
@Entity                          @Entity
public class Person              public class Adresse
{                                {
  @Id public int id;               @Id public int id;
  public String name;              public String ort;
  public String vorname;           public String strasse;
}                                }
```

Dieser Rumpf wird in den nachfolgenden Beispielen nur noch gezeigt, wenn dies für das Verständnis notwendig ist.

4.6.1 Bemerkung zu bidirektionalen Assoziationen

Betrachtet man bidirektionale Beziehungen zwischen zwei Klassen in Java, stellt man fest, dass diese jeweils über ein Feld oder als Collection im Zu-n-Fall, mit dem Typ der anderen Klasse realisiert wird, wie dies in der oberen Hälfte von Abbildung 4.7 zu sehen ist[2]. In der Klasse *Person* ist ein Feld vom Typ *Adresse*, das auf die assoziierte Adresse a1 zeigt. In der Rückrichtung, um von Adresse zu Person navigieren zu können, gilt dasselbe. Auf der Datenbankseite ist dies einfach über eine Fremdschlüsselbeziehung realisiert.

[2] Wir betrachten hier der Einfachheit halber den 1:1-Fall. Für die anderen Fälle gilt das Beschriebene analog.

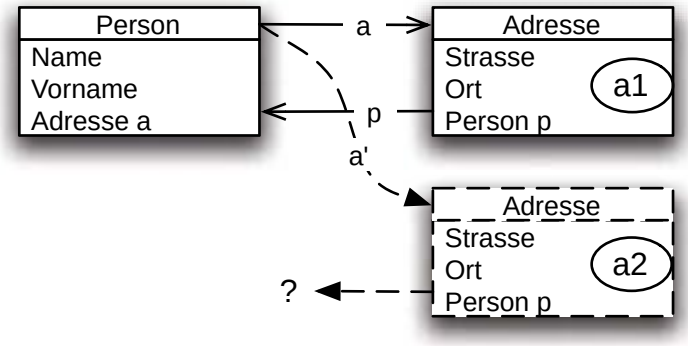


Abbildung 4.7 *Bidirektionale Assoziationen in Java*

Wird nun der Person eine neue Adresse, a2, zugewiesen, wird zwar diese in der Person eingetragen; der Link zeigt nun auf a2. Allerdings wird dies aber nicht automatisch in der Adresse a1 nachgezogen, so dass der Verweis weiterhin auf die Person zeigt. In der neuen Adresse a2 zeigt dafür der Zeiger auf die Person in die Leere – eine inkonsistente Situation entsteht.

Anders als bei EJB 2.1 Entity Beans managt der Container die Relation nicht automatisch auf beiden Seiten, so dass der Entwickler in EJB 3.0 auch die Rückrichtung der Assoziation händisch umsetzen muss. Dies könnte beispielsweise so aussehen:

Listing 4.23 *Rückrichtung der Assoziation setzen*

```
public class Person
{
  Adresse a;
  ...

  void setAdresse(Adresse adr)
  {
     a = adr;
     a.setPerson(this);
  }
}
```

4.6.2 Eins zu eins

Die einfachste Relation ist die 1:1-Relation, bei der eine Klasse aus einer anderen heraus angesprochen wird. In der JPA gibt es hierbei zwei Arten, diese Relation herzustellen, wovon die Variante über eingebettete Klassen bereits im vorherigen Abschnitt (4.4) beschrieben wurde.

Die andere Variante besteht darin, dass pro Klasse eine Tabelle angelegt wird und diese Tabellen über eine Fremdschlüsselspalte auf Datenbankebene verknüpft werden. Hierzu wird die `@OneToOne`-Annotation

@OneToOne

an den Getter oder das Feld für die »andere« Seite angebracht (also an den Typ *Adresse* innerhalb der Klasse *Person* für das Beispiel aus Abbildung 4.6):

Listing 4.24 *OneToOne unidirektional*

```
@Entity
public class Person {
  ...
  @OneToOne(cascade=ALL, optional=false)
  Adresse adresse;
  ...
}
```

In diesem einfachen unidirektionalen Fall wird in der Tabelle *Person* eine Fremdschlüsselspalte vom Container mit dem Namen *adresse_id* angelegt, die auf den Primärschlüssel der *Adresse* zeigt. Der Name der Fremdschlüsselspalte setzt sich also aus den beteiligten Feldern zusammen, die durch einen Unterstrich separiert sind und bei denen das Feld der führenden Klasse vorne steht. Soll ein anderes Mapping verwendet werden, kann dies über `@PrimaryKeyJoinColumn` bewerkstelligt werden.

Die `@OneToOne`-Annotation kennt eine Reihe von optionalen Parametern:

- targetEntity: Die Klasse »auf der anderen Seite«. Diese Angabe wird normalerweise nicht benötigt, da der Typ sich ja über das markierte Feld bzw. den markierten Getter ermitteln lässt.
- cascade: Das Cascading für diese Relation. Es ist vom Typ `CascadeType[]` und wird in Abschnitt 4.6.6 auf Seite 105 beschrieben. Ist das Attribut nicht vorhanden, wird keine Kaskadierung vorgenommen.

Default: EAGER Loading

- fetch: Der Fetch Type für die Assoziation. Dieser wird in Abschnitt 4.6.7 beschrieben und ist standardmäßig auf `EAGER` gesetzt.
- optional: Dieser boolesche Parameter gibt an, ob die Relation einen `null`-Wert als Ziel zulässt. Die Voreinstellung ist `true`.
- mappedBy: Bei bidirektionalen Relationen muss auf der inversen Seite mit diesem Parameter das abbildende Element auf der führenden Seite angegeben werden (also der Name des Feldes oder Parameters). Ein Beispiel hierfür gibt es in Listing 4.25 im nächsten Abschnitt.

Sollen beide Entitäten gemeinsam gelesen werden (also im Beispiel die Daten der Person inklusive ihrer Adresse), ist die Ablage der Daten in zwei Tabellen meist unvorteilhaft, da sie einen Join auf der Datenbank bedeutet. In diesem Fall eignet sich eine eingebettete Klasse oftmals besser. Diese sind in Abschnitt 4.4 beschrieben.

Bidirektionale 1:1-Relation

Soll die Relation auch von der Adresse aus navigierbar (also bidirektional) sein, muss die Adresse auch markiert werden. Die Klasse Person bleibt weiterhin die führende Klasse.

Listing 4.25 *OneToOne bidirektional*

```
@Entity
public class Adresse {
  ...
  @OneToOne(mappedBy="adresse")
  Person person;
  ...
}
```

Innerhalb von `@OneToOne` bezeichnet *adresse* also das Feld auf der führenden Seite der Beziehung. Würde der Parameter *mappedBy* nicht angegeben, würden die beiden mit `@OneToOne` markierten Felder nicht als Teil ein und derselben Relation angesehen werden.

Das Verhalten der Fremdschlüsselspalte ist dasselbe wie bei der unidirektionalen Variante.

4.6.3 Eins zu N

Bei einer 1:n-Relation würden einer Person mehrere Adressen zugewiesen. In den Java-Klassen wird dies über `@OneToMany` realisiert und würde dann im unidirektionalen Fall aussehen, wie in Listing 4.26 gezeigt.

Listing 4.26 *OneToMany unidirektional*

```
@Entity
public class Person {
  ...
  @OneToMany(cascade=ALL)
  Collection<Adresse> adressen;
  ...
}
```

@OneToMany

Die Annotation `@OneToMany` kennt im Prinzip dieselben Parameter wie `@OneToOne`:

- targetEntity: Die Klasse »auf der anderen Seite«. Diese Angabe wird nicht benötigt, wenn der Typ der anderen Seite als Generics-Collection angegeben ist, da er dann zur Laufzeit ermittelt werden kann. Wird eine unqualifizierte Collection verwendet, muss der Typ »der anderen Seite« mit diesem Attribut angegeben werden.
- cascade: Das Cascading für diese Relation. Es ist vom Typ `CascadeType[]` und wird in Abschnitt 4.6.6 beschrieben. Ist das Attribut nicht vorhanden, wird keine Kaskadierung vorgenommen.

Default: LAZY Loading

- fetch: Der Fetch Type für die Assoziation. Dieser wird in Abschnitt 4.6.7 beschrieben und ist standardmäßig auf `LAZY` gesetzt.
- mappedBy: Bei bidirektionalen Relationen muss auf der inversen Seite mit diesem Parameter das mappende Element der führenden Seite angegeben werden.

Auf Datenbankebene erfolgt die Abbildung dieser Beziehung standardmäßig über eine Mappingtabelle. Der Name dieser Tabelle setzt sich dabei aus den Namen der beiden beteiligten Tabellen zusammen, wobei die führende Tabelle zuerst genannt wird und die beiden Tabellennamen durch einen Unterstrich getrennt werden. Im Beispiel wäre dies also *Person_Adresse*. Über `@JoinTable` (siehe Abschnitt 4.6.10) kann die Mappingtabelle parametriert werden.

Persistenz-Provider können unidirektionale 1:n-Beziehungen auch über Fremdschlüssel abbilden. An dieser Stelle empfiehlt der Standard aber, die Relation bidirektional zu machen, um die Portabilität sicherzustellen.

Bidirektionale 1:n-Relation

Bei bidirektionalen 1:n-Beziehungen muss die n-Seite der Beziehung das führende Ende sein. Dies bedeutet, dass eine solche Beziehung letztlich immer als n:1-Beziehung von der anderen Seite her betrachtet wird.

Angenommen eine Person hat viele Adressen, dann würde der Code so aussehen:

Listing 4.27 *OneToMany bidirektional*

```
@Entity
public class Person {
...
  @OneToMany(mappedBy="person")
  Collection<Adresse> adressen;
...
}
```

```
@Entity
public class Adresse {
...
  @ManyToOne
  Person person;
...
}
```

Das Attribut *mappedBy* verweist dabei auf das Feld *person* der Klasse *Adresse*, die in diesem Fall die führende Seite darstellt.

4.6.4 N zu eins

Bei einer n:1-Relation werden mehrere Personen auf dieselbe Adresse abgebildet. Dies geschieht via `@ManyToOne`.

Listing 4.28 ManyToOne unidirektional

```
@Entity
public class Person {
  ...
  @ManyToOne
  Adresse adresse;
  ...
}
```

Hier ist die Klasse Person die führende Klasse, und die Relation wird auf Datenbankebene über einen Fremdschlüssel abgebildet.

@ManyToOne

Die Annotation `@ManyToOne` kennt nachfolgende optionale Parameter, welche bereits bekannt sind:

- targetEntity: Die Klasse »auf der anderen Seite«. Diese Angabe wird normalerweise nicht benötigt, da der Typ sich ja über das markierte Feld bzw. den markierten Getter ermitteln lässt.
- cascade: Das Cascading für diese Relation. Es ist vom Typ `CascadeType[]` und wird in Abschnitt 4.6.6 beschrieben. Ist das Attribut nicht vorhanden, wird keine Kaskadierung vorgenommen.
- fetch: Der Fetch Type für die Assoziation. Dieser wird in Abschnitt 4.6.7 beschrieben und ist standardmäßig auf `EAGER` gesetzt. *Default: EAGER Loading*
- optional: Dieser boolesche Parameter gibt an, ob die Relation einen `null`-Wert als Ziel zulässt. Die Voreinstellung ist `true`.

Bidirektionale n:1-Relation

Bei bidirektionalen n:1- oder 1:n-Beziehungen muss immer die n-Seite die führende Seite sein, weswegen der *mappedBy*-Parameter auf der 1-Seite der Beziehung angebracht sein muss.

Listing 4.29 ManyToOne bidirektional

```
@Entity
public class Person {
...
  @ManyToOne
  Adresse adresse;
...
}

@Entity
public class Adresse {
...
```

```
  @OneToMany(mappedBy="adresse")
  Collection <Person> personen;
...
}
```

Letztlich ist dies genau wie die bidirektionale 1:n-Beziehung von oben, nur dass die beiden beteiligten Klassen die Rollen getauscht haben.

4.6.5 N zu M

Dieser Fall ist der allgemeinste Fall, bei dem einer Person mehrere Adressen zugeordnet werden und umgekehrt an einer Adresse auch mehrere Personen aufgeführt werden. Die Abbildung auf Datenbankebene erfolgt dabei immer über eine Mappingtabelle. Diese wird dabei vom Persistenz-Provider verwaltet. Es gibt dabei keine Möglichkeit, in dieser Tabelle noch zusätzliche Spalten zu verwalten. Wird dies benötigt, muss das Mapping über die Geschäftslogik der Anwendung verwaltet werden.

@ManyToMany

Diese N:M-Beziehungen werden über die Annotation `@ManyToMany` definiert, welche die von `@ManyToOne` bekannten Attribute kennt:

- targetEntity: Die Klasse »auf der anderen Seite«. Diese Angabe wird nicht benötigt, wenn der Typ der anderen Seite als Generics Collection angegeben ist. Wird eine unqualifizierte Collection verwendet, muss der Typ »der anderen Seite« mit diesem Attribut angegeben werden.
- cascade: Das Cascading für diese Relation. Es ist vom Typ `CascadeType[]` und wird in Abschnitt 4.6.6 beschrieben. Ist das Attribut nicht vorhanden, wird keine Kaskadierung vorgenommen.

Default: LAZY Loading

- fetch: Der Fetch Type für die Assoziation. Dieser wird in Abschnitt 4.6.7 beschrieben und ist standardmäßig auf `LAZY` gesetzt.
- mappedBy: Bei bidirektionalen Relationen muss auf der inversen Seite mit diesem Parameter das mappende Element auf der führenden Seite angegeben werden.

Listing 4.30 *ManyToMany bidirektional*

```
@Entity
public class Person {
...
  @ManyToMany(targetEntity=Adresse.class)
  Collection adressen;
...
}
```

```
@Entity
public class Adresse {
...
  @ManyToMany(mappedBy="adresse")
  Collection <Person> personen;
...
}
```

In Listing 4.30 ist die *Person* die führende Klasse der Beziehung. Da die Collection der Adressen keine Generics-Variante ist, wurde die andere Seite über *targetEntity* definiert.

Die angelegte Mappingtabelle trägt in diesem Fall den Namen `Person_Adresse`. Soll dies geändert werden, kann dies über `@JoinTable` geschehen. Diese Annotation wird in Abschnitt 4.6.10 ab Seite 109 beschrieben.

4.6.6 Kaskadieren von Operationen

Bei der Modifikation eines Objektes in einem Objektgeflecht ist es oftmals wünschenswert, nicht nur die Änderungen am Ausgangsobjekt zu persistieren, sondern dies auch auf die restlichen Objekte des Geflechts auszudehnen. Beispielsweise ist es im Fall der UML-Komposition erforderlich, wenn das führende Objekt gelöscht wird, die angehängten Objekte ebenfalls zu löschen. Dieser Vorgang wird auch als *Kaskadierung* bezeichnet

CascadeType

Die folgenden Optionen sind für den Typ `CascadeType` gegeben. Das Verhalten und die entsprechenden Schlüsselwörter korrespondieren dabei mit den entsprechenden Operationen des Entity Managers, die in Abschnitt 4.7 bzw. 4.7.6 beschrieben sind.

- ALL: Alle anderen Optionen zusammengenommen.
- MERGE: Wird das Ausgangsobjekt via `merge()` wieder vom Zustand *detached* in den Zustand *managed* gebracht, werden auch die angehängten Objekte gemerged.
- PERSIST: Wenn beim Anlegen eines Objektnetzes das Ausgangsobjekt gespeichert wird, werden auch die angehängten Objekte mit persistiert.
- REFRESH: Wird ein Objekt aus der Datenbank via `refresh()` aktualisiert, kann hiermit angegeben werden, dass die Objekte auf der anderen Seite der Relation ebenfalls aktualisiert werden sollen.
- REMOVE: Zugehörige Objekte werden gelöscht, wenn das Hauptobjekt gelöscht wird. Dies entspricht einem *CASCADE*

DELETE auf der Datenbank. Diese Option sollte nur auf der führenden Seite von `@OneToOne` und `@OneToMany` verwendet werden.

Hier noch ein Beispiel für die Verwendung aus der Klasse *Weblog*:

Listing 4.31
Beispiel für Cascading

```
@OneToMany(mappedBy="weblog",
    cascade=CascadeType.REMOVE)
public Collection<Artikel> getArtikel()
{
  return artikel;
}
```

Wird das Blog gelöscht, sollen auch alle Artikel, die in diesem Blog geschrieben wurden, gelöscht werden.

Es ist zu beachten, dass das Kaskadieren nicht für die *UPDATE*- und *DELETE*-Operation in JP-QL (siehe Abschnitt 5.3) zutrifft. An dieser Stelle ist es besser, diese Objektnetze explizit über mehrere Aufrufe zu löschen.

Man kann dies auch über ein Cascade delete auf der Datenbank machen. Dies hat allerdings den Nachteil, dass der Entity Manager und ein eventuell vorhandener Cache im Persistenz-Provider nichts von diesem Löschen mitbekommt und somit in nachfolgenden Operationen eventuell ungültige Daten im Cache beziehungsweise dem *Persistence Context* stehen und von dort an die Anwendung ausgegeben werden.

4.6.7 Fetch Type

In Java sind Assoziationen zu anderen Beans immer vorhanden. Ist in einem Objekt beispielsweise eine Liste von anderen Objekten enthalten, kann mit diesen anderen Objekten direkt interagiert werden.

Sind diese Objekte nun als Relation in der Datenbank hinterlegt, stellt sich die Frage, wann diese Objekte aus der Datenbank geladen werden. Hier bietet der Typ `FetchType` zwei Möglichkeiten, die beide als Parameter der entsprechenden Relations-Annotation und von `@Basic` (siehe Abschnitt 4.1.3) verwendet werden können.

FetchType

- EAGER: Hierbei werden die abhängigen Objekte direkt mit dem Originalobjekt aus der Datenbank geladen. Der Vorteil ist, dass die abhängigen Objekte direkt navigierbar sind. Dies ist attraktiv, wenn das Objekt z.B. an eine JSP-Seite zur Ausgabe gegeben werden soll und die angehängten Objekte dort ausgelesen werden sollen (*detached*-Betrieb). Der Nachteil ist, dass hier teilweise Objekte von der Datenbank geladen werden, die später nicht verwendet werden.

- LAZY: Mit dieser Option werden abhängige Objekte erst geladen, wenn sie angesprochen werden. Hierzu darf das Ursprungsobjekt nicht *detached* sein. Der Vorteil ist, dass unter Umständen weniger Daten aus der Datenbank geladen werden müssen. Dies erkauft man sich durch eine größere Latenz für spätere Zugriffe auf abhängige Objekte, da diese durch einen zusätzlichen Datenbankzugriff erst nachgeladen werden müssen. Objekte, die *detached* sind, müssen ebenfalls erst wieder in den Zustand *managed* gebracht werden. Dies wird in Abschnitt 4.7.4 beschrieben.
  Eine weitere Möglichkeit, um das Problem der abgehängten Objekte zu umgehen, ist die Benutzung eines erweiterten Persistenzkontexts, wie dies in Abschnitt 4.7.3 beschrieben ist.

Hier ein Beispiel für die Verwendung von *FetchType*:

```
@OneToMany(fetch = FetchType.EAGER)
public Collection<Email> getEmails() {}
...
```

4.6.8 Sortierung

Bei Relationen, die eine Zu-n-Beziehung abbilden, kann die Reihenfolge der Elemente in der zurückgegebenen Collection nicht einfach vorhergesagt werden. Um eine Sortierung durch die Persistenz-Engine vorzunehmen, wird an die Collection zusätzlich noch eine `@OrderBy`-Markierung angebracht. Die Sortierung wird nach Möglichkeit direkt an die Datenbank delegiert und ist so wesentlich effizienter als eine Sortierung der geladenen Elemente im Hauptspeicher. Die Annotation `@OrderBy` hat nur einen String-Parameter, der die entsprechende Sortierklausel angibt.

@OrderBy

Die Klausel besteht aus einer Komma-separierten Liste bestehend aus Feldern oder Propertys und der Angabe, ob auf- oder absteigend (`ASC` oder `DESC`) sortiert werden soll. Ist keine Reihenfolge angegeben, wird die aufsteigende Sortierung verwendet.

Listing 4.32
Sortierung nach Ort und Straße

```
@Entity
public class Person {
  ...
  @OneToMany
  @OrderBy("ort ASC, strasse DESC")
  public Collection<Adresse> getAdresse()
  { ... }
  ...
}
```

Wird `@OrderBy` ohne Parameter verwendet, erfolgt die Sortierung aufsteigend über den Primärschlüssel.

4.6.9 Fremdschlüsselspalten (@JoinColumn)

@JoinColumn

Bei Relationen, die über Fremdschlüssel abgebildet werden, kann das Verhalten der Fremdschlüsselspalte über die Annotation `@JoinColumn` festgelegt werden. Die meisten Attribute der Annotation sind von `@Column` bereits bekannt. Auch hier sind die Attribute alle wieder optional.

- name: Der Name der Spalte. Ist der Parameter nicht vorhanden, wird der Name aus dem Feldnamen der führenden Klasse gefolgt von einem Unterstrich und der Primärschlüsselspalte der anderen Klasse verwendet. Ist keine referenzierte Property vorhanden, wird er aus dem Namen der Entity, gefolgt von Unterstrich und Primärschlüsselspalte gebildet.
- referencedColumnName: Der Name der Spalte, auf den der Fremdschlüssel verweist. Üblicherweise ist dies der Primärschlüssel der führenden Seite.
- unique: Dieser boolesche Wert gibt an, ob die Werte in dieser Spalte eindeutig sein müssen. In der Voreinstellung ist dies nicht der Fall. Dieses Attribut ist eine Vereinfachung gegenüber der Angabe von `@UniqueConstraint` auf Tabellenebene.
- nullable: Dieser boolesche Wert, gibt an, ob die Datenbankspalte `null`-Werte enthalten darf oder nicht. In der Voreinstellung sind `null`- Werte erlaubt.
- insertable: Mit diesem Parameter wird angegeben, ob Werte in diese Spalte über SQL-insert-Statements geschrieben werden dürfen. Die Voreinstellung ist `true`.
- updatable: Über diesen Parameter wird das Verhalten bei SQL-Update gesteuert. Ist er auf `true` (Voreinstellung) gesetzt, wird die Spalte aktualisiert.
- columnDefinition: Dieser String-Wert kann ein DDL-Fragment enthalten, das beim Anlegen der Tabelle für diese Spalte genutzt werden kann. Dieses Anlegen kann über externe Tools oder auch durch den Persistenz-Provider erfolgen. Da dies ein SQL-DDL-Fragment ist, kann dies unter Umständen nicht zwischen verschiedenen Datenbanken portabel sein.
- table: Der Name der Tabelle, in welche die Spalte geschrieben wird. Üblicherweise ist dies die Tabelle, die via `@Table` definiert wurde und muss nicht extra angegeben werden. Soll die Spalte in einer anderen Tabelle landen, die über `@SecondaryTable` definiert wurde, kann diese Tabelle hier angegeben werden.

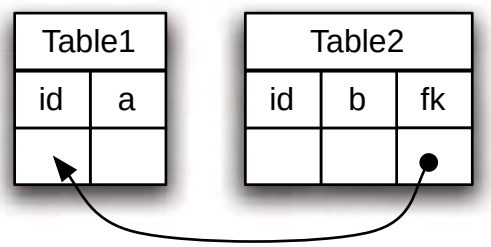


Abbildung 4.8
Beispieltabellen für JoinColumn

Der Code für das Beispiel aus Abbildung 4.8 könnte also wie folgt aussehen:

Listing 4.33
Code für das Beispiel aus Abbildung 4.8

```
@Entity
public class Table1
{
  @Id int id;

  @ManyToOne
  @JoinColumn(name="fk")
  Table2 table2;
}
```

Da auch die Annotation `@JoinColumn` nicht mehrfach an einem Element vorkommen darf, können zusammengesetzte Fremdschlüssel über `@JoinColumns` definiert werden. Das einzige Attribut dieser Annotation ist dabei eine Liste von `@JoinColumn`-Annotationen, welche die einzelnen Teile des Fremdschlüssels beschreiben.

@JoinColumns

Listing 4.34
Beispiel für @JoinColumns

```
...
  @ManyToOne
  @JoinColumns({
     @JoinColumn(name="PERS_NAME",
          referencedColumn="name"),
     @JoinColumn(name="PERS_VORNAME",
          referencedColumn="vorname")})
  private Person person;
...
```

4.6.10 Mappingtabellen (@JoinTable)

Wie bereits beschrieben, werden bei einigen Typen von Relationen die Beziehungen über Mappingtabellen abgebildet:

- Unidirektionale 1:n-Beziehungen
- Alle n:m-Beziehungen

@JoinTable

Mit der Annotation `@JoinTable` kann das Verhalten dieser Tabelle beeinflusst werden. Die Annotation muss dabei immer an der führenden Seite der Relation angegeben werden. Die Parameter von `@JoinTable` sind denen von `@Table` sehr ähnlich:

- name: Dieser Parameter vom Typ `String` gibt den Tabellennamen an. Ist dieser nicht angegeben, wird der Name aus den Namen der beteiligten Entity Beans zusammengesetzt, wobei zuerst der Name der führenden Tabelle kommt und die beiden Namen durch einen Unterstrich verkettet werden.
- catalog: Dieser `String`-Parameter gibt einen zu verwendenden Datenbankkatalog an. Ist das Attribut nicht gesetzt, wird der Standardkatalog verwendet.
- schema: Über diesen Parameter kann ein zu verwendendes Schema angegeben werden. Die Voreinstellung ist hier das Standardschema des Benutzers.
- joinColumns: Eine Liste von `@JoinColumn`-Annotationen, welche die Fremdschlüsselbeziehung der Join-Tabelle zur Tabelle des führenden Endes der Beziehung beschreibt.
- inverseJoinColumns: Analog zu den *joinColumns* aus dem Punkt vorher die Liste von Fremdschlüsselbeziehungen der Join-Tabelle zur Tabelle des nichtführenden Endes der Beziehung.
- uniqueConstraints: Eine Reihe von zusätzlichen Eindeutigkeitsbeschränkungen für bestimmte Spalten (siehe auch Abschnitt 4.1.1).

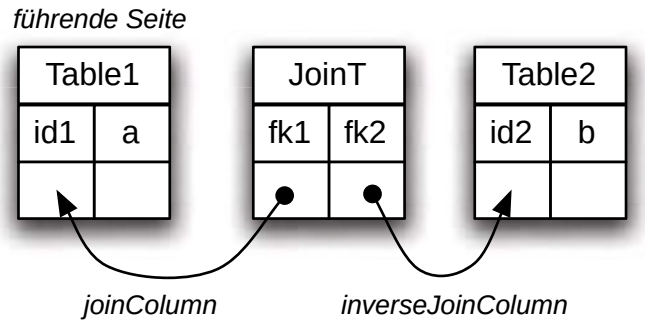


Abbildung 4.9
Beispieltabellen für JoinTable

Die Anordnung der Tabellen in Abbildung 4.9 könnte man also durch den folgenden Code erreichen:

Listing 4.35 *Implementierung für Abbildung 4.9*

```
/**
 * Entity mit führender Seite der Relation
 */
@Entity
public class Table1
{
  @Id int id1;
  int a;

  @ManyToMany
  @JoinTable(name="JoinT",
    joinColumns=@JoinColumn(
      name="fk1",
      referencedColumnName="id"1),
    inverseJoinColumns=@JoinColumn(
      name="fk2",
      referencedColumnName="id2")
   )
  Collection<Table2> table2;
}

@Entity
public class Table2
{
  @Id int id2;
  int b;

  @ManyToMany
  Collection<Table1> table1;
}
```

Ein einfacheres Beispiel, bei dem nur der Name der Join-Tabelle festgelegt wurde, ist in Weblog.getNutzer() zu sehen. Die beteiligten Tabellen sind in Abbildung 2.3 auf Seite 23 zu sehen.

4.6.11 Abbildung von java.util.Map

Im Gegensatz zu Einträgen in »einfachen« Collections besteht ein Eintrag in einer `java.util.Map` immer aus einem Schlüssel-Wert-Paar. Als Standard wird für den Schlüssel einer Map immer der Primärschlüssel des Entity Beans genommen. Soll ein anderer Schlüssel verwendet werden, kann dies via `@MapKey` angegeben werden. Das Attribut *name* gibt dabei den Namen der zu verwendenden Spalte an. Zu beachten

@MapKey

ist, dass sich der Schlüsselwert eines Eintrags in der Map nicht ändern darf. Deshalb ist es wichtig, dass die Methoden `equals()` und `hashCode()` der Entities (korrekt) überschrieben werden. Abschnitt 4.2.1 gibt einige Hinweise hierzu.

Listing 4.36
Abbildung einer Map

```
@Entity
public class Person {
...
  @OneToMany
  @MapKey(name="ort")
  Map<String, Adresse>adressen;
...
}
```

Wenn über die Annotation ein anderes Feld als der Primärschlüssel als Schlüsselwert der Map gewählt wird, muss dieser Wert eindeutig sein und sollte möglichst über einen Datenbank-Constraint als eindeutig gekennzeichnet werden. Dies kann bei der Definition der Spalte über `@Column` geschehen.

4.7 Der Entity Manager

Zugriffe auf die Entity Beans erfolgen über den *Entity Manager*, welcher einen *Persistence Context* zur Verfügung stellt. Dieser Entity Manager residiert üblicherweise in einem Session Bean bzw. dem Applikationscode, der die Zugriffe auf das oder die Entity Beans steuert.

Der Entity Manager ist nicht thread-safe und sollte deswegen nie in verschiedenen Threads gleichzeitig genutzt werden. Hier muss man sich dann pro Thread einen eigenen Entity Manager vom System holen bzw. geben lassen.

Operationen des Entity Managers können eine oder mehrere Exceptions werfen. Diese sind bei der nachfolgenden Beschreibung der Operationen hinter dem Namen der Operation als Kürzel aufgelistet und werden dann im Text nicht mehr gesondert erwähnt. Die Bedeutung der Kürzel erfolgt dann in Abschnitt 4.7.5 auf Seite 119.

4.7.1 Entity Manager holen

Der Entity Manager ist vom Typ `EntityManager` und wird üblicherweise durch Dependency Injection vom EJB-Container über die Annotation `@PersistenceContext` bereitgestellt. Im Fall von Java-SE-Anwendungen, bei denen kein EJB-Container zur Verfügung steht (Unit-Tests, Kommandozeilen-Clients oder auch Servlet-Container),

kann der Entity Manager auch programmatisch ermittelt werden. Dies wird weiter unten gezeigt.

Listing 4.37
Injizierung des Entity Managers

```
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

@Stateless
public class FacadeSessionBean implements Facade
{
   @PersistenceContext
   private EntityManager em;
   // ...
}
```

Der Entity Manager wird über `@PersistenceContext` injiziert. Die Optionen dieser Annotation werden in Abschnitt 4.7.3 beschrieben. Der Container ist in der Lage, einen existierenden Entity Manager auch an aufgerufene Komponenten weiterzugeben (beispielsweise von einem Session Bean an ein weiteres aufgerufenes Session Bean). Dabei werden vorhandene Transaktionen honoriert (zum Transaktionsverhalten im gemanagten Fall siehe Abschnitt 3.7).

In Java-SE-Anwendungen

In Umgebungen, bei denen der Entity Manager nicht durch den Container injiziert wird, kann dieser über eine *Entity Manager Factory* angefordert werden. In diesem Fall spricht man von einem *application-managed Entity Manager*.

Die Klasse *EntityManagerFactory* kennt die folgenden Methoden:

- `createEntityManager()`: Diese Methode gibt einen Standard-EntityManager zurück.
- `createEntityManager(Map props)`: Auch diese Methode liefert einen Entity Manager. Als Parameter kann eine Map von Eigenschaften mitgegeben werden, um das Verhalten des Entity Managers zu beeinflussen. Die Werte in dieser Map überschreiben dabei die Äquivalente in der Datei *persistence.xml*. Diese wird mit vordefinierten Schlüsselwerten in Abschnitt 4.9 beschrieben. Herstellerspezifische Werte, die nicht erkannt werden, müssen ignoriert werden.
- `isOpen()`: Gibt an, ob die Factory geöffnet hat – also ob sie in der Lage ist, neue Entity-Manager-Instanzen zu erzeugen.

- `close()`: Schließt die Factory und alle von ihr erlangten Entity-Manager. Die Factory kann danach auch keine neuen Entity-Manager-Instanzen mehr erzeugen.

Eine neue Entity Manager Factory kann also durch Aufruf der Methode `Persistence.createEntityManagerFactory()` erzeugt werden. Das Vorgehen insgesamt ist damit wie in Listing 4.38 gezeigt.

Listing 4.38 *Erzeugen eines Entity Managers durch die Applikation*

```
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

...
EntityManagerFactory fac =
      Persistence.createEntityManagerFactory("PU");
EntityManager em = fac.createEntityManager();
...
```

Im Normalfall ist das Erzeugen einer neuen Entity Manager Factory teuer, während es das Kreieren eines *Entity Managers* nicht ist. Entity-Manager-Factory-Objekte sollten also solange es geht verwendet werden.

Im Fall des *application-managed Entity Managers* werden meist die Transaktionen auch nicht via JTA gemanagt, so dass hier die Transaktions-API des Entity Managers eingesetzt wird.

4.7.2 JPA-Transaktions-API

Wird der Persistenzkontext durch die Applikation verwaltet, und nimmt er nicht an einer JTA-Transaktion teil, kann über die Methode `EntityManager.getTransaction()` eine Transaktion vom Typ `EntityTransaction` geholt werden. Die Schnittstelle `EntityTransaction` bietet diese Methoden:

EntityTransaction

- `begin()`: Startet eine neue Transaktion. Wirft eine `IllegalStateException`, falls der Entity Manager bereits an einer Transaktion teilnimmt.
- `commit()`: Führt ein Commit aus und schreibt ungesicherte Einträge in die Datenbank. Wirft eine `IllegalStateException`, falls keine Transaktion aktiv ist, und eine `RollbackException`, falls der Commit fehlschlägt.
- `rollback()`: Führt ein Rollback aus. Wirft eine `IllegalStateException`, falls keine Transaktion aktiv ist, und eine RBE, falls der Commit fehlschlägt.

- `setRollbackOnly()`: Deutet an, dass die aktuelle Transaktion nur noch durch ein Rollback beendbar ist. Wirft eine `IllegalStateException`, falls keine Transaktion aktiv ist.
- `isRollbackOnly()`: Gibt zurück, ob die aktuelle Transaktion nur noch durch einen Rollback beendet werden kann. Wirft eine `IllegalStateException`, falls keine Transaktion aktiv ist.
- `isActive()`: Gibt zurück, ob eine Transaktion aktiv ist.

Damit könnte der Code für das Speichern eines Objektes so aussehen:

Listing 4.39
Nutzen von Transaktionen durch die Applikation

```
...
EntityManager em = fac.createEntityManager();

EntityTransaction tx = em.getTransaction();
tx.beginn()
em.mach_was();
tx.commit();
...
```

4.7.3 Persistence Context

Der *Persistence Context* stellt eine Einheit von Aufgaben dar, die beispielsweise innerhalb einer Geschäftsmethode oder einer Anfrage eines Webclients abgearbeitet wird[3].

Innerhalb eines Persistenzkontexts gilt die Identitätsgarantie. Diese Garantie sagt aus, dass zwei Instanzen derselben Klasse mit demselben Primärschlüssel gleich sind:

Listing 4.40
Identitätsgarantie

```
...
@PersistenceContext
EntityManager em;

Entity e1 = em.find(Entity.class,new Long(123));
Entity e2 = em.find(Entity.class,new Long(123));

if (e1 == e2) // Bedingung ist wahr
```

In diesem Fall sind *e1* und *e2* identisch – die beiden Variablen zeigen auf dieselbe Stelle im Heap. Siehe auch 4.2.1 auf Seite 80.

[3] In Hibernate wird der *Persistence Context* als *Session* bezeichnet.

@PersistenceContext

Die Annotation `@PersistenceContext` kennt die folgenden Attribute:

- name: Der Name des Entity Managers. Dieser wird bei der Injizierung durch den Container nicht benötigt.
- unitName: Der Name einer Persistenzeinheit. Dieser bezieht sich auf das Attribut *name* des Elements <persistence-unit> in der Datei *persistence.xml* (Abschnitt 4.9). Dieser wird zum Beispiel dann benötigt, wenn die Entity Beans in einem anderen Archiv deployt werden als die Session Beans, in denen der Persistenzkontext genutzt werden soll.
  Ist in diesem Deployment-Deskriptor nur eine Persistence-Unit vorhanden, muss *unitName* nicht angegeben werden. Ansonsten kann damit die zu verwendende Konfiguration ausgewählt werden.

PersistenceContextType

- type: Dieses Attribut vom Typ `PersistenceContextType` gibt an, ob ein Persistenzkontext zum Einsatz kommen soll, der eine transaktionsweite (`TRANSACTION`) oder
- erweiterte (`EXTENDED`) Gültigkeit hat. Dies wird unten noch näher beschrieben.
- properties: In diesem Attribut können zusätzliche Eigenschaften des Contexts als Liste mit angegeben werden. Die einzelnen `@PersistenceProperty` bilden mit den String-Parametern *name* und *value* ein Schlüssel-Wert-Paar, das die gewünschte Eigenschaft definiert. Ein Persistenz-Provider muss Propertys, die er nicht kennt, ignorieren.

@PersistenceProperty

@PersistenceContexts

Falls man mehr als einen `@PersistenceContext` an einer Klasse benötigt, können diese über `@PersistenceContexts` als Liste zusammengefasst werden.

Erweiterter Persistenzkontext

In manchen Anwendungsfällen kann es sinnvoll sein, einen Persistenzkontext länger als für eine einzelne Anfrage vom Benutzer offen zu halten. Als Beispiel soll ein Wizard dienen, der aus mehreren Schritten besteht. Würde man den Context nach jedem Schritt schließen, würde man Änderungen in die Datenbank schreiben, die eventuell im nächsten Schritt weiter aktualisiert würden oder die man bei einem Abbruch durch den Benutzer wieder explizit löschen müsste.

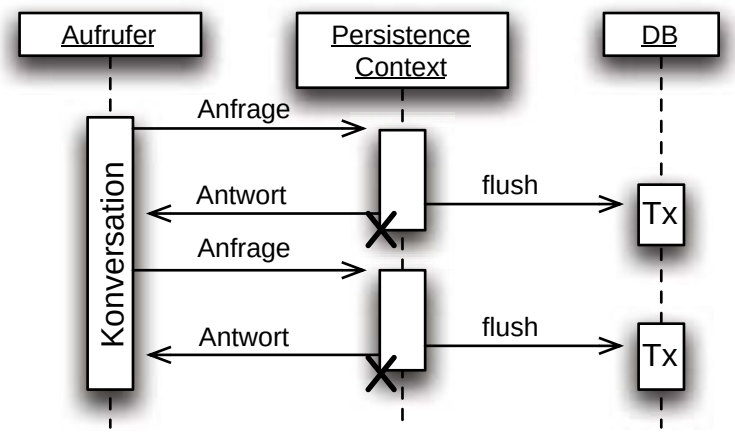


Abbildung 4.10
Transaktionaler Persistenzkontext

An dieser Stelle kann der Persistenzkontext als eine Art Puffer dienen, indem er auf die Lebenszeit der *Konversation* erweitert wird.

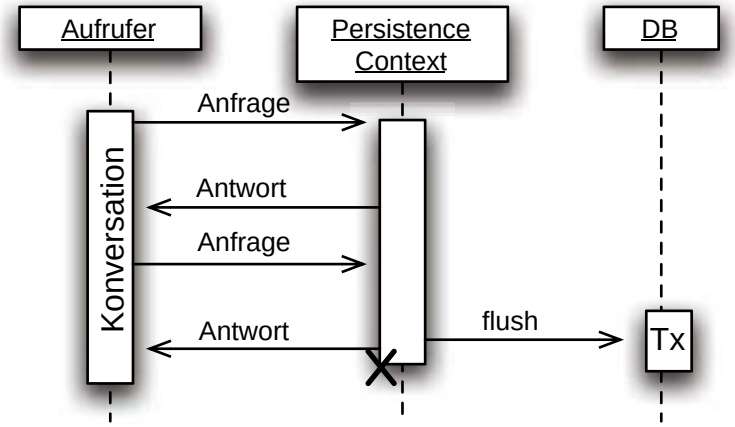


Abbildung 4.11
Erweiterter Persistenzkontext

Üblicherweise wird der Entity Manager dabei als ein Feld eines Stateful Session Beans angelegt und lebt von der ersten Instanziierung durch den Client bis zum Aufruf der mit `@Remove` gekennzeichneten Methode.

4.7.4 Entity-Bean-Lebenszyklus

Ein Entity Bean kann innerhalb seines Lebenszyklus vier Zustände annehmen:

new In diesem Zustand ist das Bean direkt nach der Instanziierung. Der Entity Manager weiß noch nichts von dem Bean und kann auch Änderungen nicht nachverfolgen.

managed Das Entity Bean ist initial mit `persist()` (s.u.) dem Entity Manager bekannt gemacht worden. Der Entity Manager ist nun in der Lage, Änderungen an dem Bean nachzuverfolgen und entsprechend in der Datenbank zu persistieren. In diesem Zustand hat das Bean eine Identität (z.B. einen Primärschlüssel).

detached In diesem Zustand ist das Bean bereits persistiert, aber vom Entity Manager abgekoppelt. Änderungen können nicht direkt persistiert werden. Erst über ein erneutes Anhängen an den Entity Manager können Änderungen wieder persistiert werden.

removed Das Bean wurde für das Löschen markiert. In diesem Zustand kann es wieder in den *managed* Zustand übernommen werden, oder es wird am Ende der Transaktion gelöscht (bzw. bei einem expliziten `flush()`.

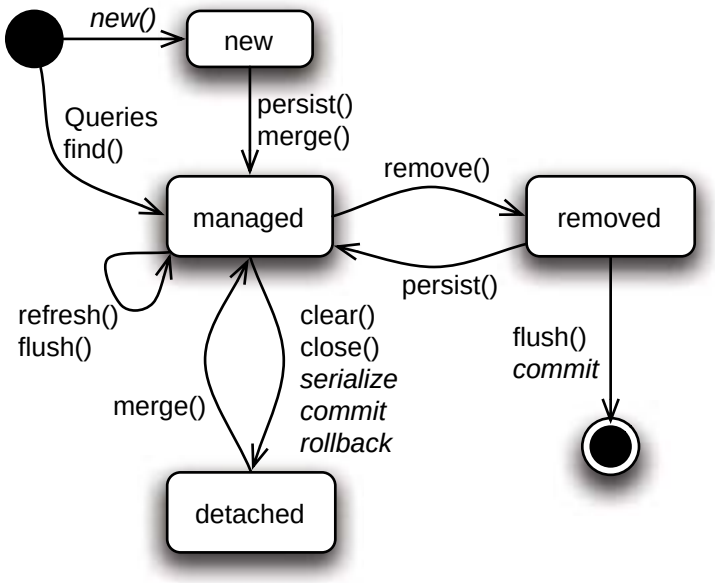


Abbildung 4.12 *Der Lebenszyklus von Entity Beans*

Abbildung 4.12 zeigt die Zustände und die Übergänge zwischen den Zuständen. Dabei sind an den Übergängen die jeweiligen Operationen

des Entity Managers angebracht, welche im übernächsten Abschnitt genauer beschrieben werden.

4.7.5 Exceptions des Entity Managers

Operationen des Entity Managers können eine oder mehrere der folgenden Exceptions werfen. Diese sind bei der Beschreibung der Operationen hinter dem Namen der Operation aufgelistet und werden dann im Text nicht mehr gesondert erwähnt.

EEE `EntityExistsException`: Die gegebene Entität existiert bereits in der Datenbank (also eine Zeile mit demselben Primärschlüssel).

ENFE `EntityNotFoundException`: Die gesuchte Entität existiert in der Datenbank nicht (mehr).

IAE `IllegalArgumentException`: Diese Exception wird üblicherweise geworfen, wenn das übergebene Objekt kein Entity Bean ist. Einige Operationen werfen diese Exception auch in anderen Fällen. Dies ist dann an der Methode beschrieben.

NRE `NoResultException`: Wird über `Query.getSingleResult()` (siehe 4.8) ein Ergebnis erwartet, es wird aber kein Ergebnis gefunden, wird diese Exception geworfen. Die Transaktion wird nicht zurückgerollt.

NURE `NoUniqueResultException`: Analog zur NRE wird diese Exception geworfen, wenn die Anfrage mehr als einen Treffer liefern würde.

OLE `OptimisticLockException`: Es wurde ein Konflikt beim Optimistic-Locking entdeckt. Mehr hierzu gibt es in Abschnitt 4.7.8.

PE `PersistenceException`: Diese Exception ist die Superklasse für fast alle hier gelisteten Exceptions (außer IAE). Sie wird geworfen, wenn ein Problem bei einer Operation aufgetreten ist, für das es keine spezifischere Exception gibt. Alle Unterklassen von PE bis auf NRE und NURE setzen den Transaktionsstatus auf *RollbackOnly*.

RE `RollbackException`: Diese Exception wird geworfen, wenn ein `EntityTransaction.rollback` fehlschlägt. In diesem Zustand hilft meist nur noch das Loggen des Fehlers.

TRE `TransactionRequiredException`: Ist der verwendete Entity Manager *container-managed*, fordert ein transaktionales Verhalten (siehe Abschnitt 4.7.3) und die Operation wird nicht innerhalb einer Transaktion aufgerufen, wird diese Exception geworfen.

4.7.6 Operationen

Neben den ab Seite 129 beschriebenen JP-QL-Abfragen bietet der Entity Manager einige weitere Operationen an, die im Folgenden beschrieben werden.

Die Methoden `close()`, `isOpen()`, `getTransaction()` und `joinTransaction()` sind dabei nur verwendbar, wenn der Entity Manager durch die Applikation gemanagt wird (also im Gegensatz zum Management durch z.B. den Java-EE-Container).

persist, IAE, TRE, EEE

Diese Methode persistiert das angegebene Objekt initial in der Datenbank und macht es gleichzeitig zum gemanagten Objekt. Die Signatur lautet:

```
void persist(Object entity)
```

Falls das übergebene Objekt bereits in der Datenbank existiert, wird eine `EntityExistsException` geworfen. Dies ist auch der Fall, wenn das Objekt *detached* ist und `persist()` aufgerufen wird.

Ist die übergebene Entity im Zustand *removed*, wird sie durch den Aufruf von `persist()` wieder *managed*. Dies funktioniert allerdings nur, wenn kein *flush* die Entity vorher entsorgt hat.

Hat das zu persistierende Objekt eine Relation zu einem anderen Objekt und ist diese Relation mit CascadeType.ALL oder CascadeType.PERSIST markiert, werden die angehängten Objekte bei `persist()` ebenfalls mitgespeichert. Die Optionen für das Kaskadieren sind in Abschnitt 4.6.6 beschrieben.

merge, IAE, TRE

Diese Methode kann im Lebenszyklus eines Entity Beans genutzt werden, um es in den Zustand *managed* zu bringen. Ist das Entity Bean *new*, wird es initial persistiert. Ist es im Zustand *detached*, wird ein Entity Bean in den vorhandenen Persitenzkontext *reattached*. Die Signatur der Methode ist:

```
<Type> Type merge(Type entityBean)
```

Das Verhalten der Methode ist auf den ersten Blick etwas ungewöhnlich:

- Ist das Entity *new*, wird eine Kopie angelegt, welche dann in den Zustand *managed* übernommen wird.

- Ist das Entity *detached*, wird der Zustand entweder in eine eventuell existierende gemanagte Kopie kopiert (deren vorheriger Zustand geht dabei verloren). Oder es wird eine neue gemanagte Kopie erzeugt und der Zustand in diese Kopie übertragen.
- Ist das Entity bereits ge*managed*, wird die Operation lediglich auf angehängte Entitys kaskadiert.

Der Rückgabewert der Methode ist die Kopie der Entity.

Eine IAE wird auch geworfen, wenn das übergebene Entity Bean in der Datenbank bereits gelöscht wurde.

Der Persistenz-Provider darf dabei keine Felder mergen, die als LAZY gekennzeichnet und nicht vorher aus der Datenbank geladen wurden.

Ist eine Versionsspalte (siehe Abschnitt 4.1.3) vorhanden, muss beim merge eine Versionsprüfung für das optimistische Locking stattfinden.

Ähnlich dem Verhalten bei `persist()` werden auch hier die Optionen für die Kaskadierung berücksichtigt. Ist die Option CascadeType.ALL oder CascadeType.MERGE gegeben, werden angehängte Objekte ebenfalls mit gemergt.

remove, IAE, TRE

Über die Methode kann eine Entity Bean aus der Datenbank gelöscht werden. Dies ist allerdings nur möglich, wenn sie einerseits in der Datenbank vorhanden ist und andererseits *managed* ist. Trifft Letzteres nicht zu, wird eine IAE geworfen.

```
void remove(Object entity)
```

Wurde die Entity Bean durch `remove()` in den Zustand *removed* gebracht, kann sie, bevor die Änderungen aus dem Persistenzkontext in die Datenbank geschrieben wurden, noch via `persist()` wieder in den Zustand *managed* gebracht werden. Allerdings darf man sich nicht darauf verlassen, dass die finale Löschung erst zum Zeitpunkt des Commit der Transaktion geschieht.

Auch bei dieser Operation werden die Kaskadierungsoptionen berücksichtigt. Ein Beispiel hierfür findet sich in Listing 4.31.

find, IAE

Diese Operation bietet eine Suche über den Primärschlüssel eines bekannten Objekts an. Das Methodensignatur hierzu lautet:

```
<Type> Type find(Class<Type>, Object pk)
```

Die Methode wirft eine IAE, wenn der erste Parameter keine als Entity Bean markierte Klasse ist oder wenn der zweite Parameter keinen Typ hat, der für den Primärschlüssel des gesuchten Entity Beans gültig ist. Die Ausführung dieser Methode geht immer auf die Datenbank, außer das gesuchte Datum ist bereits im selben Persistenzkontext vorhanden. Sie liefert ein voll initialisiertes Objekt zurück bzw. `null`, falls das gesuchte Objekt nicht in der Datenbank gefunden werden konnte. Die Suche an sich verläuft dabei polymorph. Das heißt, es wird nicht nur in der direkt angegebenen Klasse nach dem gewünschten Objekt gesucht, sondern auch in Subklassen.

getReference, ENFE, IAE

Dieser Aufruf ähnelt `find` von oben, indem eine durch einen Primärschlüssel gegebene Instanz einer Klasse in der Datenbank gesucht wird.

```
<Type> Type getReference(Class<Type>, Object pk)
```

Das zurückgegebene Objekt ist üblicherweise ein *Proxy* für das real gesuchte Objekt. Daduch kann `getReference()` den Zustand *lazy* (siehe Abschnitt 4.6.7) laden. Dieser wird dann beim Zugriff auf die Felder des Beans nachgeladen. Ist das gesuchte Objekt beim Zugriff auf die Felder nicht in der Datenbank, wird eine ENFE geworfen. Der Entity Manager darf diese Exception allerdings auch bereits beim Aufruf von `getReference()` werfen.

Wurde ein Proxy erhalten und der Persistenzkontext nimmt den Zustand *detached* an, kann danach nicht mehr auf die Felder zugegriffen werden, wenn sie nicht bereits durch einen Zugriff geladen wurden, als der Kontext noch *managed* war.

flush, PR, TRE

Schreibt den aktuellen Zustand des Persistenzkontexts in die unterliegende Datenbank. Dies geschieht im Normalbetrieb bei einem *commit* am Ende der Transaktion oder vor Abfragen automatisch. Über diese Methode kann die Synchronisation auch explizit angestoßen werden. Die Signatur der Methode lautet:

```
void flush()
```

Misslingt die Operation, wird eine PE geworfen. Ist ein Entity Bean im Zustand *removed*, wird es beim `flush()` aus der Datenbank gelöscht.

setFlushMode

Setzt den Modus, nach dem die Synchronisation mit der Datenbank erfolgen soll. Die Signatur der Methode ist

```
void setFlushMode(FlushModeType mode).
```

Der Typ `FlushModeType` kennt dabei diese beiden Werte: *FlushModeType*

- AUTO: Ist dieser Modus gewählt, stellt der Persistenz-Provider sicher, dass Änderungen an den Beans vor einer Abfrage für diese Abfrage sichtbar sind. Dies kann bedeuten, dass die Daten vor der Abfrage in die Datenbank geschrieben werden. Dieser Modus ist der Standardfall.
- COMMIT: Ist dieser Modus aktiv, synchronisiert der Persistenz-Provider Änderungen an den gemanagten Objekten beim Commit in die Datenbank. Der Provider darf darüber hinausgehend auch zu anderen Zeiten die Daten synchronisieren (z.B. wenn der Cache voll ist).

Der COMMIT-Modus ist dann interessant, wenn man ein Anwendungsmuster hat, bei dem man innerhalb einer Geschäftsmethode mehrmals nacheinander Daten sucht, ändert und dann wieder sucht. In diesem Fall würden bei AUTO alle Änderungen in vielen einzelnen Operationen direkt geschrieben und somit eventuell zu viele Updates auf der Datenbank ausgelöst. Speziell für den Fall eines einzelnen aktualisierten Objekts bedeutet die Verwendung von COMMIT, dass nur ein einzelnes Update zur Datenbank gesendet wird.

Der *Flush Mode* kann auf Entity-Manager-Ebene für einzelne Abfragen gezielt überschrieben werden – siehe Abschnitt 4.8.2.

Vor dem Endstand der Spezifikation gab es noch einen Flush Mode *NEVER*, mit dem festgelegt werden konnte, dass Daten nur explizit bei einem `flush()` an die Datenbank geschickt werden. Es ist zu erwarten, dass die meisten Hersteller von JPA-Produkten diesen zusätzlichen Modus mit anbieten. Hier muss dann eine proprietäre Annotation bzw. Version von *FlushMode* verwendet werden.

getFlushMode

Liefert den aktuellen *Flush Mode* für den Persistenzkontext zurück. Die Werte des Kontexts und deren Verhalten sind unter `setFlushMode` beschrieben:

```
FlushModeType getFlushMode()
```

clear

Diese Operation *detachet* alle Entity Beans, die am aktuellen Persistenzkontext hängen. Änderungen an den Objekten, die nicht vorher explizit über `flush()` in die Datenbank geschrieben wurden, werden durch diese Operation nicht persistiert. Die Signatur der Operation lautet:

```
void clear()
```

Da sich der Persistenzkontext für die Identitätsgarantie die angehängten Entitys merkt, kann dieser bei Massen-Inserts sehr schnell sehr viel Speicher verbrauchen. Über ein explizites `clear()` kann dieser Speicher wieder freigegeben werden:

Listing 4.41
Batch Inserts

```
...
@PersistenceContext
EntityManager em;
int BSIZE = 10; // JDBC Batch Size
...
em.setFlushMode(FlushModeType.COMMIT);
for (int i =0 ; i < GROSSE_ZAHL ; i++)
{
  DBObjekt dbo = ...
  em.persist(dbo);
  if (i % BSIZE == 0)
  {
    em.flush();
    em.clear();
  }
}
```

In Listing 4.41 werden viele Inserts durchgeführt. Alle 10 Inserts werden die angelegten Objekte an die Datenbank übertragen und danach aus dem Persistenzkontext entfernt, um Speicher zu sparen. Die Anzahl der geschriebenen Objekte entspricht dabei optimalerweise der JDBC Batch Size (die Möglichkeit, diese einzustellen, ist stark von der verwendeten Datenbank und dem verfügbaren Treiber abhängig).

refresh, IAE, TRE

Über diese Methode wird der Zustand des übergebenen Objekts erneut aus der Datenbank gelesen. Dies überschreibt lokale Änderungen am Objekt. Die Signatur ist

```
void refresh(Object entityBean)
```

Auf den ersten Blick erscheint diese Methode paradox, da man ja im Normalfall die Anzahl der Zugriffe auf die Datenbank verringern

möchte. Hat man allerdings externe Quellen wie Trigger, andere Anwendungen oder Threads, die ebenfalls schreibend auf eine Datenbankzeile zugreifen, kann eine Zeile über `refresh()` erneut aus der Datenbank gelesen werden.

lock, IAE, PE, TRE

Diese Methode setzt den *LockMode* für das angegebene Entity. Die Signatur ist

```
void lock(Object entity, LockModeType mode)
```

LockModeType

Der `LockModeType` kennt dabei die beiden Werte `READ` und `WRITE`, die im Abschnitt über das optimistische Locking (Abschnitt 4.7.8) näher beschrieben werden.

getDelegate

Diese Methode liefert den unterliegenden Persistenz-Provider (also die unterliegende Implementierung, wie beispielsweise die Hibernate Session) für den Entity Manager. Die Signatur ist

```
Object getDelegate()
```

contains, IAE

Mit dieser Methode kann geprüft werden, ob das angegebene Entity Bean zum aktuellen PersistenceContext gehört. Die Signatur der Methode ist

```
boolean contains(Object entity)
```

Sie gibt dann `true` zurück, wenn das Entity Bean im aktuellen Kontext und im Zustand *managed* ist.

close

Diese Methode schließt den Entity Manager und gibt belegte Ressourcen des Entity Managers wieder frei. Zugriffe auf die Methoden des Entity Managers (außer `isOpen()` und `getTransaction()`) oder der Klasse `Query` werfen eine `IllegalStateException`.

Ist zum Zeitpunkt des Aufrufs von `close()` noch eine Transaktion aktiv, bleibt der Persistenzkontext noch bis zum Ende der Transaktion erhalten. Daten werden damit auch erst in die Datenbank geschrieben, wenn die Transaktion endet.

Die Signatur der Methode ist ganz einfach

```
void close()
```

Wird `close()` im Container-Managed-Umfeld aufgerufen, wirft sie ebenfalls eine `IllegalStateException`.

isOpen

Diese Methode mit der Signatur

```
boolean isOpen()
```

gibt zurück, ob der Entity Manager »offen« ist oder via `close()` bereits geschlossen wurde.

getTransaction

Diese Methode dient dazu, im Fall eines *application-managed Entity Managers*, eine Transaktion bereitzustellen und zu beeinflussen. Dies wurde bereits in Abschnitt 4.7.2 beschrieben.

```
EntityTransaction getTransaction()
```

Diese Methode wirft eine `IllegalStateException`, wenn der Entity Manager an JTA-Transaktionen teilnimmt und somit das Transaktionshandling von einem (externen) Transaktionsmanager vorgenommen wird.

joinTransaction, TRE

Über diese Methode kann einem Entity Manger, der außerhalb der laufenden JTA-Transaktion erzeugt wurde, mitgeteilt werden, dass er dieser Transaktion beitreten soll. Damit kommt er unter die Kontrolle des JTA-Transaktionsmanagers, so dass dieser das Schreiben der Daten in die Datenbank kontrollieren kann. Dies ist speziell im Fall von verteilten Transaktionen interessant, bei denen die Operationen auf der Datenbank von anderen (externen) Systemen abhängen.

```
void joinTransaction()
```

4.7.7 Callbacks bei Entity Beans

Auch für Entity Beans gibt es Callback-Methoden, die zu bestimmten Zeitpunkten im Lebenszyklus aufgerufen werden. Der Lebenszyklus wurde in Abschnitt 4.7.4 auf Seite 118 beschrieben.

Generell gilt, dass eine `RuntimeException`, die in den Callbacks geworfen wird, die laufende Transaktion zurückrollt und damit die Operation auf dem Entity Manager abgebrochen wird, ohne den Zustand in der Datenbank zu modifizieren.

- `@PrePersist`: Die markierte Methode wird beim Aufruf der *persist*-Methode des Entity Managers vor dem Insert in die Datenbank aufgerufen.
  Dieser Callback wird auch für alle Entity Beans aufgerufen, die via Cascade mitpersistiert werden. *@PrePersist*
- `@PostPersist`: Die markierte Methode wird direkt nach dem SQL-Insert oder -Update aufgerufen. Der Callback kaskadiert auf mitpersistierte Entity Beans. *@PostPersist*
- `@PreRemove`: Die markierte Methode wird beim Aufruf der *remove*-Methode des Entity Managers vor dem Delete auf der Datenbank aufgerufen.
  Dieser Callback wird auch für alle Entity Beans aufgerufen, die via Cascade mitgelöscht werden. *@PreRemove*
- `@PostRemove`: Die markierte Methode wird direkt nach dem Löschen der Zeile aus der Datenbank aufgerufen.
  Dieser Callback wird auch für alle Entity Beans aufgerufen, die via Cascade mitgelöscht werden. *@PostRemove*
- `@PreUpdate`: Dieser Callback wird aufgerufen, bevor ein verändertes Entity Bean in die Datenbank zurückgeschrieben wird. *@PreUpdate*
- `@PostUpdate`: Dieser Callback wird aufgerufen, nachdem ein verändertes Entity Bean in die Datenbank zurückgeschrieben wurde. *@PostUpdate*
- `@PostLoad`: Die markierte Methode wird aufgerufen, nachdem ein Datensatz für das Entity Bean von der Datenbank geladen wurde, aber bevor die anfordernde Abfrage zurückkehrt. *@PostLoad*

Wie auch schon bei den anderen Enterprise JavaBeans gibt es auch bei den Entity Beans die Möglichkeit, die Callbacks in eine eigene Klasse auszulagern, um sie so wiederzuverwenden. Allerdings lautet die Annotation hier `@EntityListeners`. Der Parameter ist dabei ein Feld von Klassen, welche die Callbacks implementieren. *@EntityListeners*

Listing 4.42 Callbacks in einer externen Datei

```
@EntityListeners({EListener.class})
@Entity
public class MyEntityBean
{
  ...
}
```

Globale Callbacks (*Default Listeners*), die für alle Entitys in einer Anwendung gelten sollen, können im Deployment-Deskriptor definiert werden. Dies wird in [10] beschrieben. Diese globalen Callbacks können für eine Entity via `@ExcludeDefaultListeners` abgeschaltet werden.

@ExcludeDefaultListeners

Aufrufreihenfolge

Sind globale Callbacks definiert, werden diese zuerst aufgerufen. Sind mehrere Listener an einer Entity (über die Listener-Klassen) definiert, werden diese in der Reihenfolge der Definition ausgeführt. Hat eine Entity und eine Superklasse denselben Callback definiert, wird erst der der Superklasse ausgeführt. Dieses Verhalten kann über `@ExcludeSuperclassListeners` abgeschaltet werden. Diese Annotation ist nur an der Klasse möglich.

@ExcludeSuperclassListeners

4.7.8 Optimistisches Locking

In der JPA ist der Standardfall das sogenannte *Optimistic Locking*. Dabei wird eine Datenbankzeile nicht gesperrt, um später eventuell ein Update auszuführen, sondern es wird beim Update geschaut, ob der Stand der Datenbankzeile noch der ist, der er zum Zeitpunkt des Lesens der Zeile war. Wurde die Zeile durch eine andere Anwendung oder einen anderen Thread verändert, bekommt die Anwendung dies durch eine `OptimisticLockException` mitgeteilt. Sie kann dann die Verarbeitung abbrechen oder im Falle einer Konversation (siehe Abschnitt 4.7.3) zum Start dieser Konversation zurückkehren, um dem Benutzer die extern aktualisierten Daten zu präsentieren.

Dabei wird davon ausgegangen, dass auf Datenbankebene die Isolationsstufe *ReadCommitted* aktiv ist.

Wie bereits in Abschnitt 4.1.3 beschrieben, erfolgt die Überprüfung, ob eine Zeile noch aktuell ist, über eine Versionsspalte. Der Wert dieser Spalte wird dabei bei jeder Aktualisierung des Objekts, die nicht durch eine Bulk-Operation erzeugt wurde, hochgezählt. Das generierte SQL für ein Update durch den Container sieht dann in etwa so aus (die erwartete Version ist dabei die Version 2):

```
UPDATE WL_Nutzer SET
    NAME='Neuer Name', VERSION='3'
WHERE
       email='hwr@pilhuhn.de' AND
       VERSION='2'
```

Listing 4.43 *SQL-Update bei Optimistic Locking*

Wurde die Zeile in der Zwischenzeit verändert, schlägt der Update fehl, da die zu ändernde Zeile nicht mehr gefunden wird. Ansonsten wird der Versionszähler auf 3 gesetzt.

Zusätzlich zur Versionsspalte kann das Locking eines Eintrags weiter über die `lock(`*`Object`*`,`*`LockModeType`*`)`-Methode des Entity Managers beeinflusst werden. Diese Methode kann für das als ersten Parameter mitgegebene Objekt eine Sperre auf Datenbankebene erwirken. Welche Art der Sperre dies ist, wird über den zweiten Parameter vom Typ `LockModeType` angegeben. Es gibt zwei Modi:

LockModeType

- READ: Hiermit wird ein Read-Lock auf der Datenbank angefordert. Damit soll das gelockte Entity vor Dirty-Reads und Non-Repeatable-Reads geschützt werden.
- WRITE: Wie READ, aber der Entity Manager muss zusätzlich noch den Wert der Versionsspalte hochzählen, um so Aktualisierungsversuche auf die Zeile mit dem alten Versionswert zu unterbinden.

Der Entity Manager muss `EntityManager.lock()` nur unterstützen, wenn eine Versionsspalte für das Entity Bean angegeben wurde. Anderenfalls muss er eine PE werfen.

4.8 Abfragen

Abfragen werden über den Aufruf einer der `create*Query()`-Methoden des Entity Managers erzeugt. Diese liefern ein Objekt vom Typ `Query` zurück, über welches man die Abfrage dann mit Parametern versorgen und ausführen kann. Listing 4.44 liefert ein einfaches Beispiel hierfür.

```
...
@PersistenceContext
EntityManager manager;

Query q = manager.createQuery(
  "select a from Address a where name = :qname"
  );
q.setParameter("qname","Rupp"); // nach 'Rupp' suchen
List results = q.getResultList();
```

Listing 4.44 *Beispiel einer Datenbankabfrage mit JP-QL*

Die Liste *results* besteht dann aus Objekten vom Typ *Address*. Dies könnte man klarer darstellen, indem man die Liste als generische Liste erstellt:

```
List<Address> results = q.getResultList();
```

Abfragen werden immer polymorph ausgeführt. Dies bedeutet, dass als Ergebnis nicht nur Objekte des abgefragten Typs zurückgeliefert werden, sondern auch die seiner Subklassen.

Die Abfragesprache JP-QL ist dabei gegenüber dem EJB-2.x-Standard deutlich erweitert worden und wird in Kapitel 5 ab Seite 141 gesondert beschrieben.

4.8.1 Erstellen einer Abfrage

Die einzelnen Methoden zur Abfrage sind:

- `createQuery(String abfrage)`: Diese Methode erhält als Parameter den Abfragestring in JP-QL. Ein Beispiel hierfür wurde in Listing 4.44 bereits gezeigt.
- `createNamedQuery(String abfrageReferenz)`: Bei dieser Methode wird die Abfrage in einer externen Annotation hinterlegt; der Parameter der Methode verweist dann auf diese Abfrage. Diese Methode kann sowohl für Abfragen in JP-QL als auch in SQL verwendet werden. Die notwendigen Annotationen werden weiter unten beschrieben.

  Diese Version der Abfrage ist für portablere Applikationen gegenüber `createQuery()` zu bevorzugen, da die gesondert ausgewiesenen Abfragen im Deployment-Deskriptor überschrieben werden können. Persistenz-Provider sind auch bereits beim Deployment in der Lage, die Named Querys auf syntaktische Korrektheit zu überprüfen und Fehler zu melden.
- `createNativeQuery`: Bei dieser Abfrage wird ein mitgegebener SQL-String direkt an die Datenbank gesendet, ohne vor dem Absetzen noch vom EntityManager in den jeweiligen Datenbankdialekt übersetzt zu werden. Diese Methode gibt es in drei Varianten:
  - `createNativeQuery(String sql)`: Erzeugt eine SQL-Abfrage, bei der ein SQL-Statement für Massen-Updates oder -Löschungen verwendet werden soll. Hier werden entweder keine komplexen Objekte zurückerwartet, so dass keine Abbildung der Rückgabe auf Java-Objekte notwendig ist. Oder das Mapping auf die Rückgabe ist bereits in `@NamedNativeQuery` angegeben, wie dies in Abbildung 4.46 gezeigt ist.

- `createNativeQuery(String sql, Class resCl)`: Erstellt eine SQL-Abfrage. Das zurückzugebende Ergebnis ist vom Typ, der über *resCl* mitgegeben wird.
- `createNativeQuery(String sql, String sMap)`: Kreiert eine SQL-Abfrage. Die Typen des zurückzugebenden Ergebnisses werden über das in *sMap* referenzierte Mapping definiert. Diese Abbildungsvorschriften werden unten beschrieben.

Native-SQL-Abfragen sind sehr gut, um hoch optimierte Statements an die Datenbank zu senden, bergen aber das Risiko, dass die Anfragen nicht zwischen Datenbanken portabel sind.

Alle `create*Query()`-Methoden geben ein `Query`-Objekt zurück, über das die Abfrage weiter verfeinert werden kann. Dies wird dann in Abschnitt 4.8.2 beschrieben.

Named Query für JP-QL (@NamedQuery)

@NamedQuery

Eine *Named Query* wird über `@NamedQuery` definiert. Diese Annotation, die auf Klassenebene von Entitys oder Mapped Superclasses angebracht wird, kennt drei Parameter, von denen nur der Parameter *hints* optional ist.

- name: Der Name, über den die Abfrage referenziert wird. Dieser Name muss innerhalb einer Persistenzeinheit eindeutig sein.
- query: Die in JP-QL verfasste Abfrage.
- hints: Hinweise an den Persistenz-Provider, wie mit der Abfrage umgegangen werden soll. Dies ist eine Liste von `@QueryHint`-Annotationen, die *name*- und *value*-Attribute haben. Diese Hinweise sind produktspezifisch.

@QueryHint

Da pro Java-Element nur eine Annotation eines Typs möglich ist, können via `@NamedQueries` mehrere Querys zusammengefasst werden.

@NamedQueries

Named Querys sind innerhalb der gesamten Persistenzeinheit sichtbar. Um in größeren Projekten den Überblick nicht zu verlieren und die Definition der Query schnell wiederfinden zu können, empfiehlt es sich, den Namen der Query mit dem unqualifizierten Namen der Entity-Klasse als Präfix zu versehen. Dies kann beispielsweise so aussehen:

```
@NamedQuery(name="Poster.PosterListe",
  query="SELECT p FROM Poster p"),
//...
public class Poster
```

Named Query für SQL (@NamedNativeQuery)

@NamedNativeQuery

Eine *Named Native Query* dient zur Definition von SQL-Statements, welche direkt an die Datenbank gesendet werden. Hierfür kann die Annotation `@NamedNativeQuery` auf Klassenebene bei Entitys und Mapped Superclasses angebracht werden und hat zusätzlich zu den Parametern von `@NamedQuery` noch diese weiteren Parameter:

- resultClass: Dieser Parameter vom Typ `Class` gibt die von der Query zurückgegebene Klasse einer Entity an. Er wird nur benötigt, wenn die Native Query ausschließlich Entitys liefert.
- resultSetMapping: Dieser String-Parameter benennt ein Mapping des Resultsets auf die angegebene Rückgabeklasse. Die Umsetzung an sich wird dabei über `@SqlResultSetMapping` definiert, welches nachfolgend beschrieben ist.

@NamedNativeQueries

Mehrere `@NamedNativeQuery`-Annotationen an derselben Klasse müssen über `@NamedNativeQueries` zusammengefasst werden.

ResultSetMapping für Native Querys

@SqlResultSetMapping

Bei den *Native Queries* kann der Persistenz-Provider den Rückgabetyp der Anfrage nicht wissen, da in den erhaltenen ResultSets beliebige Werte enthalten sein können. Aus diesem Grund muss die Abbildung zwischen der Rückgabeklasse und dem von der Datenbank erhaltenen Resultset definiert werden. Dies geschieht über `@SqlResultSetMapping`. Diese Annotation hat drei Parameter:

- name: Der Name des Mappings, über den es referenziert wird. Dieser Parameter muss immer angegeben werden.
- entities: Ein Feld von `@EntityResult`-Einträgen, welche die Abbildung auf Entitys definieren. Diese müssen in der Reihenfolge angegeben werden, wie sie auch im ResultSet auftauchen.
- columns: Ein Feld von `@ColumnResult`-Einträgen, die alternativ zu *entities* die Umsetzung zu anderen Typen definieren.

@SqlResultSetMappings

Wie gewohnt, müssen mehrere `@SqlResultSetMappings` an einer Klasse über `@SqlResultSetMappings` zusammengefasst werden.

In einem Mapping dürfen sowohl *entities* als auch *columns* vorkommen. Die beiden Parameter schließen sich also gegenseitig nicht aus.

@EntityResult

Die Annotation `@EntityResult`, die in `@SqlResultSetMapping.entities` verwendet wird, kennt ihrerseits wieder drei Parameter:

- entityClass: Dieser Parameter vom Typ `Class` gibt den Datentyp der zurückgegebenen Klasse an.
- fields: Ein Feld von `@FieldResult`-Einträgen, welche die Spalten in der Select-Klausel auf die Felder oder Propertys der Klasse abbilden.
- discriminatorColumn: Gibt die Spalte in der Select-Klausel an, die als Diskriminator fungiert.

Listing 4.45 *Mapping einer Entity-Klasse, Artikel.java*

```
@NamedNativeQuery(name="EinExzerpt",
  resultSetMapping="toString",
  query="SELECT substring(text for 100) AS start " +
        "FROM WL_Artikel WHERE id = ?1")
@SqlResultSetMapping(name="toString",
  columns=@ColumnResult(name="start"))

@Entity
@Table(name="WL_Artikel")
public class Artikel implements Serializable
```

In Listing 4.45 werden über die SQL-Abfrage die ersten 100 Zeichen der Spalte text aus der Tabelle WL_Artikel gelesen. Diese Abfrage ist in PostgreSQL-spezifischem SQL gehalten. Im Mapping *toString* wird definiert, dass das Resultat nur die Pseudo-Spalte *start* zurückgeben soll.

Die Query wird dann aufgerufen, wie in Listing 4.46 gezeigt.

Listing 4.46 *Nutzung der SQL-Abfrage mit Mapping, FacadeSessionBean.holeArtikelExzerpt*

```
...
  Long idl = ...
  Query q = em.createNamedQuery("EinExzerpt");
  q.setParameter(1, idl);
  String ret = (String)q.getSingleResult();
...
```

@FieldResult

Mappen von Entity-Klassen `@FieldResult` bildet nun endlich die Felder auf die Spalten ab. Der Parameter *name* gibt dabei das Feld und der Parameter *column* die Spalte an. Die Spalten sind dabei die gleichen Spalten, die auch in der Select-Klausel verwendet werden.

@ColumnResult

Mappen von einzelnen Spalten Zum Umsetzen der Ergebnisspalte steht `@ColumnResult` zur Verfügung. Diese Annotation kennt nur einen Parameter *name*, der die Spalte angibt, welche zurückgegeben werden soll.

Ein Beispiel hierfür wurde bereits in Listing 4.45 weiter oben gezeigt.

4.8.2 Ausführung und Parametrierung der Abfragen

Nachdem eine *Query* über die beschriebenen *create*Query*-Methoden erzeugt wurde, kann sie mit den folgenden Methoden parametriert und ausgeführt werden.

Wenn nicht anders angegeben, liefern die Methoden ein `Query`-Objekt zurück, so dass man die Statements einfach aneinanderhängen kann.

- `getResultList()`: Diese Methode setzt die Abfrage ab und gibt eine `java.util.List` von Ergebnissen zurück. Diese Methode wirft eine `IllegalStateException`, wenn sie für Update- oder Delete-Statements verwendet wird.
- `getSingleResult()`: Diese Methode gibt ein einzelnes Ergebnis als `Object` zurück. Wurde kein Ergebnis in der Datenbank gefunden, wird eine `NoResultException` geworfen, bei mehr als einem Ergebnis eine `NonUniqueResultException`.
- `executeUpdate()`: Diese Methode dient zum Ausführen von Update- oder Delete-Statements. Diese werden im Abschnitt 4.8.3 beschrieben. Die Methode wirft eine `IllegalStateException`, wenn sie für Select-Statements verwendet wird. Ist keine Transaktion aktiv, wird eine TRE geworfen. Der Rückgabewert vom Typ `int` gibt die Anzahl der aktualisierten oder gelöschten Zeilen an.
- `setFirstResult(int n)`: Definiert die Nummer des ersten erwarteten Ergebnisses. Diese Methode kann zusammen mit `setMaxResults()` verwendet werden, um durch die Ergebnisse zu blättern.
- `setMaxResults(int n)`: Diese Methode gibt an, wie viele Ergebnisse in der zurückgegebenen Liste maximal enthalten sein sollen.
- `setFlushMode(FlushModeType mode)`: Diese Methode setzt den *Flush Mode* für diese einzelne Abfrage. Die Bedeutung wurde bereits in Abschnitt 4.7.6 auf Seite 123 beschrieben.
- `setHint(String hintName, Object value)`: Über diese Operation kann ein herstellerspezifischer Hinweis an den Persistenz-Provider zur Optimierung der Anfrage bereitgestellt werden. Ein Hinweis, den eine Implementierung nicht kennt, muss sie ignorieren.
- `setParameter()`: Über diese Methode werden die Parameter der Anfrage eingetragen. Prinzipiell sind zwei Arten von Parametern zu unterscheiden:

?n

  - Positionsparameter: Diese Parameter werden mit einem Fragezeichen und einer fortlaufenden Nummer in der Abfrage hinterlegt (*?n*), wobei ein Parameter in der Anfrage mehrfach vorkommen darf. Die Nummerierung der Parameter beginnt mit 1.

```
SELECT a FROM Adresse AS a
WHERE a.plz = ?1
```

Dieser Parameter wird dann via `setParameter(1,...)` gesetzt.

- Benannter Parameter: Der Parameter bekommt einen Namen und wird über diesen angesprochen. Dies sieht in JP-QL wie folgt aus: *:name*. Der Parameter wird also durch einen Doppelpunkt eingeleitet und erhält dann einen Namen, der dann ein gültiger *Identifier* sein muss. Diese Art der Parameter ist nur für JP-QL und nicht für die Native Queries verfügbar.

:name

```
SELECT a FROM Adresse AS a
WHERE a.plz = :thePlz
```

Dieser Parameter wird dann in einer Abfrage via

```
Query q = ... ;
q.setParameter("thePlz","12345");
```

gesetzt. Auch hier kann ein Parameter in einer Query mehrfach verwendet werden. Wichtig ist, dass bei diesen benannten Parametern zwischen Groß- und Kleinschreibung unterschieden wird.

Pro Art der Parametrierung gibt es dann drei Varianten, um die Parameter zu setzen. Das Fragezeichen steht in nachfolgender Aufzählung entweder für einen String, der den benannten Parameter darstellt, oder für eine Zahl, welche die Nummer des gewünschten Positionsparameters angibt.

- `setParameter(?, Object val)`: Dies ist die generelle Form, in der alle erlaubten Datentypen bis auf die Zeit-Typen gesetzt werden können.
- `setParameter(?, Date dat, TemporalType tt)`: Hiermit werden ein `java.util.Date` an die Abfrage gebunden. Der Parameter *tt* gibt dabei an, wie die Datenbank mit dem übergebenen Wert umgehen soll. Der `TemporalType` ist in Abschnitt 4.1.3 beschrieben.

TemporalType

- `setParameter(?, Calendar cal, TemporalType tt)`: Hier wird ein `java.util.Calendar` an die Abfrage gebunden. Auch hier beschreibt *tt*, wie mit *cal* umgegangen werden soll.

Alle Varianten werfen eine `IllegalArgumentException`, wenn der erste Parameter nicht zur Abfrage passt oder wenn der übergebene Wert einen unpassenden Typ hat.

4.8.3 Massenoperationen

In der Java Persistence API gibt es nun endlich auch die Möglichkeit, Massenupdates oder das Löschen durch die Datenbank in der Sprache selbst auszudrücken. In EJB 2.x ist das Löschen oder Aktualisieren von vielen Datenbankeinträgen nur möglich, indem diese Einträge in den Speicher geladen und aktualisiert oder gelöscht werden. Speziell im Fall der Löschung ist dies ein großer Overhead.

Diese Massenoperationen werden auch via `create*Query()` vorbereitet, wobei nur Update- oder Delete-Statements erlaubt sind. Diese Statements werden in Abschnitt 5.3 ab Seite 150 beschrieben. Die so vorbereitete Query kann dann via `setParameter()` parametriert werden. Angestoßen werden die Massenoperationen dann allerdings via `Query.executeUpdate()`. Diese Methode gibt die Anzahl der modifizierten oder gelöschten Zeilen zurück.

Listing 4.47 *Ausführen eines Massenupdates*

```
...
Query q = em.createQuery("DELETE FROM Person AS p " +
    "WHERE p.vorname LIKE :vorname";
q.setParameter("vorname","Heik_");
int rows = q.executeUpdate();
```

Zu beachten ist, dass die Massenoperationen am Entity Manager »vorbeigehen«. Dies bedeutet, dass falls in einer Transaktion bereits Daten gelesen wurden, die später durch eine Massenoperation verändert werden, die Instanzen im Speicher nicht geändert werden und es so inkonsistente Zustände geben kann. Am besten werden diese Massenoperationen nur innerhalb ihrer eigenen Transaktion ausgeführt. Ebenso werden die Kaskadierungsoptionen bei diesen Operationen nicht angewandt.

Soll ein Entity Bean in derselben Transaktion weiterverwendet werden, muss dies neu in den PersistenceContext geladen werden. Dies kann über `EntityManager.refresh()` erfolgen (siehe Abschnitt 4.7.6).

4.9 Der Persistenz-Deskriptor: persistence.xml

In dieser Datei, die innerhalb des META-INF-Unterverzeichnisses *META-INF* eines Archivs vorhanden sein kann, können spezielle Angaben zum zu verwendenden Entity Manager etc. gemacht werden. Als Archivtypen kommen WAR, EJB-JAR, EAR oder Client-JAR in Frage. Innerhalb eines WAR-Archivs liegen die Klassen wie gewohnt in *WEB-*

INF/classes. Der Deskriptor *persistence.xml* liegt dann innerhalb dieses Verzeichnisses.

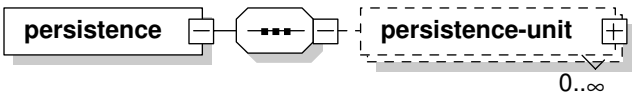


Abbildung 4.13 *Struktur des Deskriptors persistence.xml*

Listing 4.48 *Beispiel für die Datei persistence.xml*

```
<?xml version="1.0" ?>

<persistence>
  <persistence-unit name="weblog">
    <jta-data-source>java:/PostgresDS</jta-data-source>
    <properties>
      <property name="hibernate.hbm2ddl.auto"
                value="create"/>
    </properties>
  </persistence-unit>
</persistence>
```

Wie man in Abbildung 4.13 sieht, kann ein Persistenz-Deskriptor mehrere *Persistence Units* enthalten. Das XML-Schema zeigt zwar an, dass keine <persistence-unit> vorhanden sein muss, der Text im Standard fordert aber mindestens ein <persistence-unit>-Element. Das Element <persistence-unit> kennt die folgenden beiden Attribute:

- name: Der Name dieser Persistence Unit. Dieses Attribut ist Pflicht, und innerhalb eines Archives darf es keine zwei Persistence Units mit demselben Namen geben.
- description: Eine optionale Beschreibung dieser Persistenzeinheit.
- transaction-type: Mögliche Werte für diesen Parameter sind JTA und RESOURCE_LOCAL. Dies entspricht dem Typ `TransactionType`. Falls das Attribut nicht angegeben ist, wird JTA verwendet.

TransactionType

Eine Persistence Unit besteht dann aus den Elementen, die in Abbildung 4.14 gezeigt werden. In der Abbildung werden jta-datasource und non-jta-datasource als Auswahl gezeigt. Im Standard stehen sie direkt nebeneinander, so dass es theoretisch möglich ist, beide Datenquellen anzugeben. In der Praxis wird man sich aber für eine von beiden entscheiden, wie dies in der Abbildung angedeutet ist.

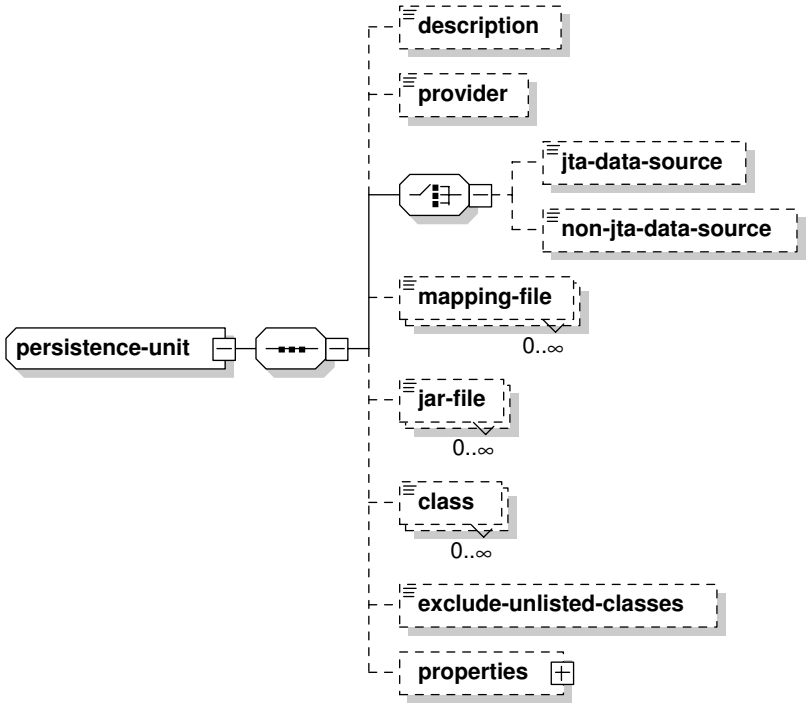


Abbildung 4.14
Struktur einer Persistence Unit

Die einzelnen Elemente sind:

- description: Eine optionale Beschreibung dieser Persistence Unit.
- provider: Der Klassenname eines Persistenz-Providers, der das Interface `javax.persistence.spi.PersistenceProvider` implementiert. Im Fall von Java EE muss er nur angegeben werden, wenn nicht der vorgegebene Provider verwendet werden soll. Im Fall von Java SE muss er immer angegeben werden – dies kann eventuell herstellerspezifisch sein.
- jta-data-source: Der JNDI-Name einer transaktionsfähigen Datenquelle.
- non-jta-data-source: Der JNDI-Name einer nichttransaktionsfähigen Datenquelle, falls keine transaktionale Datenquelle zur Verfügung steht.
- mapping-file: Der Name einer Mappingdatei, in der die Abbildungen zwischen Java-Klassen und Datenbankobjekten beschrieben werden. Diese entsprechen in etwa den *.hbm.xml*-Dateien, die von Hibernate her bekannt sind.
- jar-file: Der Name einer Jar-Datei, in der die Entity-Bean-Klassen abgelegt sind. Dieses so referenzierte JAR-Archiv muss nicht innerhalb des JARs abgelegt sein, in dem *persistence.xml* liegt.

- class: Der voll qualifizierte Klassenname einer Entity Bean.
- exclude-unlisted-classes: Normalerweise werden im Java-EE-Umfeld alle Klassen im Wurzelverzeichnis der Persistence Unit nach annotierten Klassen durchsucht. Sollen diese aus irgendwelchen Gründen nicht für die Persistenz verwendet werden, kann dieses Element auf `true` gesetzt werden. In diesem Fall müssen die zu verwendenden Klassen z.B. über <jar-file> oder <class> angegeben werden. Dieses Element ist nicht für die Verwendung in Java SE gedacht.
- properties: Die Propertys sind für herstellerspezifische Erweiterungen gedacht. Eingebettet sind <property>-Elemente, die zwei Attribute *name* und *value* haben. Ein Persistenz-Provider muss Propertys, die er nicht versteht, ignorieren.

Sind innerhalb des Deskriptors keine Angaben zum Mapping gemacht worden, werden die im Archiv vorhandenen Klassen untersucht und die Einstellungen der entsprechenden Annotationen verwendet.

Die Klassen, die via <jar-file> oder <class> geladen werden sollen, müssen bereits im Classpath vorhanden sein.

4.9.1 Sichtbarkeit von Persistence Units

Persistence Units sind nur innerhalb des definierenden Archivs sichtbar. Beispielsweise können Klassen im EJB-JAR-Archiv eines EARs nicht auf eine Persistence Unit in einem WAR-Archiv innerhalb desselben EARs zugreifen.

4.9.2 Eigenschaften für das Erzeugen eines Entity Managers

Wie in Abschnitt 4.7.1 beschrieben, kann bei `createEntityManager(Map)` eine Map mit Eigenschaften mitgegeben werden, welche Voreinstellungen in *persistence.xml* überschreiben. Tabelle 4.2 zeigt die Schlüssel der Map und den zugeordneten Eintrag aus *persistence.xml*.

Property	persistence.xml
javax.persistence.provider	provider
javax.persistence.transactionType	transaction-type Attribut
javax.persistence.jtaDataSource	jta-data-source
javax.persistence.nonJtaDataSource	non-jta-data-source

Tabelle 4.2
Eigenschaften für die Erzeugung eines Entity Managers

4.10 Der Mapping-Deskriptor orm.xml

Die Datei *orm.xml* kann dazu dienen, die Mapping-Annotationen zu überschreiben oder zu ersetzen. Dieser Deskriptor entspricht den *.hbm.xml*-Dateien von Hibernate oder den *.jdo*-Dateien in JDO.

Auch dieser Deskriptor wird hier nicht weiter vertieft. Details können in der Spezifikation oder in [10] nachgelesen werden.

5 Einführung in die Abfragesprache JP-QL 1.0

Die Abfragesprache der EJB 3.0 Entity Beans, die *Java Persistence Query Language 1.0* (im Folgenden JP-QL genannt), hat sich gegenüber ihrer Vorgängerversion, EJB-QL-2.1, stark weiterentwickelt. Der Einfluss von Hibernate ist deutlich bemerkbar. Wer Toplink, JDO oder Hibernate kennt, wird sich beim Umstieg auf JP-QL leichttun. Speziell der Sprachumfang hat sich gegenüber der Vorgängerversion vergrößert, so dass man in der Praxis deutlich seltener zu »purem« SQL greifen muss. Dies kommt der Portabilität einer Anwendung zugute.

JP-QL 1.0

JP-QL 1.0 unterscheidet zwischen Query Statements auf der einen Seite und Update und Delete Statements als Massenoperationen auf der anderen Seite. Während Erstere zur Suche nach Datensätzen dienen, können letztere Operationen über eine Menge von Datensätzen ausführen. Diese Unterscheidung wird auch bei den möglichen Operationen der Query-API des Entity Managers gemacht (`getResultList()` bzw.`getSingleResult()` und `executeUpdate()`). Dies wurde auch in Abschnitt 4.8.2 gezeigt.

Abfragen können vorformuliert werden, wie in Abbildung 4.44 auf Seite 129 gezeigt, oder können dynamisch zur Laufzeit erstellt werden.

5.1 Vorbereitende Definitionen

Bevor wir zur eigentlichen Abfragesprache kommen, noch einige Definitionen, die für Abfragen und Massenoperationen zutreffen.

5.1.1 Identifier und reservierte Wörter

Die folgenden Schlüsselwörter sind in JP-QL reserviert:

SELECT, FROM, WHERE, UPDATE, DELETE, JOIN, OUTER, INNER, LEFT, GROUP, BY, HAVING, FETCH, DISTINCT, OBJECT, NULL, TRUE, FALSE, NOT, AND,

OR, BETWEEN, LIKE, IN, AS, UNKNOWN[1], EMPTY, MEMBER, OF, IS, AVG, MAX, MIN, SUM, COUNT, ORDER, BY, ASC, DESC, MOD, UPPER, LOWER, TRIM, POSITION, CHARACTER_LENGTH, CHAR_LENGTH, BIT_LENGTH, NEW, ALL, ANY, SOME, CURRENT_TIME, CURRENT_DATE, CURRENT_TIMESTAMP, EXISTS, ALL, ANY, SOME.

Bei diesen reservierten Schlüsselwörtern wird nicht zwischen Groß- und Kleinschreibung unterschieden. Es ist aber für die Lesbarkeit hilfreich, sie trotzdem immer groß zu schreiben.

Identifier

Identifier sind Schlüsselwörter, die nicht aus der obigen Liste der reservierten Wörter stammen und die sich an das Format von Java-Identifiern halten.

IdentificationVariable

IdentificationVariable sind Variablen, die innerhalb der *FROM*-Klausel (5.2.1) deklariert werden. Diese müssen gültige *Identifier* sein und dürfen weder der Name einer Entity Bean noch ein Abstract-Schema-Name und auch kein *ejb-name* (aus dem EJB-Deployment-Deskriptor) aus derselben Persistenzeinheit sein.

5.1.2 Literale

Literal

Literale sind einfache Ausdrücke. Diese unterscheiden sich nach dem Datentyp:

String-Literal Dies ist ein Textausdruck, der in einfachen Hochkommas (') angegeben ist. Falls dieser Textausdruck selbst ein solches Hochkomma enthält, muss dieses durch ein Anführungszeichen (") ersetzt werden. String-Literale können alle Unicode-Zeichen enthalten. Anders aber als in Java sind Escape-Sequenzen (»\«) nicht zulässig.

Numerisches Literal Bei den numerischen Literalen unterscheidet man zwischen *genauen* Werten, die durch Ganzzahlwerte repräsentiert werden, und *angenäherten* Werten, welche durch Fließkommazahlen dargestellt werden. Beide Typen können einen zusätzlichen Bezeichner für die Genauigkeit haben. Dieser Bezeichner folgt dabei der Java-Sprachspezifikation.

Boolesches Literal Hier gibt es genau zwei Werte: `TRUE` und `FALSE`. Wenn sie auch üblicherweise groß geschrieben werden, ist die Groß- und Kleinschreibung dabei egal.

[1] UNKNOWN wird zwar in JP-QL nicht verwendet, ist aber trotzdem reserviert.

5.2 Abfragen von Daten: Query Statements

Ein *Query* Statement zur Abfrage von Daten operiert über ein sog. *Abstract Schema*. Dies ist bereits von EJB-QL 2.x bekannt und benennt das abzufragende Objekt unabhängig von der unterliegenden Datenbankstruktur. Das Abstract Schema wird über den *name*-Parameter von `@Entity` definiert (siehe Abschnitt 4.1). Ist das Attribut *name* nicht vorhanden, wird der unqualifizierte Klassenname der Entity verwendet. Je nach gewählter Vererbungsstrategie (siehe Abschnitt 4.3 auf Seite 86) kann sich ein Abstract Schema über mehr als eine Datenbanktabelle erstrecken.

Das Format einer Abfrage richtet sich dabei nach dem folgenden Muster:

```
SELECT ...
FROM <abstract schema> ...
[WHERE ...]
[GROUP BY ...]
[HAVING ...]
[ORDER BY]
```

Listing 5.1
Format einer Abfrage in JP-QL

Es ist also immer eine Select-Klausel notwendig, im Gegensatz zu Hibernate beispielsweise. Diese Klausel definiert die zurückzugebenden Elemente (Entitys oder einzelne Felder). Bevor wir uns diese näher anschauen, betrachten wir die From-Klausel, welche die Entitys auswählt, auf denen gearbeitet werden soll.

5.2.1 Auswahl der Entitys: From-Klausel

Mit dieser Klausel werden die Entitys ausgewählt, über welche die Abfrage erfolgen soll. Die Entitys bekommen einen symbolischen Namen, der dann in der Select-Klausel zur Auswahl der Rückgabewerte herangezogen werden kann.

```
FROM pfad [AS] variable, pfad [AS] variable ...
```

Listing 5.2
Format der From-Klausel

Ein *pfad* ist dabei in erster Linie der Abstract-Schema-Name einer Entity. Ab dem zweiten Ausdruck in der Zeile kann er aber auch einen Pfad für die Traversierung einer Relation darstellen. Dies wird gleich noch illustriert. Die *variable* stellt einen Alias dar, über den das Objekt dann in den anderen Klauseln angesprochen werden kann. Das Schlüsselwort AS ist nicht unbedingt notwendig, trägt jedoch zur Lesbarkeit der Abfrageausdrücke bei.

Listing 5.3 *Beispiele für die Verwendung der From-Klausel*

```
SELECT a FROM Adresse AS a

SELECT p FROM Person p

SELECT a,p FROM Adresse AS a, Person p ...

SELECT p.alter
FROM Adresse AS a, IN (a.inhaber) AS p ...
```

Die ersten beiden Abfragen in Listing 5.3 liefern je einfache Entitäten zurück. Die dritte Abfrage liefert ein Tupel von Entitys und die vierte das Alter einer Person, die über das Feld *inhaber* mit der Adresse verknüpft ist. Der Ausdruck `IN()` traversiert dabei die Relation von der Adresse über dieses Feld. Die Identifikationsvariable *a* muss dabei, wie gezeigt, bereits definiert worden sein.

Verknüpfen von Tabellen: Joins

Ein Join findet immer dann statt, wenn man Daten aus zwei Entitäten miteinander verknüpfen möchte. Dies kann über Relationen zwischen diesen beiden Entitäten geschehen oder auch implizit durch einen entsprechenden Ausdruck:

Listing 5.4 *Impliziter Join*

```
SELECT k
FROM Kunde AS k, Lieferant AS l
WHERE k.stadt = l.wohnort
   AND l.name = ?1
```

Hier wird das kartesische Produkt der Entitäten gebildet und die entsprechenden Einträge ermittelt. Diese Art des Joins ist sehr aufwändig für die Datenbank, da zuerst das kartesische Produkt der Tabellen ermittelt wird und daraus dann die entsprechenden Werte ausgewählt werden. Bei zwei Tabellen à 100 Zeilen besteht das Produkt bereits aus 10.000 Zeilen. Diese Form des *Inner Joins* kommt in der Praxis wegen des damit verbundenen Aufwands aber eher selten vor.

Inner Join

Gebräuchlicher ist ein (Inner) Join über eine Assoziation:

Listing 5.5 *Beispiel für einen Inner Join*

```
SELECT OBJECT (p)
FROM Person p JOIN p.adresse AS a
WHERE a.stadt LIKE ('Stutt\%')
```

Der Inner Join liefert dabei nur die Zeilen aus der Datenbank, für welche die Join-Bedingung (also die Where-Klausel) zutrifft.

Left Join

Left Joins erlauben es, Entitäten einzulesen, bei denen unter Umständen die andere Seite der Assoziation nicht vorhanden ist. Die beiden Varianten `LEFT JOIN` und `LEFT OUTER JOIN` sind dabei synonym. Die

Syntax erlaubt beide Varianten, da diese auch in der Datenbankwelt gebräuchlich sind. Angenommen man möchte alle Personen mit einem bestimmten Nachnamen und zusätzlich, falls vorhanden, die Adresse ermitteln, könnte man schreiben:

```
SELECT p FROM Person p LEFT JOIN Adresse AS a
WHERE p.name = 'Rupp'
```

Diese Left Joins werden insbesondere auch zum (Vor-)Laden von Objekten genutzt, die nicht direkt in der Rückgabe adressiert sind, von denen man aber weiß, dass sie kurz darauf benötigt werden. Dies wird dann als *Fetch Join* bezeichnet:

Fetch Join

```
SELECT p FROM Person p LEFT JOIN FETCH p.adresse
WHERE p.vorname LIKE 'Heik\%'
```

Hier wird die zur Person zugehörige Adresse direkt mit aus der Datenbank geladen und steht beim Auslesen der Person gleich zur Verfügung.

5.2.2 Definition der Rückgabe: Select-Klausel

Mit der Select-Klausel werden die zurückgegebenen Objekte bzw. Felder definiert. Dabei können sowohl über die From-Klausel definierte Entitys als auch Aggregate wie `SUM()` oder `MAX()` zurückgegeben werden, wie auch komplette JavaBeans, die selbst keine Entitys sind. Listing 5.6 zeigt das generelle Aussehen der Klausel:

```
SELECT [DISTINCT] Ausdruck [,Ausdruck] ...
```

Listing 5.6 *Format der Select-Klausel*

Über das Schlüsselwort DISTINCT kann angegeben werden, dass Duplikate in der Rückgabe eliminiert werden sollen. Einfache Beispiele für die Verwendung der Select-Klausel sind:

```
SELECT a FROM Adresse AS a ...

SELECT new MyBean(a.name, a.vorname) FROM Adresse a ...

SELECT max(a.postleitzahl) FROM Adresse AS a ...
```

Für die Migration von EJB-2.1-Finder-Abfragen gibt es auch weiterhin das Konstrukt über `OBJECT`:

```
SELECT OBJECT(a) FROM Adresse AS a ...
```

Mögliche Aggregatsfunktionen sind `MAX()`, `MIN()`, `AVG()` und `SUM()`. Des Weiteren gibt es noch die Möglichkeit, über `COUNT()` die Anzahl der Einträge einer Tabelle zu ermitteln.

Listing 5.7 *Nutzung von Aggregaten*

```
SELECT max(a.postleitzahl),min(a.postleitzahl)
FROM Adresse AS a ...

SELECT count(p.geschwister)
FROM Person p ...
```

Die beiden Abfragen in Listing 5.7 liefern also zum einen ein Tupel der größten und kleinsten Postleitzahl sowie die Anzahl der Geschwister der gesuchten Person.

5.2.3 Einschränkung der Ergebnisse: Where-Klausel

Über die Where-Klausel kann die Ergebnismenge eingeschränkt werden. Die Bedingungen, die gleich beschrieben werden, können dabei über boolesche Verknüpfungen `OR`, `AND` und `NOT` miteinander verknüpft werden, um so sehr mächtige Auswahlausdrücke zu bilden. Ausdrücke können auch geklammert werden, um Teilausdrücke in der Priorität umzugruppieren. Oftmals erhöht die Klammerung auch einfach die Lesbarkeit für den Entwickler und ist damit auch sehr hilfreich. Vor der Erläuterung der einzelnen Möglichkeiten gibt es erstmal ein Beispiel:

Listing 5.8 *Beispiel für Verknüpfungen in der Where-Klausel*

```
SELECT p FROM Person p
WHERE (p.name LIKE 'R\%')
    AND (p.adresse.plz BETWEEN '70000' AND '79999')
    AND p.alter = 20
```

Hier werden also Personen ermittelt, deren Name mit »R« beginnt, die im 7er-Postleitzahlengebiet wohnen und die 20 Jahre alt sind.

Natürlich können hier auch Platzhalter zum Einsatz kommen, welche dann in der `javax.persistence.Query` gesetzt werden können. Wie in Abschnitt 4.8.2 beschrieben, kann dies ein Positionsparameter (`?1`) oder ein benannter Parameter sein (`:parameter`).

```
SELECT p FROM Person p
WHERE p.name = :nachname
```

Hier nun eine Liste möglicher Bedingungen:

- Vergleiche: Hier stehen die Operatoren <, >, <> (dieses Symbol steht für *ungleich*), <=, >= und = zur Verfügung.
- `BETWEEN`: Testet, ob ein Wert x zwischen zwei Werten y und z ist: `x BETWEEN y AND z`. Über den Parameter NOT lässt sich auch testen, ob ein Ausdruck nicht zwischen den beiden Grenzen liegt: `x NOT BETWEEN y AND z`.

Zu beachten ist auch, dass der Vergleich die beiden Randwerte einschließt. Ein Ausdruck `age BETWEEN 18 AND 65` ist äquivalent zu $18 \leq \texttt{age} \leq 65$.

- `EXISTS`: Das Ergebnis der *ExistsExpression* ist dann wahr, wenn die angegebene Subquery einen oder mehrere Werte zurückliefert. Subquerys werden weiter unten beschrieben.
- `IN`: Testet, ob ein Wert x in der angegebenen Menge an Werten enthalten ist. Damit lässt sich die Suche nach einer Person, deren Vorname »Heiko«, »Elke« oder »Orlando« ist, so bewerkstelligen:

```
SELECT p FROM Person p
WHERE p.vorname IN ('Heiko', 'Elke', 'Orlando')
```

  Auch hier gibt es wieder die negierte Form: `x NOT IN (...)`.
- `IS EMPTY`: Mit diesem Ausdruck kann geprüft werden, ob eine Rückgabemenge leer ist. Angenommen man möchte Personen in einer Datenbank ermitteln, bei denen die Adresse noch nachgetragen werden muss, dann ließe sich dies so bewerkstelligen:

```
SELECT p FROM Person AS p
WHERE p.adresse IS EMPTY
```

  Auch hier gibt es wieder eine negierte Version: `x IS NOT EMPTY`.
- `IS NULL`: Hiermit lässt sich ermitteln, ob der angegebene Ausdruck NULL ist oder nicht. Auch hier gibt es eine negierte Version `IS NOT NULL`.
- `LIKE`: ermöglicht den Vergleich einer Zeichenkette mit einem vorgegebenen Muster. Dieses Muster kann die beiden Wildcards »%« und »_« enthalten. Wie in SQL steht der Unterstrich für ein Zeichen mit beliebigem Wert, während das Prozentzeichen für eine beliebige Zeichenkette steht.

```
SELECT p FROM Person p
WHERE p.name LIKE('Heik\%')
```

  Soll der angegebene Ausdruck mit einem Muster verglichen werden, das eines der beiden Wildcardzeichen enthält, können diese über die Angabe eines *EscapeCharacters* maskiert werden: `Eingabe LIKE '%\_' ESCAPE '\'` liefert also nur einen Match bei Eingaben, die mit einem Underscore enden. Würde der Unterstrich nicht maskiert, würde jede Eingabe mit mindestens einem Zeichen zutreffen.
- `MEMBER`: Mit diesem Ausdruck kann geprüft werden, ob eine Entität in einer Menge von Entitäten enthalten ist. Ist der Ausdruck, in dem gesucht wird, leer oder `NULL`, liefert der Gesamtausdruck *un-*

bekannt zurück. Der Ausdruck kann sowohl als `MEMBER` geschrieben werden als auch als `MEMBER OF`. Die Semantik ändert sich dabei nicht. Die negierte Form ist `NOT MEMBER`. Ein Beispiel (aus FacadeSessionBean.istPosterFuerBlog):

```
SELECT w FROM Weblog AS w, Poster AS p
WHERE p MEMBER OF w.nutzer
```

Weitere Funktionen umfassen `CONCAT` für das Verketten von Strings, `SUBSTRING` und `TRIM` für die Ermittlung von Teilstrings und dem Entfernen von Whitespaces aus Strings. Über `LOWER` und `UPPER` können Strings in ihre klein- oder großgeschriebenen Pendants gewandelt werden. Diese beiden Funktionen sind sehr hilfreich für Vergleiche, die unabhängig von der Groß- und Kleinschreibung durchgeführt werden sollen.

```
SELECT * FROM Person AS p
WHERE a.name LIKE (LOWER ('RuPp'))
```

Schließlich können mit `LENGTH` und `LOCATE` die Länge eines Strings und der Ort eines Strings in einem anderen String ermittelt werden.

Für numerische Werte können mit `ABS`, `SQRT` und `MOD` der absolute Wert, die Wurzel und der Modulus des Arguments ermittelt werden. Mit `SIZE` lässt sich die Anzahl der Elemente in einer Collection (also der n-Seite einer Relation) ermitteln.

Schließlich kann über `CURRENT_DATE`, `CURRENT_TIME` und `CURRENT_TIMESTAMP` das aktuelle Datum, die aktuelle Zeit und der aktuelle Timestamp des Datenbankservers ermittelt werden.

Subquerys

Subquery

Neu in JP-QL hinzugekommen sind auch Subquerys, mit denen sich komplexe Ausdrücke auswerten und als »Eingabe« der Where-Klausel nutzen lassen.

Subquerys sind dabei im Prinzip vollständige Querys, die innerhalb eines Teils einer *Where*-Bedingung stehen können. Einzig die OrderBy-Klausel einer vollständigen Query fehlt.

Listing 5.9 *Beispiel für die Verwendung einer Subquery*

```
SELECT DISTINCT p FROM Person p
WHERE EXISTS (
   SELECT per
   FROM Person AS per
   WHERE per = p.verheiratetMit)
```

Wie man in Listing 5.9 sieht, können in Subquerys auch Aliase verwendet werden, die in der »äußeren« Query definiert wurden. Die Abfrage

in dem Listing ermittelt also alle Personen, deren Ehepartner ebenfalls in der Tabelle zu finden sind.

In Verbindung mit den Subquerys kommen auch oft die *AllOrAny*-Ausdrücke zum Einsatz. Über diese Ausdrücke kann ein Element mit einem oder allen Resultaten einer Subquery verglichen werden. Bei `ALL` muss der Ausdruck für alle Resultate der Subquery gelten, während dies bei `ANY` nur für einige Elemente zutreffen muss; `ANY` und `SOME` sind dabei synonym.

```
SELECT a FROM Angestellte
WHERE a.gehalt > ANY (
   SELECT m.gehalt FROM Manager AS m
   WHERE m.abteilung = a.abteilung )
```

Mit dieser Query werden also die Angestellten ermittelt, deren Gehalt höher ist als das mindestens eines Managers dieser Abteilung.

5.2.4 Gruppieren von Objekten: Group-By-Klausel

Über die Group-By-Klausel können Ergebniswerte einer Abfrage gruppiert werden.

Beispielsweise können alle Mitarbeiter ausgewählt werden, die einen bestimmten Nachnamen haben. Die Ergebnismenge soll dann nach dem Wohnort des Mitarbeiters gruppiert werden.

```
SELECT p
FROM Person p, p.adresse AS a
WHERE p.name = 'Maier'
GROUP BY a.ort
```

5.2.5 Weitere Einschränkung: Having-Klausel

Über die Having-Klausel kann die Ergebnismenge weiter eingeschränkt werden. Dies kann als eine Alternative zur Where-Klausel verwendet werden. Der Unterschied zur Where-Klausel ist allerdings, dass *Having* die durch *Group By* aggregierten Werte weiter filtert, während die Where-Klausel die ursprünglichen Daten filtert.

Das nächste Beispiel zeigt die Verwendung der *Having*-Klausel:

```
SELECT a FROM Adresse AS a
WHERE a.name LIKE ('S\%')
GROUP BY a.plz
HAVING a.hausnr > 15
```

Listing 5.10 *Beispiel für die Verwendung der Having-Klausel*

Hier werden Adressen ausgewählt, deren Ortsname mit »S« beginnt. Diese ermittelten Adressen werden nach dem Kriterium *plz* gruppiert und nur die Gruppen als Ergebnis zurückgegeben, bei denen die Hausnummer größer als 15 ist.

5.2.6 Sortierung der Resultate: Order-By-Klausel

Über die *Order By*-Klausel können Ergebnisse auf- oder absteigend sortiert werden. Dabei ist es möglich, mehr als ein Sortierkriterium anzugeben. Das Format entspricht dabei

```
ORDER BY kriterium1 [ASC|DESC], kriterium2 [ASC|DESC], ...
```

Das *kriterium* gibt das Feld oder Entity an, nach dem sortiert werden soll. Über `ASC` und `DESC` kann die Sortierung auf- respektive absteigend sortiert werden. Ist keine der beiden Optionen angegeben, wird aufsteigend sortiert (ASC). Wird mehr als ein *kriterium* angegeben, erfolgt die Sortierung von links nach rechts. Das heißt, zuerst wird nach *kriterium1* sortiert und dann dessen Duplikate nach *kriterium2* und so weiter.

Zu beachten ist, dass *kriterium* in der Select-Klausel angegeben sein muss. Sei es als Teil eines dort referenzierten Abstract Schema, sei es als Feld. Der folgende Ausdruck ist nicht gültig, weil das Feld, nach dem sortiert wird (a.ort), nicht in der Select-Klausel gelistet ist.

Listing 5.11 *Ein ungültiger Ausdruck*

```
SELECT p FROM Person p, IN( p.arbeitgeber) AS a
WHERE p.ort = 'Stuttgart'
ORDER BY a.ort
```

Bei der Traversierung von Relationen auf Objektebene kann die Sortierung über `@OrderBy` erfolgen. Es ist also nicht notwendig, hierfür eine JP-QL-Abfrage zu fomulieren. `@OrderBy` ist auf Seite 107 beschrieben.

5.3 Massenoperationen

Mit diesen beiden Statements können Massendatenoperationen ausgeführt werden. Dabei ist es nicht notwendig, die entsprechenden Einträge zuerst aus der Datenbank zu laden und zu instanziieren, um sie dann zu löschen oder zu aktualisieren. Dies geschieht bei diesen Statements direkt auf der Datenbank.

Da diese Operationen am Persistenzkontext und auch an einem eventuellen Transaktionscache vorbeiarbeiten, sollten sie *nie* zusammen mit anderen Operationen innerhalb einer Transaktion ausgeführt werden. Man stelle sich den Fall vor, wo in der Transaktion zu Beginn

zehn Objekte aus der Datenbank gelesen werden. Danach werden über ein *Bulk* Update fünf von ihnen aktualisiert. Die Anwendung sieht aber noch den Originalzustand im Cache und operiert auf offensichtlich veralteten Daten.

5.3.1 Aktualisieren von Daten: Update-Klausel

Über die Update-Klausel werden Datensätze aktualisiert. Das Format der Update-Klausel folgt diesem Schema:

```
UPDATE <abstract schema> [AS] alias
SET item, item, item ....
[WHERE ...]
```

Listing 5.12 *Format der Update-Klausel*

Ein *item* folgt in diesem Fall der Form *`variable = neuer wert`*.

Im folgenden Beispiel wird also für alle Personen, die Hugo-Ferdinand heißen, das Alter auf 30 Jahre gesetzt und der Wohnort auf Stuttgart.

```
UPDATE person p
SET p.alter = '30', p.ort='Stuttgart'
WHERE p.name = 'Hugo-Ferdinand'
```

Bei den Massenupdates gibt es einen weiteren Stolperstein: Sie werden direkt in ein entsprechendes SQL Update Statement umgesetzt und laufen deshalb am Persistenzkontext vorbei. Dies hat auch zur Folge, dass

- keine Checks für das optimistische Locking durchgeführt werden.
- der Versionszähler für das optimistische Locking nicht aktualisiert wird.

Sie sollten deshalb immer nur innerhalb einer separaten Transaktion laufen. Alternativ muss sichergestellt sein, dass veränderte Objekte neu in den Persistenzkontext geladen werden. Dies kann via EntityManager.refresh() geschehen (siehe Abschnitt 4.7.6).

5.3.2 Löschen von Daten: Delete-Klausel

Die Delete-Klausel ist relativ einfach und folgt dem Schema

```
DELETE FROM <abstract schema> [AS alias]
[WHERE ...]
```

Listing 5.13 *Format der Delete-Klausel*

Über die Where-Klausel (siehe Abschnitt 5.2.3) kann die Menge der zu löschenden Entitys eingeschränkt werden. Wird keine Where-Klausel angegeben, werden alle Einträge des Abstract Schema und damit alle Objekte der beteiligten Tabellen gelöscht. Durch die Vererbung und die

gewünschte Abbildungsstrategie kann es sich hierbei um mehr als eine Tabelle handeln.

Der folgende Ausdruck löscht also alle Personen aus der Datenbank, deren Wohnort Stuttgart ist.

```
DELETE FROM Person AS p
WHERE p.ort = 'Stuttgart'
```

Zu beachten ist, dass beim Löschen von Seiten des Persistenz-Frameworks nur die Objekte des angegebenen Abstract Schema gelöscht werden (dies schließt Objekte von Unterklassen mit ein, da diese ja im selben Abstract Schema liegen). Objekte, die über eine Relation damit verbunden sind, werden nicht mitgelöscht. Ist dies gefordert, muss es über einen entsprechenden Constraint auf der Datenbankebene gelöst werden.

6 Naming

In den vergangenen EJB-Versionen war relativ viel Code notwendig, um Objekte und Referenzen im JNDI zu finden und diese dann zu instanziieren. Dieser Infrastrukturcode wird nun vom Container via Dependency Injection ersetzt. Dies hat zum einen den Vorteil des geringeren Aufwands und auf der anderen Seite auch der deutlich besseren Testbarkeit.

6.1 Dependency Injection

Alle Ressourcen werden vom Container nach dem Erzeugen der Klasse und vor Aufruf der ersten Geschäftsmethode injiziert. Auch Methoden, die über `@PostConstruct` markiert wurden, werden erst nach vollständiger DI aufgerufen.

Generell gibt es zwei Möglichkeiten, die Abhängigkeiten vom Container aus zu injizieren: direkt in die Felder oder über die Verwendung der Setter.

Nach dem Vergleich der beiden Möglichkeiten werden die einzelnen Annotationen hierfür vorgestellt.

6.1.1 Injektion in Felder

Bei der Injektion in Felder, auch *Field Injection* genannt, wird die Ressource in einem markierten Feld abgelegt.

Listing 6.1
Injektion in Felder

```
...
@EJB
MyOtherBean bean;
...
bean.doSomething();
```

Hier wird direkt das Feld *bean* vom Container gesetzt. Dabei ist es unerheblich, welche Sichtbarkeit das Feld *bean* hat.

6.1.2 Injizierung über Setter

Bei der *Setter Injection* wird für die Übergabe der Ressource ein markierter Setter aufgerufen.

Listing 6.2
Injizierung via Setter

```
...
MyOtherBean bean;
...
bean.doSomething();

@EJB
public setMyOtherBean(MyOtherBean x) {
  bean = x;
}
```

Hier erfolgt die Depencency Injection über den Aufruf des Setters, welcher dann die Variable *bean* setzt. Auch hier ist die Sichtbarkeit des Setters für die DI unerheblich.

6.2 Verwendung der Dependency Injection

JSR-220 und JSR-250 definieren eine Reihe von Annotationen, die innerhalb von Enterprise JavaBeans genutzt werden können, um andere Enterprise JavaBeans, Umgebungsvariablen oder auch Ressourcen vom Container injizieren zu lassen. Die Nutzung der Annotationen von JSR-250 ist dabei nicht auf die EJBs beschränkt.

6.2.1 Injektion anderer EJBs

Beim Aufruf anderer EJBs musste man in der Vergangenheit explizit einen Naming Context öffnen und durchsuchen, um über das Home-Objekt dann das eigentliche Remote- oder Local-Objekt zu erhalten. In EJB 3.0 kann der Container über Dependency Injection dies für einen erledigen. Hierzu muss lediglich die gewünschte EJB-Klasse mit einem `@EJB`-Tag versehen werden.

Listing 6.3 *Injektion eines Session Beans*

```
package com.acme.client;

import javax.ejb.EJB;
import com.acme.Facade;

public class MyEjbClient
{
   public void foo()
   {
     @EJB
     Facade fa;
     Object o = fa.suchePerson(name, vorname);
     // ...
   }
}
```

@EJB

Die Annotation `@EJB` kennt die folgenden Parameter, die alle optional sind:

- beanName: Dieser String-Parameter bezieht sich auf den innerhalb von `@Stateful` oder `@Stateless` angegebenen *name* bzw. das Element <ejb-name> in *ejb-jar.xml*, um gezielt ein Bean auszuwählen, falls es mehrere Beans gibt, die das gesuchte Interface implementieren.
  Die Referenz kann auch auf ein Bean in einem anderen JAR-Archiv zeigen. In diesem Fall besteht der *beanName* aus zwei Teilen. Der erste Teil ist ein Pfad zum gesuchten JAR-Archiv. Dieser Pfad ist relativ zum JAR-Archiv, in dem das aktuelle Bean ist. Der zweite Teil besteht aus dem Hashmark (#) gefolgt vom Namen des gesuchten Beans, wie beschrieben.
- beanInterface: Der Typ des Eintrags, der gesucht werden soll, falls er sich nicht über das Feld oder die Property ermitteln lässt, an dem die Annotation steht. Dies dient beispielsweise zur Unterscheidung zwischen dem Local- und Remote-Interfaces eines Beans, falls dies beide Sichten abietet.
- name: Dieser String-Parameter gibt den Namen an, unter dem das EJB im JNDI gesucht werden soll. Ist er nicht angegeben, wird ein containerspezifischer Wert verwendet (bzw. aus dem Namen des Feldes oder der Property ermittelt).
  Wird die Annotation auf Klassenebene für EJB-Referenzen verwendet (s.u.), gibt dies den Namen unterhalb des ENC an, über den die Referenz sichtbar sein soll.
- mappedName: Ein herstellerspezifischer Wert, auf den die Bean-Referenz gemappt werden soll.
- description: Eine Beschreibung z.B. über das referenzierte Bean.

Über `@EJB` lässt sich auch die Funktion von <ejb-ref> im Code schreiben:

Listing 6.4 *Beispiel für eine EJB-Referenz*

```
@EJB(name="ejb/OtherBean",
   beanInterface=SomeOtherLocal.class,
   beanName="SomeOtherBean")
@Stateless
public class Foo extends FooIF
{
  public void bar()
  {
    // ...

    InitialContext ic = new InitialContext();
    Object o =
       ic.lookup("java:/comp/env/ejb/OtherBean");
    SomeOtherLocal s = (SomeOtherBean)o;

    // ...
  }
}
```

Der Lookup von *java:/comp/env/ejb/OtherBean* wird also über die Annotation auf ein *SomeOtherBean* umgeleitet. Diese Indirektion ist grafisch in Abbildung 6.1 auf Seite 159 für `@Resource` dargestellt und funktioniert mit `@EJB` genauso.

@EJBs

Soll auf Klassenebene mehr als eine `@EJB`-Annotation verwendet werden, lassen sich diese über `@EJBs` zusammenfassen.

```
@EJBs(
  @EJB(name="ejb/a", beanName="A"),
  @EJB(name="ejb/b", beanName="B")
)
@Stateless
public class Foo extends FooIF
{
  ...
}
```

Dies ist notwendig, da an einem Element eine Annotation immer nur einmal stehen darf.

6.2.2 Injektion von Ressourcen

Ressourcen, wie beispielsweise Datenquellen oder ConnectionFactorys, können über die `@Resource`-Annotation in ein Bean injiziert werden. Diese Verwendung erfolgt auf Methoden- oder Feldebene. Auf Klas-

senebene dient die Annotation zur Definition von Referenzen auf Ressourcen. Dies ist weiter unten beschrieben.

Listing 6.5 *Beispiele für die Verwendung von Ressourcen*

```
import javax.annotation.Resource;

public class MyClass
  // (1) Voll qualifizierter Eintrag
  @Resource(
    name="ConnectionFactory",
    type=QueueConnectionFactory.class
    )
  QueueConnectionFactory factory=null;

  // (2) Name wird vom Container ermittelt
  @Resource
  private DataSource aDB;

  // ...
```

@Resource

Die `@Resource`-Annotation kennt die folgenden Attribute:

- name: Der Name der zu injizierenden Ressource. Dieser wird nur zwingend benötigt, wenn die Annotation auf Klassenebene verwendet wird. Bei der Verwendung an einer Methode oder einem Feld kann der Name aus dem Methoden- oder Feldnamen ermittelt werden, wobei er immer noch in der Annotation überschrieben werden kann. Wird das Attribut an einer Klasse verwendet, ist es Pflicht. Im zweiten Eintrag in Listing 6.5 wird die Datenquelle im JNDI unter `foo.MyClass/aDB` innerhalb des Komponenten-lokalen Namensraums (`java:comp/env`) gesucht.
- type: Der Typ der Ressource. Dies ist der voll qualifizierte Java-Klassenname. Wie auch beim Namen kann der Typ bei Methoden und Feldern aus dem gekennzeichneten Element geschlossen und hiermit überschrieben werden. Wichtig ist, dass der angegebene Typ zuweisungskompatibel zum Typ des gegebenen Java-Elements ist.
- authenticationType: Dieses Attribut beschreibt, wie die Authentifizierung auf die Ressource erfolgen soll, gilt dabei aber nur für *ConnectionFactorys* und darf bei anderen Ressourcentypen nicht angegeben werden. Das Attribut kann zwei Werte annehmen: CONTAINER und APPLICATION, wobei Ersteres der Default ist.
- shareable: Dieses boolesche Attribut gibt an, ob die Ressource von mehreren Objekten gemeinsam genutzt werden kann. Eine gemeinsame Verwendung ist nur für unveränderliche Objek-

te, wie Strings oder Singleton-artige Objekte sinnvoll. Für eine Datenbankverbindung wäre solch eine gemeinsame Nutzung beispielsweise nicht sinnvoll. Das Attribut darf nur bei *ConnectionFactory* und Referenzen auf ORBs (Object Request Broker) angegeben werden.

- mappedName: Über dieses Attribut kann der Name der Ressource auf einen anderen implementierungsspezifischen Namen gemappt werden. Da Applikationsserver dieses Mapping nicht unterstützen müssen, ist dieses Attribut nicht portabel.
  Angenommen eine Datenquelle als Ressource wäre im JNDI unter `java:/DefaultDS` zu finden, könnte die Verwendung so aussehen:

```
public void foo()
{
  @Resource(mappedName="java:/DefaultDS")
  DataSource ds;

  Connection x = ds.getConnection();
  // ...
}
```

- description: Über dieses Feld kann die Ressource genauer beschrieben werden. Es hat allerdings keinen Einfluss zur Laufzeit.

@Resources

Auch diese Einträge lassen sich wieder über Einträge im Deployment-Deskriptor überschreiben. Da eine Annotation an einem Element nur einmal vorkommen darf, können über `@Resources` auch mehrere Ressourcen injiziert werden. Dies ist allerdings nur auf Klassenebene sinnvoll (und erlaubt) und nicht etwa bei Feldern.

```
@Resources({
   @Resource(name="myDB"
            type="javax.sql.DataSource"),
   @Resource(name="secondDB"
            type="javax.sql.DataSource")})
public class MyClassUsingTwoDBs {
   // ...
}
```

6.2.3 Injektion von Umgebungsvariablen

Umgebungsvariablen können ebenfalls über die `@Resource`-Annotation deklariert werden.

```
@Resource
int eineVariable;
```

Bei dieser Form wird vom Container innerhalb des Deployment-Deskriptors nach einem Umgebungseintrag für die Variable *eineVariable* gesucht und diese dem Bean zur Verfügung gestellt. Innerhalb des Deployment-Deskriptors, *ejb-jar.xml*, ist dies ein bekannter <env-entry>-Eintrag.

Es ist aber auch möglich, der Annotation direkt einen Wert mitzugeben, der als Voreinstellung genommen wird, und innerhalb des Deployment-Deskriptors überschrieben werden kann.

```
@Resource
int eineVariable = 7;
```

Falls in diesem Fall kein zusätzlicher <env-entry>-Eintrag im Deployment-Deskriptor angelegt wird, kann der Wert von *eineVariable* nicht durch explizite JNDI-Suche via `InitialContext.lookup()` ermittelt werden.

6.2.4 Definition von Referenzen auf Ressourcen

Wie schon erwähnt, kann `@Resource` auf Klassenebene genutzt werden, um Referenzen auf Ressourcen im ENC zu definieren. Diese können dann im Code referenziert werden. Ändert sich die Zielressource, muss der Code nicht geändert werden. Es ist lediglich notwendig, im ENC die Referenz auf die Zielressource zu ändern.

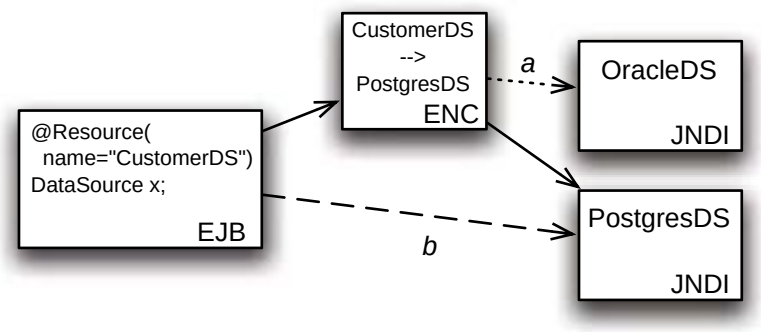


Abbildung 6.1
Nutzung einer Ressourcenreferenz

Im EJB in Abbildung 6.1 existiert ein Verweis auf eine Datenquelle mit Namen *CustomerDS*. Dieser Verweis wird im ENC auf die Zieldatenquelle mit Namen *PostgresDS* umgemappt. Soll nun die Datenquelle *OracleDS* genutzt werden, muss lediglich das Mapping im ENC auf die andere Datenquelle umgehängt werden, wie dies der gepunktete Pfeil (*a*) zeigt. Wäre die Referenz auf die Datenquelle *PostgresDS* hart im EJB einkodiert (speziell im Fall eines expliziten Lookups), wie dies der gestrichelte Pfeil *b* andeutet, müsste der Code geändert und neu kompiliert werden.

Listing 6.6
Beispiel für die Nutzung einer Referenz

```
@Resource(name="jdbc/CustomerDS",
  mappedName="java:/PostgresDS",
  type="javax.sql.DataSource")
@Stateless
public class MyEJB implements SomeIF
{
  public void versionA()
  {
    // ...
    InitialContext ic = new InitialContext();
    Object o =
        ic.lookup("java:comp/env/jdbc/CustomerDS");
    DataSource x = (DataSource) o;
    // ...
  }

  public void versionB()
  {
    @Resource(name="jdbc/CustomerDS")
    DataSource x ;
    // ...
  }
}
```

In Listing 6.6 zeigt die Methode *versionA()* einen expliziten Lookup auf die auf Klassenebene definierte Datenquelle *CustomerDS*. In *versionB()* wird die Datenquelle direkt in das Feld *x* injiziert.

6.2.5 Ressourcentypen im Deployment-Deskriptor

Wie man gesehen hat, kann `@Resource` für verschiedene Arten von Ressourcen genutzt werden. Tabelle 6.1 fasst die Ressourcen und die dafür gültigen Java-Typen zusammen.

Tabelle 6.1
Ressourcentypen

DD-Element	Java-Typ
env-entry	Objektversionen der Basistypen aus dem Paket `java.lang`: ❏ Boolean ❏ Byte ❏ Character ❏ Double ❏ Float ❏ Integer ❏ Long ❏ Short ❏ String
service-ref	Referenzen auf Webservices: ❏ javax.xml.rpc.Service ❏ javax.xml.ws.Service ❏ javax.jws.WebService
resource-ref	Referenzen auf allgemeine Ressourcen, insbesondere ConnectionFactorys: ❏ javax.jms.ConnectionFactory und Unterklassen ❏ javax.mail.Session ❏ java.net.URL ❏ javax.resource.cci.ConnectionFactory ❏ javax.sql.DataSource ❏ org.omg.CORBA_2_3.ORB ❏ sonstige ConnectionFactorys
message-destination-ref	Verweis auf JMS-Queues und Topics ❏ javax.jms.Queue ❏ javax.jms.Topic,
resource-env-ref	Verweise auf Ressourcen, die lokal zur Komponente liegen: ❏ javax.resource.cci.InteractionSpec ❏ javax.transaction.UserTransaktionen ❏ Alles, was sonst nirgends hinpasst

6.3 Naming in Clients

In Java EE Application Clients, die in einem EJB Client Container laufen, kann die Dependency Injection genauso wie in EJBs genutzt werden, um Referenzen auf Session Beans und Ressourcen zu erhalten. Allerdings müssen die Felder, in die injiziert wird, `static` sein, was sie z.B. in Servlets oder EJBs nicht sein dürfen.

Listing 6.7
Dependency Injection im Application Client

```
public class MyClient
{
  @EJB
  private static Facade facade;

  public static void main(String[] args)
  {
    facade.machWasAufDemServer();
  }
}
```

Läuft der Client nicht in einem EJB-Container, oder funktioniert die Dependency Injection nicht, können die EJBs auch wie gewohnt im JNDI gesucht werden. Allerdings ist es nun auch hier nicht mehr notwendig, sich über das Remote Home ein Remote-Objekt zu kreieren, sondern man kann direkt mit dem Ergebnis des Lookups arbeiten:

Listing 6.8
EJB-Lookup aus einem Client heraus

```
...
InitialContext ic = new InitialContext();
BeanRemoteInterface rem = (BeanRemoteInterface)
  ic.lookup(BeanRemoteInterface.class.getName());
rem.doSomething();
...
```

Beim Lookup in Listing 6.8 wird der implizit vergebene Name des Remote Interfaces verwendet. Es ist natürlich auch möglich, einen explizit gebundenen Namen zu verwenden. Wie eine Ressource explizit im JNDI gebunden wird, ist leider auch in EJB 3.0 noch containerspezifisch.

6.4 JNDI-Suche in EJBs

Innerhalb von Message-Driven und Session Beans muss für den Lookup nicht unbedingt ein `InitialContext` für die Suche erzeugt werden. In EJB 3.0 gibt es ebenfalls die Möglichkeit, die Suche über die neu hinzugekommene Methode `lookup(String name)` der Klasse `EJBContext` und ihrer Subklassen durchzuführen. Hierbei wird allerdings nur der

Enterprise Naming Context (ENC) durchsucht, welcher im JNDI unter `java:/comp/env/` zu finden ist.

Listing 6.9
Suche im JNDI über den EJBContext

```
@Stateless
public class MySessionBean implements MySessionIf
{
  @Resource
  SessionContext ctx;

  public void doSomething()
  {
    // Lookup von java:comp/env/jdbc/CustomerDS
    Object o = ctx.lookup("jdbc/CustomerDS");
    ...
  }
}
```

6.4.1 Wegfall des PortableRemoteObject

In EJB 2.1 und Vorgängerversionen ist es auf Grund der Corba-Wurzeln der EJBs notwendig, Remote Objects und Remote Homes, die man über einen JNDI-Lookup erhalten hat, via `javax.rmi.PortableRemoteObject.narrow()` auf den Zieltyp zu wandeln. Dies ist für EJB-3.0-Objekte nicht notwendig.

Erfolgt der Lookup eines EJB 2.1 Session Beans über den EJB-Kontext, ist diese Umwandlung ebenfalls nicht notwendig, so dass ein einfacher Java-Cast ausreichend ist.

7 Webservices

Parallel zum EJB-3.0-Standard wurde innerhalb von Java EE 5 mit JSR-181 auch die Entwicklung der Webservices vereinfacht. In J2EE 1.4 sind noch einige zusätzliche Dateien wie z.B. *webservices.xml* und externe Werkzeuge notwendig, um einen Webservice zum Laufen zu bringen; die Entwicklung gestaltet sich dabei eher aufwändig (siehe z.B. [18]). In Java EE 5 kann die Steuerung des Verhaltens nun ebenfalls über Annotationen erfolgen. Applikationsserver können dann intern das entsprechende Tooling haben, um beispielsweise beim Deploy einer Anwendung WSDL-Dateien aus den Annotationen erstellen zu können. Es ist also nicht mehr unbedingt notwendig, das Java Webservices Development Pack herunterzuladen, um Webservices implementieren zu können, sondern in vielen Fällen wird es ausreichen, ein existierendes Stateless Session Bean als Webservice zu kennzeichnen, um die gebotenen Dienste entsprechend zu exportieren.

Alternativ können auch Deploy Tools diesen Umsetzungsschritt erledigen. Natürlich ist es auch weiterhin möglich, ausgehend von einer vorhandenen WSDL-Datei die Annotationen so zu gestalten, dass die Schnittstellenbeschreibung aus der WSDL-Datei eingehalten wird.

In Java EE 5 wurde neben der Optimierung der Entwicklung weiterhin starker Wert auf Interoperabilität gelegt, was sich in der Konformität zum *Basic Profile 1.1* ausdrückt [24].

Dieses Kapitel beschreibt nur die »neuen« Webservices mit JAX-WS 2.0 in Verbindung mit JSR-181. Und hier auch nur die Verwendung in Verbindung mit EJB 3.0 Session Beans. Für andere Arten von Webservices und die Verwendung im Rahmen von Servlets sei auf weiterführende Literatur verwiesen.

Dieses Kapitel gibt als Erstes einen Einblick in das Programmiermodell der Webservices, gefolgt von der Implementierung von serverseitigen Diensten. Nach einem Abschnitt über die Verwendung von Session Beans als Webservice Client folgt noch ein Abschnitt über Handler, die Interceptoren für Webservices.

7.1 Übersicht über das Programmiermodell

Abbildung 7.1 zeigt die Komponenten des Programmiermodells sowohl auf der Serverseite (links) als auch im Client (rechts).

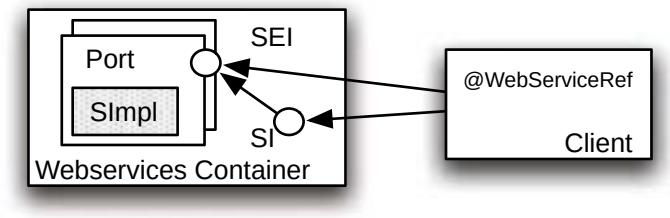


Abbildung 7.1
Client- und Serversicht der Webservices

Diese Komponenten werden durch JAX-WS 2.0 (JSR-224) und JSR-109 (Webservices for Java EE 1.2) vorgegeben. Über die Annotationen aus JSR-181 kann die Entwicklung der Dienste deutlich vereinfacht werden.

7.1.1 Serverseite

Auf der Serverseite bietet das *Service Interface* (SI) einen Zugang zu den sog. *Ports*, welche die Dienste implementieren. Der Zugang zur eigentlichen Implementierung ist über das *Service Endpoint Interface* (SEI) definiert. Im Server wird das SEI durch ein Java Interface realisiert. Innerhalb des Webservice Containers kann es mehrere Ports und SEI geben. Die Implementierung des Dienstes (SImpl) kann sowohl über ein Servlet als auch über ein Stateless Session Bean erfolgen.

In Java EE 5 ist es nun möglich, nur die Implementierung *SImpl* zu schreiben und mit Annotationen zu markieren. Die anderen Artefakte können dann vom Container generiert werden.

7.1.2 Clientseite

Das Programmiermodell definiert nicht nur die Serverseite, sondern auch den Zugriff von Jave-EE-Komponenten auf Webservices. Ein Client kann dabei das Service Interface (SI) injiziert bekommen oder im JNDI suchen und nutzt die erhaltene Information, um dann auf das SEI zuzugreifen. Auch hier kann der Zugriff wieder aus EJBs oder Servlets heraus erfolgen.

Auf der Clientseite kann eine WSDL-Datei des Services vorhanden sein, so dass die angebotenen Methoden denen der WSDL-Datei entsprechen. Ohne WSDL-Datei kann der Webservice auf eine generische Art und Weise angesprochen werden.

Für den Client ist ein Webservice immer vorhanden. Er muss also nicht erst einen Service »erzeugen«, um ihn nutzen zu können. Der Webservice Container kümmert sich um die Bereitstellung des Dienstes und beispielsweise um das Anlegen weiterer Session Beans.

7.2 Serverseitige Webservices via Session Beans

Dieser Abschnitt beschreibt die Vorgehensweise(n) für die Entwicklung von serverseitigen Webservices. Als Erstes wird die Ausgangslage besprochen, gefolgt von der Definition eines Services und der Webservice-Operationen. Anschließend werden noch die Parameter für das Mapping auf das SOAP-Protokoll beschrieben.

7.2.1 Vorgehensweisen

Dieser Abschnitt beschriebt die drei verschiedenen Ausgangslagen bei der Entwicklung der Services. In den seltensten Fällen ist es ja so, dass man einen Dienst ohne vorliegendes WSDL entwickeln kann. Hier ist das Vorgehen ein anderes, als wenn die WSDL-Datei nicht vorhanden ist.

Start mit einem Session Bean

Bei diesem Vorgehen startet der Entwickler mit einem (existierenden) Stateless Session Bean. Alle Methoden dieses Bean werden dann über `@WebService` als Webservice exportiert. Sollen nicht alle Methoden als Webservice zur Verfügung stehen, müssen die zu exportierenden Methoden über `@WebMethod` gekennzeichnet werden.

Danach wird das Session Bean entweder durch einen Präprozessor geschickt, der die restlichen Artefakte wie die WSDL-Datei und die SEI und SI erzeugt. Alternativ kann dies vom Applikationsserver auch beim Deployment dynamisch erfolgen. Es ist dabei möglich, die Generierung des WSDL durch Annotationen an vorhandene Anforderungen anzupassen.

Werden Webservices auf der »grünen Wiese« entwickelt, reicht dieses Vorgehen bereits aus, um den Webservice bereitzustellen.

Start mit einer WSDL-Datei

Liegt bereits eine WSDL-Datei vor, die den Webservice beschreibt, erzeugt man zuerst (möglichst mit Hilfe von Werkzeugen) das SEI und dann das Session Bean, welches dieses SEI implementiert.

Start mit Session Bean und einer WSDL-Datei

Dieses Vorgehen ähnelt dem Start mit einem Session Bean, wobei die Namen der Java-Methoden und Methodenparameter über Annotationen an die Namen im WSDL-Dokument angepasst werden.

7.2.2 Definition eines Web Services

@WebService

Die Annotation `@WebService` dient dazu, eine Klasse als Webservice zu definieren, und kennt fünf Parameter, die alle fakultativ sind. Steht diese Annotation alleine an der Klasse, werden alle öffentlichen Methoden als Webservice sichtbar.

- name: Der Name des Webservice. Dieser wird im WSDL als Name des Ports angegeben. Ist der Parameter nicht definiert, wird der unqualifizierte Name der Klasse genommen. Je nach Implementierung wird noch *Port* an den Namen angehängt.
- targetNamespace: Der XML-Namespace für die generierten WSDL- und XML-Elemente. Dieser Eintrag ist implementierungsspezifisch und wird meist über den Namen des Pakets abgebildet, in dem die Implementierungsklasse des Webservice liegt.
- serviceName: Der Name des Services, wie er im <wsdl:service>-Element aufgeführt wird. Ist der Parameter nicht gesetzt, wird der unqualifizierte Name der Implementierungsklasse gefolgt von *Service* verwendet.
- wsdlLocation: Eine URL, die den Ort einer extern definierten WSDL-Datei angibt. Ist eine solche Datei angegeben, müssen es die Entwicklungswerkzeuge melden, wenn der implementierte Webservice nicht diesem WSDL entspricht.
- endpointInterface: Der qualifizierte Klassenname einer Schnittstelle, die das SEI bezeichnet. Ist das SEI angegeben, wird das WSDL für den Service aus dieser SEI-Klasse erstellt.

Listing 7.1 *Definition eines Webservices ...*

```
@WebService(targetNamespace="http://bsd.de/weblog/")
@Stateless
public class FacadeSessionBean implements Facade
{
  @WebMethod
  public int anzahlArtikelInBlog(String blog)
  {
    ...
  }
}
```

```
<definitions name="FacadeSessionBeanService"
      targetNamespace="http://bsd.de/ws/">
  ...
  <service name="FacadeSesionBeanService">
    <port binding="tns:FacadeSessionBeanBinding"
      name="FacadeSessionbeanPort">
      <soap:address
       location='http://lilly:8080/weblogws/FacadeSessionBean'/>
    </port>
  </service>
</definitions>
```

Listing 7.2
... und das generierte WSDL (Ausschnitt)

Wird die Entwicklung nicht von einer Java-Klasse aus gestartet, sondern von einem existierenden WSDL-Dokument, muss die `@WebService`-Annotation über den Parameter *endpointInterface* auf ein SEI zeigen, welches aus dem WSDL generiert wurde. Die Implementierungsklasse muss alle Methoden des SEI implementieren, allerdings muss keine direkte `implements`-Beziehung bestehen.

Webservices können zusätzlich noch Handler erhalten. Diese Handler sind quasi Interceptoren auf Ebene der Webservices und werden in Abschnitt 7.4 beschrieben.

7.2.3 Deklaration der Operationen

Methoden, die als Webservice-Operationen exportiert werden sollen, werden über `@WebMethod` markiert. Diese Methoden müssen `public`, aber weder `static` noch `final` sein. Die Umsetzung der Java-Methoden in Webservice-Operationen bzw. WSDL-Einträge kann über einige weitere Annotationen und diese beiden optionalen Parameter für `@WebMethod` beeinflusst werden.

Sobald `@WebMethod` in einer Klasse angegeben ist, werden nur noch so markierte Methoden als Webservice-Operationen bekannt gemacht.

@WebMethod

Auch `@WebMethod` kennt wieder einige Parameter:

- operationName: Der Name der Operation, wie er im WSDL innerhalb von <wsdl:operation>-Elementen erscheinen soll. Ist dieser Parameter nicht gegeben, wird der Java-Name der Methode verwendet.
- action: Eine Action für diese Operation. Beim SOAP Binding wird dieser Wert in den SOAP Action Header übernommen. Der Default ist dabei leer.
- exclude: Die Methode soll nicht als Webservice exportiert werden. Dieser Parameter kann nützlich sein, wenn die Methode von einer anderen erbt, welche als Webservice zugänglich gemacht

wird. Ist dieser Parameter gegeben, dürfen die beiden anderen nicht verwendet werden.

Listing 7.1 auf Seite 168 zeigt die Nutzung von `@WebMethod`.

Aufrufe ohne Rückgabewert

@Oneway

Normalerweise wird ein Webservice synchron aufgerufen. Sprich, der Aufrufer wartet, bis der Server die Anfrage bearbeitet hat. Wenn man weiß, dass auf das Ergebnis des Webservice-Aufrufs nicht gewartet werden soll, kann eine Methode mit `@Oneway` gekennzeichnet werden. Der Client kann dann direkt nach dem Absetzen des Aufrufs seine Verarbeitung fortführen. Falls die Signatur der mit `@Oneway` markierten Methode einen Rückgabewert anders als `void` hat, muss das Webservice-Subsystem einen Fehler beim Deployment melden, was normalerweise im Abbruch des Starts der Anwendung endet. Diese Annotation hat keine Parameter.

Für die mit `@Oneway` gekennzeichneten Methoden wird im WSDL innerhalb von <operation> kein <output>-Element erzeugt.

Listing 7.3 *Beispiel für die Verwendung von @OneWay*

```
@Stateless
@WebService
public class LogOneWay implements LoggerIf
{
   @WebMethod
   @Oneway
   public void logOneWay(String logString)
   {
       MyLogger.log(logString);
   }
}
```

Mapping einzelner Parameter

@WebParam

Das Mapping einzelner Parameter der Java-Methode auf die Namen im WSDL kann über `@WebParam` beeinflusst werden. Diese Annotation kennt fünf optionale Argumente:

- name: Der Name des Parameters, wie er im WSDL hinterlegt wird. Die Regeln sind hier etwas kompliziert und hängen vom SOAP Binding ab. Default ist der Operation-Name aus `@WebMethod.operation`, falls die Operation Document-Style ist und der Parameterstil BARE ist. Ansonsten werden die Argumente einfach mit `arg`*n* mit 0 beginnend in der Reihenfolge der Parameter

durchnummeriert. Falls der Modus INOUT oder OUT ist, muss ein Name explizit angegeben werden.

- targetNamespace: Der XML-Namespace für den Parameter. Dieser wird nur benötigt, wenn die via Webservice ausgetauschten Nachrichten als Dokument kodiert sind. Als Standard wird der Namespace des Dienstes an sich verwendet.
- partName: Der Name für <wsdl:part>, falls der Dienst RPC-Style verwendet oder Document-Style mit BARE-Parametern (s.u.). Ansonsten wird der Wert von WebParam.name verwendet.
- mode: Parameter sind standardmäßig nur für den Versand von Daten an den Service gedacht. Über den *mode* kann dieses Verhalten geändert werden. Der *mode* ist dabei vom Typ `WebParam.Mode` und kennt die Werte

WebParam.Mode

  - IN: Der Parameter dient nur zur Eingabe.
  - INOUT: Der Parameter dient zur Ein- und Rückgabe.
  - OUT: Über den Parameter werden nur Werte vom Dienst zurückgegeben, aber keine Eingaben erwartet.

  INOUT und OUT sind nur für RPC-kodierte Nachrichten erlaubt, und auch nur dann, wenn der Parameter aus dem Nachrichtenkopf stammt.
- header: Steht dieser Wert auf `true`, wird der Parameter im Nachrichtenkopf gesucht, ansonsten im Body. Default ist hier `false`.

Listing 7.4 zeigt ein Beispiel für die Verwendung von `@WebParam`. In der Praxis wird man dieses Mapping nur anwenden wollen, wenn bereits ein WSDL vorhanden ist, für das die Defaults nicht passen.

Listing 7.4 *Beispiel für das Mapping eines Parameters*

```
@Stateless
@WebService
public class WSTest implements IWSTest
{
  @WebMethod
  public void log(
      @WebParam(name="logInPut", header=true)
     final String in
    )
  {
      // ...
  }
}
```

Der Parameter in Listing 7.4 würde also als *logInPut* im Kopf einer Nachricht erscheinen.

Mapping des Ergebnisses

@WebResult

Analog zu `@WebParam` lässt sich über `@WebResult` das Mapping des Ergebnisses modifizieren. Hier sind auch wieder vier optionale Parameter möglich:

- name: Der Name des Rückgabewertes, wie er im WSDL erscheint. Ist die Operation im Document-Stil verfasst mit dem Parameterstil BARE, wird als Standard der Name der Operation mit angehängtem *Response* verwendet. Ansonsten wird standardmäßig *return* verwendet.
- partName: Der Name des <wsdl:name>-Elements, in dem dieser Rückgabewert abgelegt wurde. Dieser wird nur verwendet, wenn der Webservice RPC-Stil ist oder bei Document Style, wenn der Parameterstil BARE ist.
- targetNamespace: Der Namespace des Rückgabewerts. Standard ist hier der Namespace des Webservice. Dieser Namespace wird nur verwendet, wenn die Operation Document-Style ist oder der Rückgabewert im Header zu suchen ist.
- header: Gibt an, ob der Parameter im Header zu finden ist oder im Body der Nachricht. Default ist false, so dass der Parameter im Body gesucht wird.

Listing 7.5 zeigt ein Beispiel für die Verwendung von `@WebResult`. Auch hier wird man dieses Mapping nur anwenden wollen, wenn bereits ein WSDL vorhanden ist, für das die Defaults nicht passen.

Listing 7.5 *Beispiel für das Mapping des Rückgabewerts*

```
@Stateless
@WebService(targetNamespace="http://com.acme.ws")
public class WSTest implements IWSTest
{
  @WebMethod
  @WebResult(name="resultat",
    targetNamespace="http://com.acme.ws.math")
  public int add(int a, int b)
  {
      return a + b;
  }
}
```

Der Rückgabewert für die Methode *add* würde im WSDL mit dem Namen *resultat* erscheinen. Außerdem würde er statt im serviceweiten Namespace in einem eigenen Namespace (`http://com.acme.ws.math`) angelegt.

7.2.4 Mapping auf das SOAP-Protokoll

Einige der bereits vorgestellten Annotationen beinhalten bereits Informationen, wie ein Java-Element letztlich nach SOAP gemappt werden soll. Die beiden Annotationen dieses Abschnitts bestimmen nun, wie das Mapping des eigentlich vom Transportprotokoll unabhängigen Webservice auf das Transportprotokoll SOAP erfolgen soll.

@SOAPBinding

Die Annotation `@SOAPBinding` bestimmt die Parameter für die Nachrichtenkodierung. Drei Optionen stehen hier zur Verfügung:

- style: Dieser Parameter vom Typ `SOAPBinding.Style` bestimmt den Stil der Nachricht und kann entweder DOCUMENT oder RPC sein, wobei Ersteres der Default ist. *SOAPBinding.Style*
- use: Dieser Parameter vom Typ `SOAPBinding.Use` bestimmt, ob die Nachricht literal (LITERAL) oder kodiert (ENCODED) versendet werden soll. LITERAL ist hier der Standard. *SOAPBinding.Use*
- parameterStyle: Dieser Parameter vom Typ `SOAPBinding.ParameterStyle` bestimmt, ob Methodenparameter den Körper der gesamten Nachricht darstellen (BARE) oder ob sie Elemente darstellen, welche in einem Top-Level-Element eingepackt werden, das nach der Operation benannt ist (WRAPPED, Default). *SOAPBinding.ParameterStyle*

Es ist zu beachten, dass für die Konformität zu Basic Profile 1.1 nur die Übertragung der Nachrichten im literal-Style erlaubt ist.

Das nachfolgende Beispiel in Listing 7.6 zeigt, wie die Bindung auf SOAP von document-literal auf rcp-literal umgestellt werden kann.

Listing 7.6 *Beispiel für ein SOAP-Binding*

```
@Stateless
@WebService(targetNamespace="http://com.acme.ws.log")
@SOAPBinding(style=SOAPBinding.Style.RPC)
public class WSTest implements IWSTest
{
  @WebMethod
  @OneWay
  public void log(String in)
  {
      MyLogger.log(in);
  }
}
```

In Version 1.1 von JSR-181 gab es auch noch SOAPMessageHandlers. Diese sind in Version 2.0 ersatzlos gestrichen worden. Der Zugriff auf die Daten des Transportprotokolls lässt sich nun mit den normalen Handlern erledigen. Diese werden in Abschnitt 7.4 weiter unten kurz beschrieben.

7.3 Verwendung von Webservices innerhalb von EJBs

Soll ein Session- oder Message-Driven Bean einen angebotenen Webservice als Client nutzen, erfolgt dies sehr ähnlich der Nutzung eines anderen Session Beans. Analog der Referenz auf ein anderes EJB via `@EJB` kann ein Webservice über `@WebServiceRef` in das System injiziert werden.

Listing 7.7
Injektion einer Referenz auf einen Webservice

```
@Stateless
@Resource(name="service/Telbuch",
    type="javax.jws.WebService",
    mappedName="TelbuchLookupService")
public class WSClientSB implements SomeIF
{
  public String sucheTel(String name)
  {
    @WebServiceRef(name="java:comp/env/service/Telbuch)
    TelbuchService tbs;

    TelefonNummer tel = tbs.getNummer(name);

    return tel.toString();
  }
}
```

Beim Deployment muss dem Webservice ein Name innerhalb des ENC (per Konvention im Subkontext *service*) zugewiesen werden. Dieser Service-Name wird dann in der Annotation referenziert. Das so gebundene Objekt ist dabei vom Typ `javax.xml.ws.Service` oder dessen Subklassen. Im Beispiel in Listing 7.7 wurde dies über `@Resource` ausgeführt.

@WebServiceRef

Die Annotation `@WebServiceRef` ist im Paket `java.xml.ws` definiert und kennt folgende Parameter, die alle optional sind:

- name: Ein Name, der den Webservice innerhalb des JNDI-ENC identifiziert. Dieser Name wird üblicherweise im Subkontext *service* abgelegt. Das Anlegen des Dienstes im JNDI ist Arbeit des Deployers.
- wsdlLocation: Der (lokale) Speicherort der WSDL-Datei, die den Service beschreibt. Der Ort ist dabei immer relativ zur Wurzel des Moduls zu sehen. Üblicherweise werden WSDL-Dateien in *META-INF/wsdl/* abgelegt.
  Alternativ kann hier auch eine http-URL angegeben werden, von der die WSDL-Datei bezogen wird.

Ist dieser Parameter nicht angegeben, wird die WSDL-Datei von der Service-Klasse bezogen, sofern diese mit einer korrekten `@WebServiceClient`-Annotation markiert wurde. Diese Annotation wird hier nicht weiter vertieft.

- ❏ type: Gibt das zu nutzende SEI an. Siehe auch unten.
- ❏ value: Gibt die zu nutzende Service-Klasse an. Siehe auch unten.
- ❏ mappedName: Ein produktspezifisches Mapping des Namens; beispielsweise auf einen global sichtbaren JNDI-Eintrag.

Ist der Webservice ein generierter Service, können *type* und *value* beide auf das Service Interface zeigen. Falls der Typ aus dem Typ des Felds oder der Methode ermittelt werden kann, an der die Annotation steht, können sie sogar unbenutzt bleiben.

Ist der Webservice nicht generiert, sondern zeigt auf ein SEI, muss der *value* auf das Service Interface (SI) zeigen, welches eine Unterklasse von `javax.xml.ws.Service` sein muss. Falls der Typ des SEI aus dem Typ des Felds oder Methode ermittelt werden kann, an der die Annotation steht, kann *type* unbenutzt bleiben. Ansonsten muss im *type* die Klasse des SEI gegeben werden.

Wie auch bei anderen Annotationen darf nicht mehr als eine `@WebServiceRef`-Annotation an einer Klasse stehen, weswegen diese Annotationen über `@WebServiceRefs` zusammengefasst werden können.

@WebServiceRefs

Am Service kann eine `@HandlerChain` (siehe unten) verwendet werden, um Nachrichten nach dem Absenden und vor dem Empfang durch das Session Bean darüber filtern zu können.

Der Client sollten immer gegen das Service Interface arbeiten und keine Stubs generieren. Außerdem sollte er immer davon ausgehen, dass der angebotene Webservice zustandslos ist.

Bei der Entwicklung eines Webservice Client kann das Werkzeug *wsimport* aus der JAX-WS-Referenzimplementierung genutzt werden, welches aus einer WSDL-Datei die entsprechenden SI- und SEI-Artefakte generieren kann.

7.4 Filter für Aufrufe: Handler

Über `@HandlerChain` kann eine Reihe von sog. *Handlern* definiert werden, über die Webservice-Aufrufe geleitet werden. Dieses Konzept entspricht den Interceptoren bei Session Beans. Handler können beispielsweise genutzt werden, um Security-Daten wie einen Benutzernamen aus dem Kopf eines Requests auszulesen bzw. beim Senden mitzugeben.

Leider ist die Konfiguration nicht ganz so elegant wie bei den Interceptoren, da die Liste der Handler in einer externen XML-Datei hinterlegt wird und nicht direkt in der Annotation angegeben werden kann.

@HandlerChain

Die Annotation `@HandlerChain` kennt zwei Parameter:

- file: Dies ist entweder eine URL oder ein relativer Dateiname zu einer XML-Konfigurationsdatei, in der die Handler definiert sind.
- name: Der Name einer in der Konfigurationsdatei definierten Handler Chain, die hier angewendet werden soll. In Verbindung mit JAX-WS darf dieser Parameter nicht verwendet werden und wird in späteren Versionen von JSR-181 komplett entfernt.

Die Annotation `@HandlerChain` kann sowohl an der Implementierung des Dienstes als auch an einer Referenz für einen Webservice (siehe Abschnitt 7.3) verwendet werden. Sie ist so implementiert, dass sie auf Ebene von Klassen, Methoden und Feldern angebracht werden kann. Allerdings ist mit JAX-WS 2.0 nur die Verwendung auf Methodenebene erlaubt. Wird sie an SEI und Implementierungsklasse angebracht, wird die Annotation nur an der Implementierungsklasse berücksichtigt. Listing 7.8 zeigt die Nutzung an einer Implementierungsklasse.

Listing 7.8
Beispiel für die Nutzung der HandlerChain

```
@Stateless
@WebService
@HandlerChain(file="handler_config.xml")
public class WsSessionBean implements SomeIF
{
 ...
}
```

Die XML-Konfigurationsdatei besteht dabei aus einem umklammernden <handler-chains>-Element, welches mindestens eine <handler-chain> enthält.

Listing 7.9
Beispiel für die XML-Konfigurationsdatei

```
<handler-chains>
  <handler-chain>< !-- erste Chain, alle Ports -->
    <handler>
      <handler-class>com.acme.MyHandler</handler-class>
    </handler>
  </handler-chain>
  <handler-chain> <!-- zweite Chain, nur log* Ports -->
    <port-name-pattern>log*</port-name-pattern>
    <handler>
      <handler-class>com.ws.OtherHandler</handler-class>
    </handler>
  </handler-chain>
</handler-chains>
```

Handler Chains können über die Elemente <port-name-pattern>, <protocol-bindings> und <service-name-pattern> auf bestimmte Ports, Protokolle oder Services limitiert werden. Ansonsten werden sie in der definierten Reihenfolge abgearbeitet, wobei diese Reihenfolge unterschiedlich ist, je nach Ort der Nutzung:

- Serviceprovider: Hier wird der am weitesten unten in der Datei stehende Handler zuerst aufgerufen.
- Nutzung am Client: Hier wird der erste Handler in der Konfiguration als Erstes aufgerufen.

Handler für Webservices an Stateless Session Beans werden bei eingehenden Nachrichten vor den Interceptoren des Session Beans aufgerufen.

7.4.1 Lebenszyklus eines Handlers

Der Lebenszyklus eines Handlers ähnelt dem von Stateless Session Beans. Abbildung 7.2 stellt ihn dar.

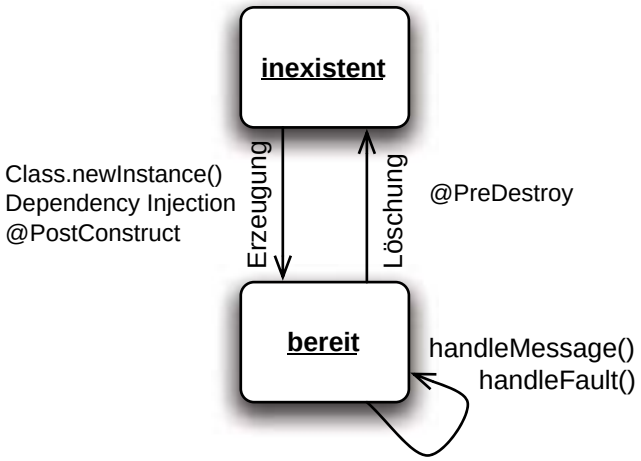


Abbildung 7.2
Lebenszyklus eines Handlers

Nachdem der Handler erzeugt wurde, kann er vom Container in einem Pool verwaltet werden. Verschiedene Handler können dabei in unterschiedlichen Threads laufen, und zwei Handler nacheinander können durch unterschiedliche Instanzen abgearbeitet werden (selbst wenn sie desselben Typs sind). Deshalb ist es nicht möglich, Daten zwischen Handlern beispielsweise über ein ThreadLocal auszutauschen. Für diesen Zweck gibt es einen `MessageContext` aus dem Paket `javax.xml.ws.handler`.

MessageContext

7.4.2 Implementierung der Handler

Prinzipiell gibt es zwei Arten von Handlern. Das Basis-Interface ist `javax.xml.ws.handler.Handler`.

- Logical: Diese Handler bekommen nichts vom darunter liegenden Transportprotokoll mit. Sie haben nur Zugriff auf die Nutzdaten und den MessageContext. Logische Handler implementieren das von Handler abgeleitete Interface LogicalHandler.
- Protocol: Mit diesen Handlern kann auch auf das unterliegende Transportprotokoll (z.B. SOAP) zugegriffen werden. Diese Handler dürfen beliebige von `Handler` abgeleitete Schnittstellen (außer LogicalHandler und Sub-Interfaces) implementieren. Dies kann beispielsweise das Interface `javax.xm.ws.handler.soap.SOAPHandler` für den Zugriff auf den SOAP-Header sein.

Wie bereits im Abschnitt über den Lebenszyklus gezeigt, kennt ein Handler zwei Methoden, die vom Container beim Eintreffen einer Nachricht beim Handler aufgerufen werden. Beide Methoden bekommen ein passendes `MessageContext`-Objekt übergeben, das Informationen zur Nachricht enthält. Der `MessageContext` ist dabei eine Map mit einigen vordefinierten Schlüsseln, deren Werte vom Container belegt werden. Die beiden Methoden sind:

- boolean handleMessage(): Diese Methode wird bei einer normalen Nachricht aufgerufen.
- boolean handleFault(): Diese Methode wird zur Behandlung von Fehlernachrichten aufgerufen.

Bei beiden Methoden führt die Rückgabe von `true` zur Fortführung der Verarbeitung und `false` zum Abbruch.

Zusätzlich gibt es noch eine Methode `close()`, die am Ende eines sogenannten *Message Exchange Patterns* aufgerufen wird und die für Aufräumarbeiten verwendet werden kann. Diese Methode sollte nicht mit der mit `@PreDestroy` markierten Methode verwechselt werden, die beim Löschen der Instanz aufgerufen wird.

Listing 7.10 *Beispielimplementierung eines Handlers*

```
import javax.xml.ws.handler.Handler;
import javax.xml.ws.handler.MessageContext;

import com.acme.MyLogger;

public class TraceHandler
   implements Handler<MessageContext>
{
   boolean handleMessage(MessageContext ctx)
   {
     Boolean bdir = (Boolean)
       ctx.get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);
     boolean dir = bdir.booleanValue();
     if (dir == false) // eingehende Nachricht
     {
        MyLogger.log("Eingehend");
        MyLogger.log("Operation ist " +
           ctx.get(MesssageContext.WSDL_OPERATION));
     }
     else // true = ausgehende Nachricht
     {
        MyLogger.log("Ausgehend");
     }
   }
}
```

Der Handler in Listing 7.10 prüft bei jedem Aufruf, ob die Nachricht eingehend ist (die Anfrage beim Server beziehungsweise die Antwort auf dem Client) und loggt bei eingehenden Nachrichten die gewünschte WSDL-Operation.

8 Hinweise zum Entwicklungsprozess

Gegenüber den Vorgängerversionen ändert sich in EJB 3.0 letztlich auch der Entwicklungsprozess, da bestimmte Artefakte nicht mehr zwingend notwendig sind. Dies wurde bereits in den vorangehenden Kapiteln angesprochen und wird hier noch einmal zusammengefasst.

8.1 Metadaten

Neben den eigentlichen Java-Klassen werden für Enterprise JavaBeans auch Metadaten benötigt. Diese können nun in der Form von Annotationen im Java-Quellcode hinterlegt werden.

An dieser Stelle hat sich während der Entwicklung des EJB-3.0-Standards auch der Irrglaube eingeschlichen, dass für eine Änderung dieser Metadaten der Quellcode geändert und neu kompiliert werden müsste. Dies ist definitiv nicht der Fall.

- Metadaten können weiterhin in XML-Deployment-Deskriptoren hinterlegt werden. Diese Metadaten im Deployment-Deskriptor überschreiben die Daten in den Annotationen.
- Annotationen werden mit in den *.class*-Dateien abgelegt. Deployment-Werkzeuge können diesen Bytecode lesen und verändern. Damit wird kein XML-Deployment-Deskriptor benötigt, und die Anwendung muss trotzdem nicht neu übersetzt werden.
- Die Indirektion von Abfragen im JNDI über den ENC besteht weiterhin, so dass selbst »hart kodierte« Referenzen auf Objekte unter `java:comp/env/` über Änderungen im Deployment-Deskriptor auf unterschiedliche Ressourcen umgebogen werden können.

8.2 Paketierung

Enterprise JavaBeans – auch die Entity Beans der Java Persistence API – werden in normale JAR-Archive gepackt. Je nach beinhalteten Beans ist ein Deployment-Deskriptor nicht unbedingt notwendig.

- Sind nur Session- und Message-Driven Beans vorhanden, kann der Deskriptor *ejb-jar.xml* entfallen, sofern alle notwendigen Metadaten über Annotationen geliefert wurden. Siehe Abschnitt 3.9 auf Seite 66.
- Für Entity Beans müssen im Java-EE-Fall über *persistence.xml* weitere Angaben über eine Persistenzeinheit gemacht werden (insbesondere der Name). Wird die JPA in einer Java-SE-Umgebung verwendet, müssen die Persistenzklassen zusätzlich über *persistence.xml* aufgelistet werden. Siehe Abschnitt 4.9 auf Seite 136.
- Metadaten für das Mapping in JPA können über Annotationen bereitgestellt werden. Diese können auch über den Deskriptor *orm.xml* geliefert werden und überschreiben dann entsprechende Annotationen. Dies ist aber nicht unbedingt notwendig.

Zusammenfassend lässt sich also sagen, dass lediglich *persistence.xml* im Fall der Nutzung von EJB 3.0 Entity Beans zwingend notwendig ist. Alle anderen Deskriptoren sind optional, sofern die Metadaten dann über Annotationen zur Verfügung gestellt werden.

8.3 Generierung von Entity Beans

Dieser Abschnitt möchte noch einige Betrachtungen zum Erstellen von EJB 3.0 Entity Beans geben. Da die Entity Beans nun POJOs sind, lassen sie sich aus vielen UML-Modellierungswerkzeugen direkt generieren. Einige der Werkzeuge sind dabei auch in der Lage, die entsprechenden Annotationen zu erzeugen.

Meist ist es allerdings so, dass man nicht auf der grünen Wiese beginnt, sondern ein Datenmodell in Form einer existierenden Datenbank oder von EJB 2.x Entity Beans vorliegt. Liegt beides vor (beispielsweise wenn man eine EJB-2.x-Altanwendung auf EJB 3 portieren möchte, sollte man sich gut überlegen, welchen der möglichen Wege man geht, da es jeweils Vor- und Nachteile gibt. Diese beiden Wege werden nun beschrieben.

8.3.1 Existierendes DB-Modell

Existiert bereits ein Datenbankmodell, besteht meist die Notwendigkeit, dieses weiterhin zu nutzen. Einige Werkzeuge bieten hier die Möglichkeit, ein Reverse Engineering zu betreiben und die Entity Beans inklusive der Annotationen aus der DDL zu generieren. Die Liste der Werkzeuge beinhaltet NetBeans [15], Hibernate Tools [6] (mit einem Plugin für Eclipse) bzw. als ant-Tasks oder auch Middlegen [13].

Das Erzeugen der POJOs für die Persistenz geht sehr schnell vonstatten. Sind bereits alle Fremdschlüsselbeziehungen in der Datenbank vorhanden, sind die Werkzeuge teilweise in der Lage, alle Annotationen zusätzlich für die Felder zu erzeugen (z.B. mit NetBeans).

Diese Werkzeuge sind allerdings ohne umfangreiche Parametrierung nicht in der Lage, aus den Tabellen- und Spaltennamen die korrekten Werte für Felder und Klassen zu erzeugen. Speziell auch bei Beziehungen werden Namen für Felder erzeugt, die sich am Namen der Fremdschlüssel orientieren. Sollen die Entity Beans dann in existierenden Code eingebaut werden, muss durch sehr viel Code manuell durchgegangen werden, um die Namen der Felder bzw. der Accessoren zu korrigieren. Einstellungen für das Cascading und Abfragen auf der Datenbank können ebenfalls nicht automatisch generiert werden.

Ein weiterer Knackpunkt ist, dass einige Werkzeuge nicht in der Lage sind, herauszufinden, ob eine Datenbankspalte Null-Werte enthalten darf oder nicht. So werden alle numerischen Felder als primitive Typen angelegt und damit diese Nullable-Bedingung ignoriert. Auch an dieser Stelle muss man mit der DDL als Referenz durch alle erzeugten Klassen gehen und die Felder mit ihren Accessoren entsprechend auf die Objekttypen umstellen.

8.3.2 Existierende EJB 2.x Entity Beans

Liegen bereits EJB 2.x Entity Beans vor, können diese als Vorlage für die EJB 3.0 Entity Beans dienen. Aus den vorhandenen Gettern und Settern können über ein Skript oder von Hand die entsprechenden Accessoren für die JPA-Beans generiert werden. Die Accessoren für die Collections sind bereits vorhanden. Aus den Deployment-Deskriptoren lassen sich ebenfalls per Skript der Name für die Tabelle und die Spalten ermitteln und die entsprechenden Annotationen an die Getter oder Felder anbringen. Aus den Findern, die in EJB-QL formuliert sind, lassen sich ebenfalls relativ einfach per Skript Named Queries erzeugen, da JP-QL ein Superset von EJB-QL ist. In diesem Fall sollten zusammengesetzte Primärschlüssel via `@IdClass` abgebildet werden, wie es in

Abschnitt 4.2.3 beschrieben ist, da hier die Abfragen nicht verändert werden müssen.

Auch bei diesem Vorgehen muss viel Code angefasst werden. Der Vorteil ist aber, dass der existierende Code bereits existiert und als Vorlage dienen kann.

Falls man die Metadaten für Entity Beans nicht über Annotationen, sondern über den Deskriptor *orm.xml* bereitstellen möchte, lässt sich dieser auch via XSLT-Transformation aus den existierenden Deskriptoren automatisch erstellen.

A Anhang

Dieser Anhang listet die Annotationen noch einmal auf und bietet eine Reihe von Referenzen für interessierte Entwickler.

A.1 Übersicht über die Annotationen

Dieser Abschnitt listet noch einmal kurz die vorkommenden Annotationen auf und gibt eine Referenz auf die Seite, auf der die Annotation erklärt wird.

Die Liste gibt es auch in zwei Sortierungen (alphabetisch und nach Paket) auf der Begleitseite zum Buch (`http://bsd.de/e3fu/`) sowie der Buch-Seite beim dpunkt.verlag (`http://www.dpunkt.de/buecher/3-89864-429-4.html`) zum Download. Damit kann man die Liste ausdrucken und beim Arbeiten neben den Rechner legen.

Für das Java-Paket, in dem die Annotation definiert ist, gibt es die folgenden Kürzel hinter dem Namen der Annotation:

- A: javax.annotation (JSR-250)
- E: javax.ejb (JSR-220)
- P: javax.persistence (JSR-220)
- S: javax.annotation.security (JSR-220)
- W: javax.jws (JSR-181)
- X: javax.xml.ws (JSR-224, JAX-WS)

Hier nun die alphabetisch sortierte Liste der Annotationen:

A-E

- `@ActivationConfigProperty` (E): Definiert einzelne Parameter, die bestimmen, wie ein Message-Driven Bean mit Nachrichten versorgt wird. (S. 42)
- `@ApplicationException` (E): Kennzeichnet die Exception als Application Exception. (S. 58)
- `@AroundInvoke` (E): Kennzeichnet die Methode als einen Interceptor. (S. 49)

- `@AssociationOverride` (P): Überschreibt Informationen einer (in einer Superklasse definierten) Relation. (S. 96)
- `@AssociationOverrides` (P): Fasst mehrere `@AssociationOverride`-Annotationen zusammen. (S. 97)
- `@AttributeOverride` (P): Überschreibt Informationen für Felder z.B. für eine eingebettete Klasse. (S. 95)
- `@AttributeOverrides` (P): Fasst mehrere `@AttributeOverride`-Annotationen zusammen. (S. 95)
- `@Basic` (P): Gibt für eine Feld an, wann es geladen werden soll und ob in der zugehörigen Spalte NULL stehen darf. (S. 74)
- `@Column` (P): Gibt zusätzliche Daten zu einem Feld, wie einen expliziten Spaltennamen oder die maximale Länge, an. (S. 75)
- `@ColumnResult` (P): Definiert das Mapping von Ergebnisspalten bei nativen Querys. (S. 133)
- `@DeclareRoles` (S): Deklariert in der Applikation verwendete Rollen. (S. 66)
- `@DenyAll` (S): Gewährt keiner Rolle Zugriff auf die Methode. (S. 65)
- `@DiscriminatorColumn` (P): Gibt den Namen einer sog. Discriminator-Spalte an, über welche Daten unterschiedlicher Klassen, die auf eine Tabelle gemappt wurden, unterschieden werden können. (S. 87)
- `@DiscriminatorValue` (P): Gibt den Wert an, der für eine Klasse in die Discriminator-Spalte geschrieben werden soll. (S. 88)
- `@EJB` (E): In die markierte Variable soll ein Session Bean injiziert werden. (S. 155)
- `@EJBs` (E): Fasst mehrere `@EJB`-Annotationen zusammen. (S. 156)
- `@Embeddable` (P): Markiert die Klasse als in ein Entity einbettbar. (S. 93)
- `@Embedded` (P): Markiert das Feld als eine eingebettete Klasse. (S. 93)
- `@EmbeddedId` (P): Definiert, dass eine eingebettete Klasse als Primärschlüssel dienen soll. (S. 82)
- `@Entity` (P): Definiert die markierte Klasse als ein Entity Bean. (S. 71)
- `@EntityListeners` (P): Listet Klassen, in denen Callbacks für Entity Manager-Operationen hinterlegt sind. (S. 127)
- `@EntityResult` (P): Definiert das Mapping der Ergebnisse einer nativen Query auf Entitys. (S. 132)
- `@Enumerated` (P): Bestimmt das Mapping von Java SE 5.0 Enums auf Entity-Werte. (S. 77)
- `@ExcludeClassInterceptors` (E): Listet klassenweite Interceptoren auf, die an dieser Methode nicht verwendet werden sollen. (S. 50)

- `@ExcludeDefaultInterceptors` (E): Fasst mehrere `@ExcludeClassInterceptors`-Annotationen zusammen. (S. 51)
- `@ExcludeDefaultListeners` (P): Schließt globale Callbacks für ein Entity aus. (S. 128)
- `@ExcludeSuperclassListeners` (P): Schließt Callbacks der Superklasse für ein Entity aus. (S. 128)

F-J

- `@FieldResult` (P): Mapping von Spalten in der Select-Klasusel einer nativen Query auf Felder einer Entity. (S. 133)
- `@GeneratedValue` (P): Bestimmt, dass der Primärschlüssel nicht von der Applikation bereitgestellt wird und extern erzeugt werden soll. (S. 83)
- `@HandlerChain` (W): Definiert eine Kette von Interceptoren für einen Webservice. (S. 176)
- `@Id` (P): Deklariert das Feld als Primärschlüsselfeld. (S. 81)
- `@IdClass` (P): Bestimmt, dass ein Primärschlüssel aus mehreren Feldern genutzt werden soll. Diese Felder müssen zusätzlich in einer Primärschlüsselklasse hinterlegt werden. (S. 82)
- `@Inheritance` (P): Bestimmt die Abbildungsstrategie einer Klassenhierarchie auf eine Menge von Datenbanktabellen. (S. 87)
- `@Init` (E): Definiert eine EJB-2.1-create-Methode für ein EJB 3.0 Stateful Session Bean. (S. 33)
- `@Interceptors` (E): Listet die auf Klassen- oder Methodenebene anzuwendenden Interceptor-Klassen. (S. 48)
- `@JoinColumn` (P): Definiert eine Fremdschlüsselspalte für Relationen. (S. 108)
- `@JoinColumns` (P): Fasst mehrere `@JoinColumn`-Annotationen zusammen. (S. 109)
- `@JoinTable` (P): Definiert eine Mappingtabelle für M:N-Relationen. (S. 110)

K-O

- `@Lob` (P): Gibt an, dass es sich bei dieser Spalte um ein BLOB oder CLOB handelt. (S. 78)
- `@Local` (E): Kennzeichnet die Geschäftsschnittstelle eines Session Beans als Local-sichtbar. (S. 35)
- `@LocalHome` (E): Gibt eine Klasse an, die das Local Home Interface für die Ansprache durch EJB-2.1-Komponenten bereitstellt. (S. 38)

- `@ManyToMany` (P): Definiert die Collection als M:N-Relation. (S. 104)
- `@ManyToOne` (P): Deklariert das Feld als N:1-Relation. (S. 103)
- `@MapKey` (P): Bestimmt das Feld, das als Schlüssel in eine java.util.Map dient. (S. 111)
- `@MappedSuperclass` (P): Bestimmt eine Klasse, die selbst kein Entity ist, aber als Superklasse von Entitys dienen kann. (S. 92)
- `@MessageDriven` (E): Definiert die Klasse als Message-Driven Bean. (S. 41)
- `@NamedNativeQueries` (P): Fasst mehrere `@NamedNativeQuery`-Annotationen zusammen. (S. 132)
- `@NamedNativeQuery` (P): Definiert eine Named Query in SQL. (S. 132)
- `@NamedQueries` (P): Fasst mehrere `@NamedQuery`-Annotationen zusammen. (S. 131)
- `@NamedQuery` (P): Definiert eine Named Query in JP-QL. (S. 131)
- `@OneToMany` (P): Definiert die Collection als 1:N-Relation. (S. 101)
- `@OneToOne` (P): Definiert das Feld als 1:1-Relation. (S. 99)
- `@Oneway` (W): Gibt an, dass diese Webservice-Operation keinen Rückgabewert hat. (S. 170)
- `@OrderBy` (P): Bestimmt, wie die N-Seite einer Relation sortiert werden soll. (S. 107)

P-T

- `@PermitAll` (S): Gibt an, dass die bezeichnete Methode oder Klasse von allen Rollen genutzt werden kann. (S. 65)
- `@PersistenceContext` (P): Injiziert einen Persistenzkontext. (S. 116)
- `@PersistenceContexts` (P): Fasst mehrere `@PersistenceContext`-Annotationen zusammen. (S. 116)
- `@PersistenceProperty` (P): Ein Schlüssel-Wert-Paar, das an den Persistenz-Provider gereicht wird. (S. 116)
- `@PostActivate` (E): Callback, der nach dem Aktivieren von Stateful Session Beans aufgerufen wird. (S. 32)
- `@PostConstruct` (A): Callback, der nach der Instanziierung und der Dependency Injection einer Klasse aufgerufen wird. (S. 44)
- `@PostLoad` (P): Callback, der nach dem Laden einer Entity aufgerufen wird. (S. 127)
- `@PostPersist` (P): Callback, der nach dem Sql-Insert einer Entity aufgerufen wird. (S. 127)
- `@PostRemove` (P): Callback, der vor nach Sql-Delete einer Entity aufgerufen wird (S. 127)

- `@PostUpdate` (P): Callback, der vor nach Sql-Update einer Entity aufgerufen wird. (S. 127)
- `@PreDestroy` (A): Callack, der aufgerufen wird, wenn das Objekt gelöscht werden soll. (S. 44)
- `@PrePassivate` (E): Markiert eine Callback-Methode, die vor dem Passivieren eines Stateful Session Beans aufgerufen wird. (S. 32)
- `@PrePersist` (P): Callback, der vor dem Sql-Insert einer Entity aufgerufen wird. (S. 127)
- `@PreRemove` (P): Callback, der vor dem SQL-Delete aufgerufen wird. (S. 127)
- `@PreUpdate` (P): Callback, der vor dem Sql-Update aufgerufen wird. (S. 127)
- `@PrimaryKeyJoinColumn` (P): Definition für Primärschlüssel bei der Abbildung einer Vererbungshierarchie. (S. 91)
- `@PrimaryKeyJoinColumns` (P): Fasst mehrere `@PrimaryKeyJoinColumn`-Annotationen zusammen. (S. 92)
- `@QueryHint` (P): Produktspezifische Hinweise an den Persistenz-Provider zur Ausführung einer Named Query. (S. 131)
- `@Remote` (E): Kennzeichnet die Geschäftsschnittstelle eines Session Beans als Remote-sichtbar. (S. 35)
- `@RemoteHome` (E): Gibt eine Klasse an, die das Remote Home Interface für die Ansprache durch EJB-2.1-Komponenten bereitstellt. (S. 38)
- `@Remove` (E): Bezeichnet eine Methode, die bei Stateful Session Beans zum Löschen des Beans dient. (S. 33)
- `@Resource` (A): Markiert eine allgemeine Ressource, die injiziert werden soll. (S. 157)
- `@Resources` (S): Fasst mehrere `@Resource`-Annotationen zusammen. (S. 158)
- `@RolesAllowed` (A): Gibt die Rollen an, die Zugriff zu einer Klasse oder Methode haben. (S. 65)
- `@RunAs` (S): Gibt die Rolle an, in der das Bean laufen soll. (S. 65)
- `@SOAPBinding` (W): Definiert die Bindung des Webservice auf das SOAP-Protokoll. (S. 173)
- `@SecondaryTable` (P): Gibt eine sekundäre Tabelle für ein Entity Bean an. (S. 72)
- `@SecondaryTables` (P): Fasst mehrere `@SecondaryTable`-Annotationen zusammen. (S. 73)
- `@SequenceGenerator` (P): Parametriert den Primärschlüsselgenerator auf Sequenzbasis. (S. 84)
- `@SqlResultSetMapping` (P): Definiert das Mapping der Resultate von nativen Querys auf Java-Felder. (S. 132)

- `@SqlResultSetMappings` (P): Fasst mehrere `@SqlResultSetMapping`-Annotationen zusammen. (S. 132)
- `@Stateful` (E): Kennzeichnet die Klasse als Stateful Session Bean. (S. 31)
- `@Stateless` (E): Kennzeichnet die Klasse als Stateless Session Bean. (S. 29)
- `@Table` (P): Definiert den Tabellennamen für das Entity Bean. (S. 72)
- `@TableGenerator` (P): Parametriert den Primärschlüsselgenerator auf Tabellenbasis. (S. 85)
- `@Temporal` (P): Definiert die Spalte als einen Zeit-Wert. (S. 78)
- `@Timeout` (E): Markiert eine Methode als Timeout-Callback. (S. 55)
- `@TransactionAttribute` (E): Setzt die Transaktionsflags (Required etc.) an eine Klasse oder Methode. (S. 60)
- `@TransactionManagement` (E): Gibt an, ob eine Ressource Bean-Managed Transactions oder Container-Managed Transactions nutzen soll. (S. 59)
- `@Transient` (P): Kennzeichnet das Feld oder die Property als nicht zu persistierend. (S. 73)

U-Z

- `@UniqueConstraint` (P): Definition einer zusätzlichen (Eindeutigkeits-)Bedingung für eine Spalte. (S. 72)
- `@Version` (P): Kennzeichnet die Spalte als Versionsspalte für das Optimistic Locking. (S. 79)
- `@WebMethod` (W): Deutet an, dass die Methode als Webservice-Operation exportiert werden soll. (S. 169)
- `@WebParam` (W): Überscheibt die Standard-Mappings für die Parameter einer Webservice-Operation. (S. 170)
- `@WebResult` (W): Überschreibt das Standard-Mapping für den Rückgabewert einer Webservice-Operation. (S. 172)
- `@WebService` (W): Deklariert ein Stateless Session Bean als Webservice. (S. 168)
- `@WebServiceRef` (X): Injiziert einen Webservice in einen Webservice Client. (S. 174)
- `@WebServiceRefs` (X): Fasst mehrere `@WebServiceRefs`-Annotationen zusammen. (S. 175)

A.2 Referenzen

In diesem Abschnitt werden noch einige Referenzen zum Thema EJB 3.0 aufgelistet, die für ein weiteres Studium nützlich sein können. Wie immer gilt bei URLs, dass diese sich ändern können. Ich versuche auf der Begleitseite zum Buch http://bsd.de/e3fu/ diese Liste akutell zu halten und auch weitere Referenzen mit aufzunehmen.

A.2.1 Implementierungen

- ObjectWeb EasyBeans: Diese Implementierung des Objectweb-Konsortiums, das auch den JOnAS-Applicationserver erstellt, bietet einen Container für Session- und Message-Driven Beans sowie eine JPA-Implementierung, die ihrerseits Hibernate oder Toplink Essentials als Persistenz-Provider nutzen kann (es ist geplant, OpenJPA ebenfalls zu unterstützen). Die EasyBeans-Implementierung steht unter der LGPL. http://easybeans.objectweb.org/
- Sun Projekt GlassFish: GlassFish ist die Referenzimplementierung von EJB 3.0. Der Server, bei dem der meiste Teil unter der CDD-Lizenz[1] steht, dient auch als Basis für Suns kommerziellen Java EE 5 Applikationsserver. Der JPA-Teil wurde von Oracle beigesteuert. Es soll in Zukunft alternativ auch möglich sein, OpenJPA zu verwenden. http://java.sun.com/javaee/community/glassfish/
- JBoss EJB 3.0 Container: JBoss hatte einige Mitarbeiter in der JSR-220-Expertengruppe, so dass diese Implementierung eine der frühesten auf dem Markt war. Als Persistenz-Provider wird hier Hibernate verwendet. Der EJB 3.0 Container ist dabei nicht nur im JBoss-Applikationsserver lauffähig, sondern kann auch in andere Server oder gar Unit-Tests eingehängt werden. http://labs.jboss.com/portal/jbossejb3
- TMax Jeus: Jeus ist der Applikationsserver der koreanischen Firma Tmax. Die Version 6 des Servers soll Java-EE-5-kompatibel werden. Auf der Webseite steht eine Version für Linux zum Download bereit. http://www.tmax.co.kr/tmax_en/jeus/index.html
- Oracle AS: Oracle stellt für seinen Applikationsserver ebenfalls eine EJB-3.0-Implementierung bereit. Die Persistenz wird dabei über das hauseigene Produkt Toplink realisiert. Eine Preview für den OracleAS 10g steht bereit. http://www.oracle.com/technology/tech/java/ejb30.html

[1] Diese Lizenz ist ein Abkömmling der Mozilla Public License und wurde ebenfalls von der Open Source Initiative (http://www.opensource.org/) als Open-Source-Lizenz anerkannt.

- Oracle Toplink JPA: Die Referenzimplementierung für den JPA-Teil des Standards. Diese Implementierung ist auch Teil des GlassFish-Servers von Sun (s.o.). http://www.oracle.com/technology/products/ias/toplink/JPA/index.html
- Bea Weblogic Server: Bea als einer der großen Hersteller von Applikationsservern wird für den Weblogic 9.2 Server auch EJB 3.0 anbieten. Aktuell gibt es eine separat herunterzuladende Tech Preview. Für die JPA-Komponente hat Bea den JDO-Anbieter Solarmetric übernommen. http://edocs.bea.com/wls/docs92/ejb30TP.html
- Hibernate: Hibernate dient als Persistenz-Provider für eine Reihe von JPA-Implementierungen und stellt selbst ein sehr erfolgreiches O/R-Mapping-Framework dar. Aufsetzend auf die Standard-Distribution gibt es Annotationen, welche die aus der JPA noch erweitern und einen EntityManager. http://www.hibernate.org/
- Apache OpenJPA: Diese Implementierung basiert auf dem KODO-Code von Bea, von dem Teile an die Apache Foundation gegeben wurden. Bea hat auch Angestellte, die an OpenJPA weiterarbeiten, so dass sich die beiden Produkte gegenseitig befruchten. Der kommerzielle KODO-Code soll gegenüber OpenJPA noch einige Erweiterungen bezüglich Skalierbarkeit und der Werkzeugunterstützung erhalten. http://incubator.apache.org/projects/openjpa.html
- Apache Cayenne: Eine eigenständige Open-Source-Implementierung eines Object Relational Mappers inklusive einer Implementierung der JPA. http://objectstyle.org/cayenne/
- Resin Amber: Amber ist eine Implementierung der JPA der Firma Resin. http://www.caucho.com/resin-3.0/amber/index.xtp

A.2.2 Dokumentation

- Java EE 5.0 API: Diese Seite bietet die Java EE 5.0 API als Javadoc-Seite online an. Auf den Java-Seiten von Sun steht ebenfalls eine Version zum Download bereit. http://java.sun.com/javaee/5/docs/api/
- JSR-109: Beschreibung des Programmiermodells für Webservice innerhalb der Java Enterprise Edition.
- JSR-154: Servlet-Spezifikation 2.4 mit Updates (»Maintenance Release«) für die Spezifikation 2.5, die keinen eigenen JSR bekommen hat. Insbesondere werden in diesem Dokument die Annotationen definiert, die in Servlets innerhalb von Java EE 5 verwendet werden können.

- JSR-181: In diesem JSR sind die Annotationen für die Webservices spezifiziert. [23]
- JSR-220: Dieser JSR definiert die Version 3.0 der Enterprise JavaBeans. Er ist in drei Teile untergliedert:
  - Simplified: In diesem Teil werden im Wesentlichen die Neuerungen von EJB 3.0 vorgestellt. [2]
  - Core: Hier wird die komplette EJB-API inklusive der Entity Beans früherer Versionen ausführlich beschrieben. Dieses Dokument richtet sich insbesondere an Implementierer von EJB-Containern. [3]
  - Persistence: Dieser Teil beschreibt die Entity Beans als Implementierung der Java Persistence API 1.0. [4]
- JSR-222: Dieser JSR definiert die Java API for XML Bindung (JAXB) 2.0. Auf diese API setzen auch die Webservices auf.
- JSR-224: JAX-WS 2.0. Dieses Dokument beschreibt die neuen Webservices, die in Java EE 5 die alten Webservices nach JAX-RPC quasi ablösen (Letztere stehen aber weiterhin zur Verfügung). [8]
- JSR-244: Der Java-EE-5.0-Mantelstandard. Dieses Dokument beschreibt die Komponenten, aus denen diese Version der Enterprise Edition besteht und in Teilen auch, wie diese miteinander interagieren.
- JSR-250: Dieser JSR definiert Annotationen, die sowohl in Java EE als auch in Java SE verwendet werden können.
- Namespaces in XML: Dieses Dokument beschreibt die Verwendung von Namespaces in XML, wie sie von den Webservices genutzt werden. `http://www.w3.org/TR/1999/REC-xml-names-19990114/`
- XML Schema: Deployment-Deskriptoren werden seit EJB 2.1 über XML Schema beschrieben. Ebenso können WSDL-Dokumente Datentypen über XML Schema definieren. `http://www.w3.org/XML/Schema`
- Java BluePrints: Sun hat mit den BluePrints Erfahrungen und Entwurfsmuster dokumentiert. Für EJB 3.0 sind diese noch am Anfang, da einige der etablierten Muster so nicht mehr notwendig sind. `http://java.sun.com/reference/blueprints/`
- Nutzung von EJB 3.0 mit Spring: Das Dokument beschreibt, wie EJB 3.0 Session Beans im Spring Container genutzt werden können. `http://springide.org/project/wiki/UsingJsr220inWebContainer`
- JAX-WS-Seiten bei java.net. Von dort kann auch die Referenzimplementierung inklusive Quellcode heruntergeladen werden.

Die Implementierung enthält auch einige Beispiele und Erklärungen, die für das Verständnis von JSR-224 hilfreich sind. `https://jax-ws.dev.java.net/`

A.2.3 Literatur

- *Enterprise JavaBeans 3.0*: Dieses Buch von Bill Burke und Richard Monson-Haefel ist die fünfte Auflage des Klassikers von Richard Monson-Haefel. Bill Burke hatte bei einer früheren Auflage bereits das JBoss-Workbook zu diesem EJB-Buch (zusammen mit Sacha Labourey) geschrieben und hat das Buch nun, da Richard nicht mehr weitermachen wollte, von ihm übernommen und für EJB 3.0 zu großen Teilen neu geschrieben. Bill ist Mitglied in der EJB-3.0-Spezifizierungsgruppe. [14]
- *EJB 3.0 professionell*: Oliver Ihns ist ebenfalls Mitglied der JSR-220-Expertengruppe und hat in der Vergangenheit eines der deutschen Referenzwerke zu EJB 2.1 geschrieben. Diese Erfahrung und die der Mitarbeit an JSR-220 sind in dieses Buch eingeflossen. [10]
- *Java Persistence with Hibernate*: Christian Bauer und Gavin King vom Hibernate-Team, die beide Mitglied in der JSR-220-Expertengruppe sind, beschreiben in diesem Buch Hibernate 3.0, das mit den JPA-Zusätzen (Annotationen und EntityManager) als Basis für einige EJB-3.0-Implementierungen dient. Das Buch ist der fast komplett neu geschriebene Nachfolger von »Hibernate in Action« [1].
- *Pro EJB 3. Java Persistence API*. Michael Keith von der Firma Oracle hat als Co-Spec-Lead die Entwicklung der JPA vorangetrieben und seine Erfahrung in diesem Buch dargestellt. [9]

Literaturverzeichnis

[1] Christian Bauer und Gavin King. *Java Persistence with Hibernate*. Manning Publications Co., Greenwich, August 2006. Beschreibt Hibernate 3.2 und die Java-Persistenz-Architektur.

[2] *EJB 3.0, Simplified API*. http://jcp.org/en/jsr/detail?id=220.

[3] *EJB 3.0, Core Contracts and Requirements*. http://jcp.org/en/jsr/detail?id=220.

[4] *EJB 3.0, Java Persistence API*. http://jcp.org/en/jsr/detail?id=220.

[5] *Hibernate Homepage*. http://www.hibernate.org/.

[6] *Hibernate Tools*. http://tools.hibernate.org/.

[7] *Java Message Service, Version 1.1*, April 2002. http://java.sun.com/products/jms/docs.html.

[8] *The Java API for XML-Based Web Services (JAX-WS) 2.0*, April 2006. http://jcp.org/en/jsr/detail?id=224.

[9] Michael C. Keith. *Pro EJB 3*. Apress, 2006. Michael Keith von Oracle ist stellvertretender Vorsitzender des EJB-3-Expertenkomitees.

[10] Oliver Ihns, Dierk Harbeck, Stefan M. Heldt und Holger Koschek. *EJB 3.0 professionell*. dpunkt.verlag, Heidelberg, voraussichtlich 2007. Oliver Ihns ist Mitglied im JSR-220-Standardisierungskomitee für EJB 3.0.

[11] Debu Panda, Reza Rahman and Derek Lane. *EJB3 in Action*. Manning Publications Co., Greenwich, 2007.

[12] Johannes Link. *Softwaretests mit JUnit*. dpunkt.verlag, Heidelberg, Januar 2005.

[13] *Middlegen*. http://boss.bekk.no/boss/middlegen/.

[14] Bill Burke and Richard Monson-Haefel. *Enterprise Java Beans 3.0*. O'Reilly & Associates, Sebastopol, 2006.

[15] *NetBeans IDE*. http://www.netbeans.org/.

[16] *Picocontainer Homepage*. http://www.picocontainer.org/.

[17] *PostgreSQL Datenbank*. http://www.postgresql.org/.

[18] Heiko W. Rupp. *JBoss*. dpunkt.verlag, Heidelberg, 2005.

[19] Martin Bakschat und Bernd Rücker. *Enterprise JavaBeans 3.0*. Spektrum Akademischer Verlag, 2007.

[20] Rima Ratel Sriganesh, Gerald Brose and Micah Silverman. *Mastering Enterprise JavaBeans 3.0*. John Wiley & Sons, 2006.

[21] *Spring Framework Homepage*. http://www.springframework.org/.

[22] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[23] *Web Services Metadata for the JavaTM Platform*. http://jcp.org/en/jsr/detail?id=181.

[24] *Basic Profile – Version 1.1*. http://www.ws-i.org/Profiles/BasicProfile-1.1-2004-08-24.html.

[25] Michael Juntao Yuan. *Lightweight Java Web Applicaton Development: Leveragin Ejb 3.0, Jsf, Pojo and Seam*. Prentice Hall PTR, 2007.

Index