Enterprise JavaBeans™ 2.0

Die Reihe Programmer's Choice

Von Profis für Profis
Folgende Titel sind bereits erschienen:

Bjarne Stroustrup
Die C++-Programmiersprache
1072 Seiten, ISBN 3-8273-1660-X

Elmar Warken
Kylix – Delphi für Linux
1018 Seiten, ISBN 3-8273-1686-3

Don Box, Aaron Skonnard, John Lam
Essential XML
320 Seiten, ISBN 3-8273-1769-X

Elmar Warken
Delphi 6
1334 Seiten, ISBN 3-8273-1773-8

Bruno Schienmann
Kontinuierliches Anforderungsmanagement
392 Seiten, ISBN 3-8273-1787-8

Damian Conway
Objektorientiertes Programmieren mit Perl
632 Seiten, ISBN 3-8273-1812-2

Ken Arnold, James Gosling, David Holmes
Die Programmiersprache Java
628 Seiten, ISBN 3-8273-1821-1

Kent Beck, Martin Fowler
Extreme Programming planen
152 Seiten, ISBN 3-8273-1832-7

Jens Hartwig
PostgreSQL – professionell und praxisnah
456 Seiten, ISBN 3-8273-1860-2

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
Entwurfsmuster
480 Seiten, ISBN 3-8273-1862-9

Heinz-Gerd Raymans
MySQL im Einsatz
618 Seiten, ISBN 3-8273-1887-4

Dusan Petkovic, Markus Brüderl
Java in Datenbanksystemen
424 Seiten, ISBN 3-8273-1889-0

Joshua Bloch
Effektiv Java programmieren
250 Seiten, ISBN 3-8273-1933-1

PEARSON Education

ADDISON-WESLEY

Stefan Denninger
Ingo Peters

Enterprise JavaBeans™ 2.0

2. Auflage

ADDISON-WESLEY
An imprint of Pearson Education
München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

Die Deutsche Bibliothek – CIP-Einheitsaufnahme

Ein Titeldatensatz für diese Publikation ist bei
Der Deutschen Bibliothek erhältlich.

Umwelthinweis:
Dieses Produkt wurde auf chlorfrei gebleichtem Papier gedruckt.
Die Einschrumpffolie – zum Schutz vor Verschmutzung – ist aus umweltverträglichem und recyclingfähigem PE-Material.

5 4 3 2 1

05 04 03 02

ISBN 3-8273-1765-7

ein Imprint der Pearson Education Deutschland GmbH,
Martin-Kollar-Straße 10–12, D-81829 München/Germany

Einbandgestaltung: Christine Rechl, München
Titelbild: Phacelia congesta, Phazelie. © Karl Blossfeldt Archiv – Ann und Jürgen Wilde, Zülpich/ VG Bild-Kunst Bonn, 2002
Lektorat: Martin Asbach, masbach@pearson.de
Korrektorat: Christine Depta, München
Herstellung: Monika Weiher, mweiher@pearson.de
Satz: reemers publishing services gmbh, Krefeld, www.reemers.de
Druck und Verarbeitung: Kösel, Kempten, www.Koeselbuch.de
Printed in Germany

Inhalt

Vorwort

Enterprise JavaBeans (Kurzform: EJB) ist die Standard Komponentenarchitektur für die Erstellung von verteilten Geschäftsanwendungen in der Programmiersprache Java. EJB bietet alle Mechanismen, die für die Erstellung von Anwendungen für den unternehmensweiten Einsatz und für die Steuerung kritischer Geschäftsprozesse erforderlich sind. Die Anwendungsentwickler können von Verteilung, Transaktionssicherung, Pooling von Netzverbindungen, synchroner und asynchroner Benachrichtigung, Multithreading, Objektpersistenz und Plattformunabhängigkeit profitieren – die Programmierung bleibt durch die Konzepte von EJB relativ einfach.

Im März 1998 wurde die erste Spezifikation der Enterprise JavaBeans von Sun veröffentlicht. In relativ kurzem Abstand folgte im Dezember 1999 die konsolidierte Version 1.1. Inzwischen hat die Komponentenarchitektur im Markt für Applikationsserver für viel Bewegung gesorgt, so dass der größte Teil der kommerziellen Applikationsserver heute EJB unterstützt. Auch im wachsenden Markt für fertige Anwendungskomponenten hat EJB sich etabliert. Von spezialisierten Einzellösungen bis hin zu vollständigen Anwendungsframeworks existiert heute ein umfassender Markt.

Im August 2001 wurde die Version 2.0 verabschiedet. Die neue Version bietet wesentliche Erweiterungen und Verbesserungen. Auf die Version 2.0 abgestimmt liegt dieses Buch nun in der 2. Auflage vor. Die 1. Auflage wurde weitgehend überarbeitet, um die folgenden Neuerungen aufzunehmen:

- Die neuen Message-Driven-Beans (EJB 2.0) bieten völlig neue Möglichkeiten zur asynchronen Kommunikation und zur parallelen Verarbeitung von Geschäftslogik. Ermöglicht wird dies durch die Integration des Java Message Service (JMS) in die EJB-Architektur.
- Local-Interfaces (EJB 2.0) ermöglichen eine Optimierung der prozessinternen Kommunikation unter Enterprise-Beans bzw. zwischen lokalen Clients und Enterprise-Beans.
- Der Persistence-Manager wurde eingeführt (EJB 2.0). Mit Beziehungen zwischen Entity Beans können jetzt komplexe Datenstrukturen modelliert werden. Die zugehörige Abfragesprache EJB QL ermöglicht das effiziente Arbeiten mit diesen Strukturen.

- Die Sicherheitskonzepte von EJB werden in einem neuen Kapitel vermittelt.
- Das Kapitel zu Aspekten der praktischen Anwendung wurde wesentlich erweitert.
- Alle Beispiele wurden überarbeitet. Dabei wurde viel Wert darauf gelegt, den Lesern das Ausführen der Beispiele zu erleichtern.

Heute kann mit gutem Recht behauptet werden, dass Enterprise JavaBeans den Erfolg der Programmiersprache Java auf den Server getragen haben. Besonders in den Bereichen Portale und Integration von Altanwendungen ist EJB zur verbreiteten Lösungsstrategie geworden. Doch der IT steht ein neuer Wandel bevor. Der Trend geht zu frei kombinierbaren Anwendungskomponenten unterschiedlicher Hersteller, die sich über Webservices verknüpfen. Für die daraus wachsenden Bedürfnisse kann die neue EJB Version 2.0 Antworten bieten. Besonders die neuen Entwicklungen in den Bereichen Marktplätze, private Börse, eProcurement und elektronischer Zahlungsverkehr können von EJB als Basisarchitektur profitieren.

Dieses Buch richtet sich an alle, die mehr über Enterprise JavaBeans erfahren wollen. Es vermittelt die der EJB-Architektur zugrunde liegenden Konzepte. Darauf aufbauend erklärt es die technischen Details und die konkrete Programmierung von Enterprise-Beans. Es wurde viel Wert darauf gelegt, das vermittelte Wissen durch umfangreiche Beispiele zu ergänzen. Der Quellcode aller Beispiele zu diesem Buch kann von der auf dem Umschlag angegebenen Quelle bezogen werden. Kenntnisse der Programmiersprache Java sowie Grundkenntnisse im Bereich der verteilten Programmierung sind Voraussetzung für die Lektüre dieses Buches.

Danksagung

Die Arbeit an der zweiten Auflage hat wieder sehr viel Spaß gemacht. Wegen der langen Diskussionen über Local-Interfaces oder Dependent-Objects mussten einige Kapitel mehrfach überarbeitet werden. Letzten Endes waren die Richtungswechsel in der Spezifikationsphase Grundlage für viele interessante Diskussionen und Erkenntnisse.

Ohne die tatkräftige Unterstützung von Kollegen und Bekannten wäre die Arbeit nur halb so spannend und nur halb so fruchtbar gewesen. Besonders Bedanken möchten wir uns bei Stefan Schulze und Alexander Greisle. Ihr stets konstruktives und äußerst kompetentes Feedback hat wesentlich zur Verbesserung der Qualität dieses Buches beigetragen. Außerdem möchten wir uns bei den Lesern der 1. Auflage bedanken, die durch ihr großes Interesse und ihr Feedback diese 2. Auflage ermöglicht haben. Zuletzt möchten wir unserem Lektor Herrn Martin Asbach vom Addison-Wesley-Verlag für die kompetente Betreuung und die gute Zusammenarbeit danken.

Stefan Denninger, Ingo Peters
München, Februar 2002

1 Einleitung

Ein Beispielszenario

Stellen Sie sich vor, Sie sind ein Anwendungsprogrammierer, der seine Dienste in einer mittelständischen Firma verrichtet, deren vier Standorte sich über mehrere europäische Länder verteilen. Diese Firma verfolgt die Strategie, Business-Anwendungen selbst zu entwickeln. Das mag auf den ersten Blick ungewöhnlich klingen. Denken Sie jedoch beispielsweise an ein Unternehmen der Automobil-Zulieferindustrie, in der heftiger Konkurrenzkampf und wachsender Preisdruck auch den Softwaresystemen höchste Flexibilität und Stabilität abverlangen. Mit den Entwicklern im Haus lassen sich diese Eigenschaften des Softwaresystems gewährleisten, was die Entscheidung des Managements für eine Eigenentwicklung rechtfertigt.

Bislang ist an jedem der vier Standorte ein eigenes Softwaresystem im Einsatz, da die Entwicklungsteams in den Niederlassungen unabhängig voneinander arbeiten.

Die Hiobsbotschaft

Eines Morgens werden Sie zu Ihrem Abteilungsleiter gerufen, der Ihnen mitteilt, dass sich im EDV-Bereich Ihres Unternehmens bahnbrechende Veränderungen ankündigen. Im Zuge der wachsenden Internationalisierung sowie aus Rationalisierungsgründen soll die im Unternehmen eingesetzte Software für alle Standorte vereinheitlicht werden. Die Entwicklerteams bleiben an den Standorten, sollen aber ab sofort unter einem neu eingesetzten Bereichsleiter zusammenarbeiten. Die bisherige, standortabhängige Software soll schrittweise durch eine Neuentwicklung abgelöst werden. Die im Unternehmen eingesetzten Insellösungen (Personalverwaltungs- und Finanzbuchhaltungssystem) sollen in das neue Gesamtsystem integriert werden. Damit sind auch Sie als Spezialist für Buchhaltungs- und Warenwirtschaftssysteme von den anstehenden Änderungen betroffen.

Die Aufgabe

Ihre Aufgabe ist es nun, für dieses neue System einen Prototyp zu entwickeln, der nach der Evaluierungsphase zu einem stabilen Systembaustein weiterentwickelt werden soll. Es handelt sich dabei um ein rudimentäres Buchungssystem. Dazu passende

Inventur- sowie Jahresabschlussmodule werden in den anderen Standorten als Prototypen entwickelt und sollen zusammen mit dem Buchungsmodul der Geschäftsleitung vorgestellt werden. Es sollen mehrere Benutzer aus verschiedenen Abteilungen gleichzeitig damit arbeiten können. Da die Firma Telearbeit einführen will, soll das System auch von außen (z.B. über das Internet) ansprechbar sein. Um in kritischen Phasen (z.B. zum Quartals- oder Jahresabschluss) keine Engpässe zu erzeugen, muss das System in hohem Maße verfügbar sein. Ferner soll der wachsende Personalstamm berücksichtigt werden.

Abbildung 1.1: Konsequenzen eines Businessproblems

Die Konsequenzen

Wenn man den vorangegangenen Text oberflächlich liest, so könnte man zu dem Schluss kommen, dass die gestellte Anforderung dem Anwendungsentwickler, der ja ein Experte für betriebswirtschaftliche Applikationen ist, keine schlaflosen Nächte bereiten dürfte. Liest man den Text etwas genauer, stellt sich eine andere Situation dar (wie Abbildung 1.1 zeigt). Die Krux liegt für unseren Anwendungsspezialisten in den geänderten Rahmenbedingungen. Diese bringen Probleme mit sich, die ihm beim Entwurf eines Prototyps für ein rudimentäres Buchungssystem im Wege stehen. Lassen Sie uns diese Probleme in den nun folgenden Ausführungen näher erläutern.

Mehrbenutzerfähigkeit

Das Buchungssystem, das bislang als Einzelanwendung in Form einer Insellösung konzipiert war, soll nun in das Gesamtsystem integriert werden und mehrere Benutzer in die Lage versetzen, gleichzeitig damit arbeiten zu können. Dazu muss das System

- Mehrprozessfähigkeit (Multithreading),
- Transaktionen,
- Benutzerverwaltung,
- Sicherheitsmechanismen und
- Netzwerkfähigkeit

unterstützen.

Skalierbarkeit

Um den wachsenden Personalstamm in angemessener Weise berücksichtigen zu können, muss dafür gesorgt werden, dass das System skalierbar ist. Wenn die bisherigen Ressourcen mit der Anzahl der Benutzer überfordert sind, muss es möglich sein, die Software auf zusätzlich geschaffene Ressourcen auszudehnen. Die Konsequenz daraus ist die Verteilung der Anwendungen auf mehrere Computeranlagen. Das System sollte dann (gegebenenfalls nach einer Neukonfiguration) die neuen Ressourcen nutzen und so eine verbesserte Leistungsfähigkeit ohne Engpässe bieten.

Verfügbarkeit

Generell, vor allem aber in kritischen Phasen wie dem Quartals- oder Jahresabschluss, muss das System eine hohe Verfügbarkeit gewährleisten. Sollte ein Rechner im Verbund ausfallen, muss ein anderer Rechner ohne größere Umstellungsarbeiten seine Funktion übernehmen können, um das System am Leben zu erhalten. Diese Anforderung verlangt dem System die Fähigkeit ab, funktionale Einheiten auf andere Computer verlagern zu können.

Verbindung mit der Außenwelt

Da die Firma mit einem einheitlichen System stärker im Verbund arbeiten wird und zudem Telearbeit einführen will, muss das System eine Schnittstelle zur Außenwelt bieten (z.B. in Form einer Internetanbindung). Das erhöht nicht nur die Anforderungen an die Sicherheit, sondern auch an die Systemarchitektur im Allgemeinen, da sich die Techniken für Internetanwendungen von denen herkömmlicher Anwendungen wesentlich unterscheiden.

Integration mit anderen Anwendungen und leichte Erweiterbarkeit

Im Idealfall stellt sich Integrationsfähigkeit so dar, dass man verschiedene Anwendungen, die nach einer Art Baukastenprinzip entwickelt worden sind, zu einem System zusammenstecken kann. Jede Anwendung sollte für sich wiederum aus Bausteinen zusammengesetzt sein und anderen Anwendungen Teilfunktionalität über fest definierte Schnittstellen zur Verfügung stellen können. Durch Hinzufügen weiterer Bausteine kann das System erweitert werden. Das setzt voraus, dass die dem System zugrunde liegende Technik eine entsprechende Feinkörnigkeit (Granularität) von Anwendungen unterstützt.

Kurze Entwicklungszyklen

Bei der Softwareentwicklung steht im Zuge von Rationalisierungen grundsätzlich eine Maßnahme im Vordergrund: die Verkürzung der Entwicklungszyklen und damit die Senkung der Entwicklungskosten. Erreicht wird dies in der Regel durch die Erhöhung der Feinkörnigkeit. Aus einem großen, unüberschaubaren Projekt macht man viele kleine, leicht beherrschbare Projekte, deren Ergebnis ein Baustein ist. Die Summe aller Bausteine ergibt am Ende das System. Somit trägt eine Art Baukastensystem nicht nur zur Verbesserung der Integrität bzw. Erweiterbarkeit, sondern tendenziell auch zur Verkürzung des Entwicklungszyklus bei.

Konfigurierbarkeit

Damit die Anwendungen, die an einem Standort entwickelt worden sind, auch an anderen Standorten eingesetzt werden können (denken Sie an unterschiedliche nationale Gesetzgebung, unterschiedliche Sprachen, nationale Sonderzeichen etc.), müssen diese in hohem Maße konfigurierbar und damit anpassbar sein, ohne eine Neukompilierung erforderlich zu machen. Als weitere Alternative könnte man sich vorstellen (um noch einmal auf den bereits mehrfach angesprochenen Baukasten zurückzukommen) bestimmte Systembausteine, die in dieser Form an einem anderen Standort nicht einsetzbar sind, durch standortoptimierte Bausteine zu ersetzen.

Schrittweise Migration und Datenablage

Da eine Systemumstellung ab einer gewissen Firmengröße nicht von heute auf morgen erfolgen kann, ist die schrittweise Umstellung auf ein neues System ein wichtiges Thema. Schnittstellen zu bestehenden Systemen sollen auf einfache Art und Weise verwirklicht werden können. Aus dieser Anforderung folgt indirekt eine wesentlich weitergehendere. Im Idealfall spielt es für die einzelne Anwendung keine Rolle, wo die Daten letzten Endes abgelegt werden. Sie können in der zum neuen System gehörenden Datenbank, in Datenbeständen von bestehenden Systemen oder gar in Datenbeständen von Fremdsystemen gespeichert werden.

Fazit

Die Lösung der oben genannten Probleme kann die Fähigkeiten eines normalen Anwendungsentwicklers unter Umständen übersteigen. Heutzutage gibt es jedoch keinen Grund mehr, vor derartigen Anforderungen zu kapitulieren. Diverse Softwarehäuser stellen Entwicklern mehrere Möglichkeiten zur Verfügung, um die Probleme bewältigen zu können. Dieses Buch wird eine dieser Möglichkeiten im Detail behandeln: Enterprise JavaBeans (EJB).

Wie geht es weiter?

In Kapitel 2 werden wir die Grundlagen von Enterprise JavaBeans behandeln und versuchen, es im Kontext der Softwareentwicklungs-Technologien einzuordnen. In Kapitel 3 widmen wir uns der grundlegenden Architektur von EJB sowie den Rollen, die ein Entwickler annehmen kann. Die unterschiedlichen Arten von Beans, nämlich Entity-, Session- und Message-Driven-Beans, erklären die Kapitel 4, 5 und 6. Darauf aufbauend werden wir das Thema Transaktionen in Kapitel 7 behandeln. Kapitel 8 widmet sich dem Aspekt der Sicherheit. Alle diese Kapitel werden den behandelten Stoff mit ausführlichen Beispielen veranschaulichen. Kapitel 9 wird abschließend einige Aspekte der praktischen Anwendung (ebenfalls mit ausführlichen Beispielen) behandeln.

Nachdem Sie dieses Buch gelesen haben, sollten Sie beurteilen können, ob Enterprise JavaBeans der richtige Ansatz für die Lösung Ihrer Probleme ist. Ferner sollten Sie in der Lage sein, das im Buch vermittelte Wissen in die Praxis umzusetzen.

2 Grundlagen

Enterprise JavaBeans ist eine Architektur (Framework) für komponentenorientierte, verteilte Anwendungen. *Enterprise-Beans* sind Komponenten verteilter, transaktionsorientierter Geschäftsanwendungen.
[Sun Microsystems, 2001]

2.1 Enterprise

Die Programmiersprache Java wurde im Wesentlichen durch zwei Dinge bekannt: durch ihre Plattformunabhängigkeit und durch ihre Fähigkeit, Webseiten in Form von Applets zu verschönern. Nun hat die Programmiersprache Java aber weitaus mehr Fähigkeiten zu bieten, als einfach nur Webseiten zu verzieren. Java ist eine vollständig objektorientierte Programmiersprache, die zunehmend auch für die Entwicklung unternehmenskritischer Anwendungen eingesetzt wird. Wie Abbildung 2.1 zeigt, ist Enterprise JavaBeans ein Baustein im Produkt- und Schnittstellenkatalog von Sun Microsystems zur Entwicklung von Enterprise-Anwendungen.

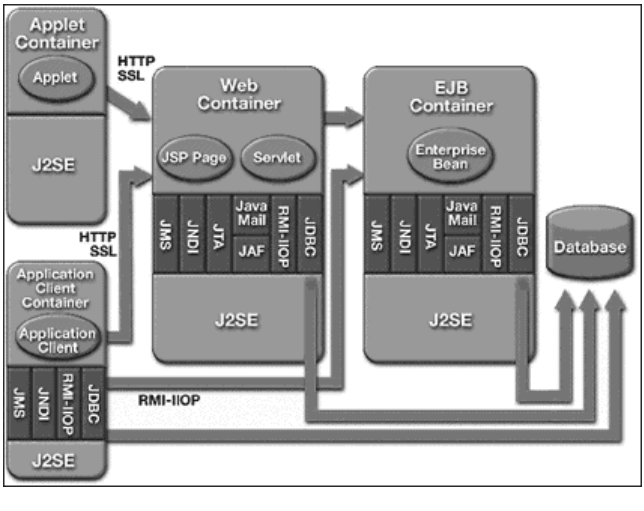

Abbildung 2.1: Einordnung von EJB in Suns Enterprise-Konzept [J2EE-APM, 2000]

Abkürzung	Bedeutung
J2EE	Java-2-Plattform, Enterprise Edition
JSP	Java Server Pages
EJB	Enterprise JavaBeans

Tabelle 2.1: Legende zu Suns Enterprise-Konzept

Enterprise JavaBeans ist kein Produkt, sondern lediglich eine Spezifikation. Jeder, der sich dazu berufen fühlt, kann eine Implementierung der Enterprise-JavaBeans-Spezifikation auf den Markt bringen.

Dass es sich bei Enterprise JavaBeans um eine Client-Server-Architektur handelt, dürfte nicht weiter für Verwunderung sorgen (SUN positioniert Enterprise JavaBeans im Bereich serverseitiger Anwendungslogik). Die so genannte Three-Tier-Architecture (Dreischichten-Architektur) ist derzeit wohl diejenige, für die sich die meisten Systemarchitekten entscheiden (der Trend geht in Richtung so genannter Multi-Tier- oder Mehrschicht-Architekturen).

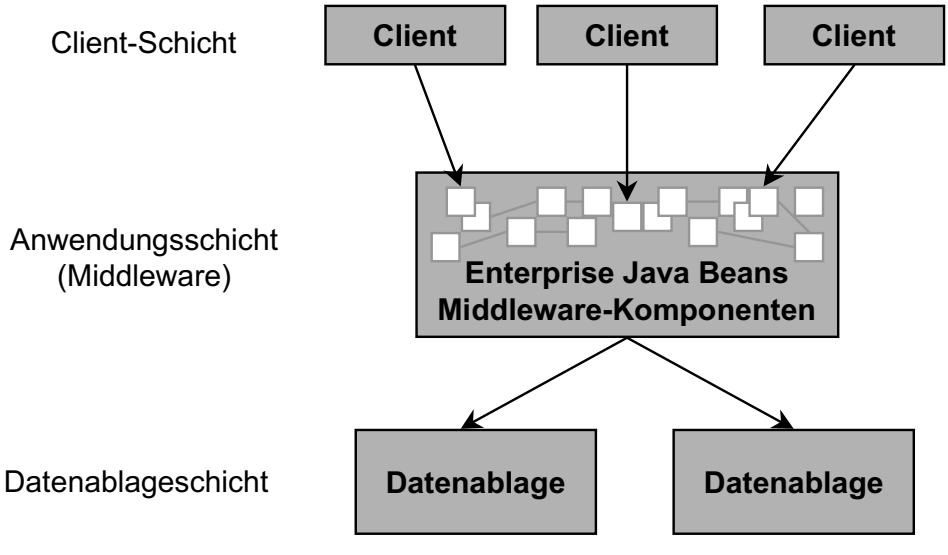

Abbildung 2.2: Dreischichten-Architektur

Dreischichtige Systeme zeichnen sich dadurch aus, dass die eigentliche Programmlogik (im Zusammenhang mit Unternehmensanwendungen auch oft Business-Logik genannt) in der Mittelschicht (auf einem Applikationsserver) liegt (vgl. Abbildung 2.2). Die Bündelung der Anwendungslogik auf einem eigenen Server in einer Schicht zwi-

schen den Clients und der Datenablage bietet gegenüber der herkömmlichen Two-Tier-Architecture (Zweischichten-Architektur) folgende Vorteile:

- Die Clientprogramme werden in ihrem Umfang wesentlich kleiner (daher oft Thin-Clients genannt) und beanspruchen so auf dem Clientrechner weniger Ressourcen.
- Die Datenablage wird für den Cliententwickler transparent. Die Mittelschicht (Anwendungsschicht) abstrahiert vollständig vom Zugriff auf die Daten und kümmert sich um die Datenkonsistenz (was die Entwicklung von Clientprogrammen wesentlich vereinfacht).
- Die Anwendungen werden besser skalierbar, da bereits auf Applikationsserverebene eine Lastverteilung stattfinden kann (z. B. kann im Fall einer Überlastung des Servers die Anfrage des Clients an einen anderen Server weitergegeben werden).
- Eine bestimmte Anwendungslogik wird nur einmal programmiert und zentral in der Mittelschicht allen Clients zur Verfügung gestellt, wodurch das System als Ganzes leichter zu warten und zu erweitern ist. Bei einer Änderung in der Applikationslogik ist nur eine zentrale Komponente in der Mittelschicht betroffen und nicht eine Vielzahl von Clientanwendungen.

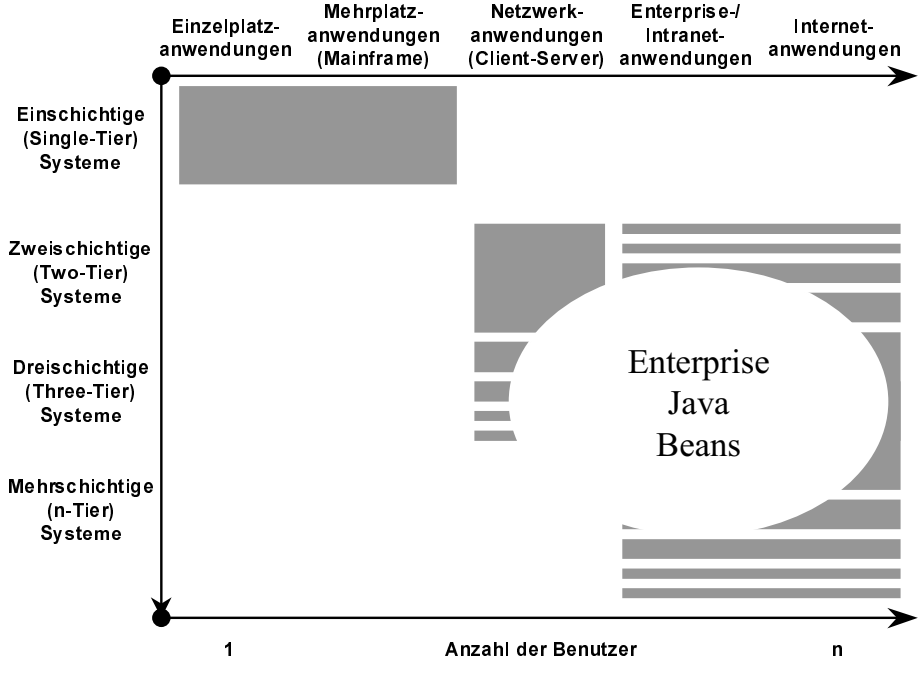

Abbildung 2.3: Einordnung von EJB in einem Systemportfolio

Versucht man bestehende Systemarchitekturen zu klassifizieren, um Enterprise JavaBeans zuordnen zu können, könnte sich ein Bild ergeben, wie es Abbildung 2.3 darstellt.

EJB findet seinen Platz vor allem im Bereich von Unternehmens-, Intranet- und Internet-Anwendungen, die hauptsächlich auf einer dreischichtigen und bisweilen auch n-schichtigen Architektur aufbauen und in einer objektorientierten Sprache (in diesem Fall Java) entwickelt sind. Von einer mehrschichtigen Architektur (n-Tier oder Multi-Tier genannt) spricht man dann, wenn neben der Client-, Applikations- und Datenablageschicht noch zusätzliche Schichten zur Erhöhung des Abstraktionsgrades oder zur Verbesserung der Skalierbarkeit eingezogen oder wenn mehrere dreischichtige Systeme kaskadiert werden.

Da das Wort Enterprise (Unternehmen) ausdrücklich in der Bezeichnung auftaucht, drängt sich die Frage auf, was die Enterprise-JavaBeans-Spezifikation leistet, um Entwicklern zu helfen, den heutigen Unternehmensanforderungen gerecht zu werden. Um diese Frage beantworten zu können, sollte man sich zunächst überlegen, welche Kriterien bei der Entscheidung für oder gegen eine Technologie zugrunde gelegt werden.

Wirtschaftlichkeit

Die Wirtschaftlichkeit bezieht sich auf die Bereiche Entwicklung bzw. Anschaffung, Wartung, Weiterentwicklung und Anpassung. Wirtschaftlichkeit in der Entwicklung entsteht dann, wenn die Technologie die Verkürzung des Entwicklungszyklus aktiv unterstützt. Somit können durch eine Steigerung der Produktivität eingesparte Kosten im Preis weitergegeben werden (was für den Nicht-Entwickler Wirtschaftlichkeit in der Anschaffung bedeutet). Wirtschaftlichkeit in der Wartung entsteht durch Stabilität, die weniger Wartung erforderlich macht. Wirtschaftlichkeit bezüglich Weiterentwicklung und Anpassung bietet eine Technologie, die es ermöglicht, ein System ohne größeren Aufwand und Risiken erweitern bzw. anpassen zu können (ein großer Aufwand und eintretende Risiken sind immer mit Kosten verbunden).

Sicherheit

Der Aspekt der Sicherheit muss ebenfalls sehr differenziert betrachtet werden. Sicherheit in Bezug auf Investitionssicherheit ist für ein Unternehmen dann gegeben, wenn die Technologie für das Unternehmen eine gewisse Langlebigkeit verspricht und eine langfristige Unterstützung sichergestellt ist. Sicherheit meint auch die Ausfallsicherheit, also die Verfügbarkeit. Es gilt, Kosten zu vermeiden, die durch Systemfehlfunktionen oder Systemausfälle verursacht werden. Sicherheit bedeutet auch die Fähigkeit einer Technologie, sensible Daten vor neugierigen Augen bzw. das System vor unliebsamen Eindringlingen schützen zu können.

Bedarfsorientierung

Der Einsatz einer bestimmten Technologie rechtfertigt sich schließlich erst dann, wenn auch tatsächlich ein Bedarf nach deren Eigenschaften gegeben ist. Ein Unternehmen wird eine Technologie oder ein Produkt nicht allein wegen seiner momentanen Popularität einsetzen, ohne einen wirklichen Bedarf an den eigentlichen Fähigkeiten zu haben.

Dieses Buch wird in den folgenden Kapiteln noch im Detail darlegen, wie die Enterprise-JavaBeans-Spezifikation versucht, den Anforderungen moderner Unternehmen an ein stabiles System gerecht zu werden.

2.2 Java

Durch Enterprise JavaBeans sollte die Programmiersprache Java den Sprung von der Client- auf die Serverplattform schaffen. Natürlich war es auch ohne Enterprise JavaBeans möglich, einen Server in Java zu entwickeln. Java bietet hierfür in seinen Standardbibliotheken eine umfangreiche Unterstützung für Netzwerkkommunikation sowie eine Unterstützung für Threads. Durch EJB wird es auch für »normale« Programmierer, die keine Serverspezialisten sind, möglich, einen Anwendungsserver um Funktionalität zu erweitern bzw. serverseitige Logik zu entwickeln.

Welchen Beitrag kann Java leisten, um EJB zu einem erfolgreichen Konzept für die Entwicklung unternehmenskritischer Anwendungen zu machen?

Objektorientierung

Java ist eine objektorientierte Programmiersprache und bietet dem Entwickler somit die Vorzüge objektorientierter Techniken. Damit ist Java auch gut für den nächsten Schritt gerüstet, nämlich für den Schritt in Richtung komponentenorientierter Softwareentwicklung, für die objektorientierte Technologien wohl die beste Ausgangsbasis sind (dazu mehr im Abschnitt 2.3).

Plattformunabhängigkeit

Viele Unternehmen sehen sich heute mit dem Problem einer heterogenen Hard- und Softwarelandschaft konfrontiert, was zu teilweise unüberwindlichen Schwierigkeiten führen kann. Viele Softwaresysteme sind eben nur für bestimmte Plattformen verfügbar. Der Java-Quellcode wird vom Java-Compiler nicht in Maschinencode, sondern in maschinenneutralen Byte-Code übersetzt, der dann von einer Java-Laufzeitumgebung interpretiert wird. Daher sind Applikationen, die in »reinem« Java entwickelt wurden, auf allen Plattformen lauffähig, für die eine Java-Laufzeitumgebung verfügbar ist. Die Laufzeitumgebung ist plattformabhängig und stellt dem Java-Programm eine Imple-

mentierung abstrakter Schnittstellen für systemabhängige Operationen zur Verfügung (z.B. für den Zugriff auf Dateisysteme, Zugriff auf Netzwerkschnittstellen, grafische Oberflächen etc.). Java-Anwendungen bieten durch ihre Plattformunabhängigkeit für ein Unternehmen mehr Sicherheit und Kontinuität.

Dynamik

Java bringt als Interpretersprache die Fähigkeit mit, Byte-Code entweder vom lokalen Dateisystem oder über das Netz in den Adressraum eines Prozesses zu laden und aus diesem zur Laufzeit Objekte zu erzeugen (d.h., Programmteile können zur Laufzeit – auch über das Netzwerk – nachgeladen werden). Damit ebnet sich der Weg von starren, unflexiblen Systemen hin zu laufzeitdynamischen Systemen mit hoher Anpassungsfähigkeit. Auch damit wird ein Beitrag zu mehr Flexibilität und Kontinuität geleistet. Java-Klassen können zur Laufzeit untersucht werden (Reflection). So können Methoden dynamisch aufgerufen, Attribute erkannt und modifiziert werden etc. Durch die zunehmende Unterstützung von Java in Datenbanksystemen (z.B. Oracle), Groupwaresystemen (z.B. Lotus Notes), Webservern (durch die Servlet-API und Java-ServerPages) und Browsern bieten sich umfangreiche Kombinationsmöglichkeiten beim Systementwurf. Auch die Verbindung mit anderen plattformunabhängigen Technologien wie z.B. XML bietet interessante Perspektiven. So löste XML in der Version 1.1 der EJB-Spezifikation serialisierte Klassen als Deployment-Deskriptor ab (mehr Details zum Deployment-Descriptor in Kapitel 3).

Stabilität

Die Programmiersprache Java ist relativ leicht zu erlernen (verglichen beispielsweise mit C++) und insofern weniger fehlerträchtig, da bestimmte sprachliche Mittel zugunsten einer konsequenten Objektorientierung nicht existieren (z.B. Zeiger auf Variablen und Funktionen). Durch die Plattformunabhängigkeit sind, sofern das System in reinem Java entwickelt wurde, keine Portierungen notwendig. Dadurch können auch keine Portierungsfehler entstehen. Da Java eine Interpretersprache ist, führen schwere Laufzeitfehler (z.B. Zugriff auf ungültige Objektreferenzen, Null-Pointer) nicht zu unkontrollierten Systemabstürzen. Das Programm wird von der Laufzeitumgebung kontrolliert beendet. Kritische Fehler können somit schneller gefunden und beseitigt werden.

Sicherheit

Java unterstützt den Aspekt der Sicherheit durch Spracheigenschaften. So ist der direkte Zugriff auf Speicherbereiche über Zeiger nicht möglich, Stack-Überläufe werden ebenso wie das Überschreiten von Array-Grenzen abgefangen und in Form einer Ausnahme (Exception) als Fehler angezeigt etc. Java unterstützt auch das so genannte Sandbox-Konzept, das vor allem bei Applets Anwendung findet.

Andererseits wird Sicherheit bei Java durch Programmierschnittstellen und Implementierungen unterstützt, die zum Standardumfang der Laufzeit- und Entwicklungsumgebung gehören (momentan die Version 1.3). Durch eine an die individuellen Bedürfnisse angepasste Implementierung eines Java-SecurityManagers z.B. können kritische Operationen von Objekten (z.B. das Lesen und Schreiben von Dateien, das Öffnen von Netzwerkverbindungen etc.) überwacht und gegebenenfalls verboten werden. Java bietet eine Verwaltung für private und öffentliche Schlüssel sowie eine Programmierschnittstelle für die Verschlüsselung. Es besteht auch die Möglichkeit, Java-Archivdateien (jar-Dateien) zu signieren um die Manipulation des Byte-Codes durch Dritte zu verhindern. Eine sehr gute und vollständige Behandlung des Aspekts der Sicherheit im Zusammenhang mit der Programmiersprache Java bietet [Oaks, 1998].

Performanz

Die Vorzüge, die Java aus den Eigenschaften einer Interpretersprache gewinnt, muss es auf der anderen Seite mit Problemen bei der Performanz bezahlen. Obwohl bereits viel unternommen wird, um die Java-Performanz zu verbessern (z.B. durch Just-in-time-Compiler), ist die Ausführungsgeschwindigkeit immer noch ein kritischer Punkt (bei komplexen Clientanwendungen sowie bei Serveranwendungen). Durch die stetige Verbesserung der Hardware und der Java-Laufzeitumgebungen wird dieses Problem vermutlich zunehmend relativiert. Bis Java jedoch die Ausführungsgeschwindigkeit einer Compilersprache erreicht, wird noch Entwicklungsarbeit im Bereich der virtuellen Maschine notwendig sein.

2.3 Beans

Von Beans ist immer dann die Rede, wenn es bei der Firma Sun um Komponenten geht. Die wohl bislang populärsten Komponenten von Java sind die JavaBeans. Bevor wir uns mit dem Unterschied zwischen JavaBeans und Enterprise JavaBeans auseinandersetzen, wollen wir kurz in die Welt der Componentware eintauchen, ohne dabei den Anspruch erheben zu wollen, eine vollständige Diskussion des Komponentenparadigmas anzubieten. Die Diskussion über Komponenten füllt zahlreiche Bücher, die zeigen, wie differenziert dieses Thema diskutiert werden kann. An dieser Stelle soll lediglich ein Einblick vermittelt werden, um das Verständnis für die Komponententhematik (die bei Enterprise JavaBeans eine wichtige Rolle spielt) zu verbessern.

Um Missverständnisse zu vermeiden, soll bereits an dieser Stelle ausdrücklich erwähnt werden, dass die Intention dieses Buches nicht darin besteht, dem Leser eine Anleitung zu geben, wie gute Komponenten zu entwickeln sind. Dieses Buch beschreibt die Komponentenarchitektur Enterprise JavaBeans, deren praktische Anwendung und die Voraussetzungen, die Komponenten erfüllen müssen, um in einem EJB-Server eingesetzt werden zu können.

Softwarekomponenten

Wir wollen uns an eine Definition halten, die in [Griffel, 1998] sowie in [Orfali et al., 1996] zitiert wird und einen relativ neutralen Mittelweg zwischen ausschweifenden und sehr knappen Definitionen darstellt:

Eine Komponente ist ein Stück Software, welches klein genug ist, um es in einem Stück erzeugen und pflegen zu können, groß genug ist, um eine sinnvoll einsetzbare Funktionalität zu bieten und eine individuelle Unterstützung zu rechtfertigen, sowie mit standardisierten Schnittstellen ausgestattet ist, um mit anderen Komponenten zusammenzuarbeiten.

Zunächst wird die Sicht auf eine Komponente erleichtert, indem man sie sich als eine Art Lego-Baustein vorstellt. Das Innere des Bausteins bleibt verborgen. Man sieht jedoch, dass er durch seine Verbinder mit anderen Bausteinen zusammengesteckt werden kann. Die Kombination passender Bausteine ergibt ein Gebilde mit einem bestimmten Zweck (Häuser, Garagen, Straßen etc.). Softwarekomponenten sind ebenfalls Bausteine, in die man nicht hineinsehen kann (Abbildung 2.4 veranschaulicht dies beispielhaft). Sie bieten eine in sich geschlossene Teilfunktionalität (in Analogie zum Lego-Baustein etwa den Rahmen für ein Fenster, einen Stein als Teil einer Wand oder eine Platte für den Grundstock eines Hauses), verbergen dabei aber ihre Implementierung. Ihre Funktionalität ist nur anhand der öffentlichen Schnittstelle ablesbar, die neben der reinen Anwendbarkeit auch die Kopplung mit anderen Komponenten ermöglicht. Wie bei Lego-Bausteinen liegt auch bei Softwarekomponenten der Schwerpunkt auf der Wiederverwendbarkeit. Eine Komponente, die in nur einem Anwendungsszenario einsetzbar ist, ist keine echte Komponente.

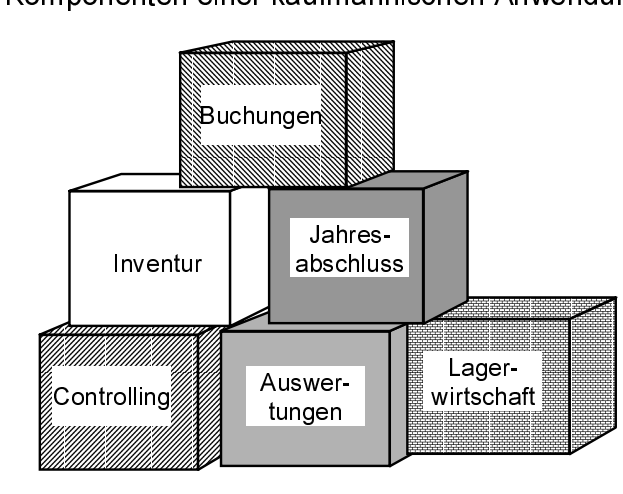

Abbildung 2.4: Beispiel für Komponenten einer kaufmännischen Anwendung

Eine Komponente unterscheidet sich von herkömmlichen Objekten bzw. Klassen in vielerlei Hinsicht (vgl. auch [Griffel, 1998]) :

- Komponenten sollen in verschiedenen Anwendungszenarien einsetzbar und damit mehrfach wiederverwendbar sein. Herkömmliche Objekte dagegen werden in der Regel für ein bestimmtes Szenario entwickelt. Zudem bedienen sie sich zur Erfüllung ihrer Aufgaben häufig anderer Objekte und erschweren durch diese Verstrickung die Wiederverwendbarkeit.
- Objekte sind in der Regel nicht groß genug, um eine in sich geschlossene Aufgabenstellung abzudecken. Sie dienen zur Strukturierung und zur Abbildung auf Modelle.
- Der Entwicklungsaufwand einer Komponente ist wesentlich höher als der eines herkömmlichen Objekts, da bei der Komponente verstärkt auf die Wiederverwendbarkeit in unterschiedlichen Anwendungen und auf die Integrationsfähigkeit geachtet werden muss.
- Eine Komponente kann lediglich in einem bestimmten Rahmen, der über die öffentlichen Schnittstellen vorgegeben wird, angepasst werden. Objekte können durch Ableitung (fast) beliebig verändert werden.
- Objekte bieten zwar ein Schnittstellenkonzept, das in der Regel aber eng an die darunter liegende Systemtechnik gekoppelt ist und deswegen die Interoperabilität einschränkt.

Zweifelsohne ist die enge Verwandtschaft zwischen Objekten und Komponenten deutlich zu erkennen. Objektorientierte Denkweisen und Techniken bieten daher die wohl beste Basis für die Entwicklung von Komponenten bzw. komponentenorientierter Software.

Ein wesentliches Konzept des Komponentenparadigmas ist die *Schnittstelle*. Die Schnittstelle einer Komponente ist eine Art Vertrag, den zu erfüllen sich die Komponente verpflichtet. Sie ist der Interaktionspunkt mit der Komponente, sie dokumentiert deren Fähigkeiten und steht stellvertretend für deren Abstraktionscharakter. Eine Komponente kann mehrere Schnittstellen haben. Jede Schnittstelle repräsentiert einen durch die Komponente zur Verfügung gestellten Dienst (für eine detaillierte Diskussion des Schnittstellenaspekts vergleiche [Szyperski, 1998]). Kapitel 3 wird im Detail zeigen, wie dieser Aspekt bei Enterprise JavaBeans zur Umsetzung kommt.

Ein Vorteil der komponentenorientierten Softwareentwicklung liegt (wie bereits erwähnt) in der Wiederverwendbarkeit von Code. Ein weiterer Vorteil ist die Möglichkeit, mit Hilfe von fertigen Komponenten in kurzer Zeit Anwendungsprototypen erstellen zu können (für eine ausführliche Abhandlung dazu siehe [IBM, 1997]). Durch die frühe Verfügbarkeit von Prototypen können bereits im Anfangsstadium der Entwicklung Erkenntnisse über Designentscheidungen getroffen werden; (Pilot-)Kunden oder

(Pilot-)Anwender können früher in den Entwicklungsprozess einbezogen werden etc. Durch die Wiederverwendung von Code in Form von (ausgereiften) Softwarekomponenten können Entwicklungszyklen verkürzt und Entwicklungskosten gespart werden.

Die Komponentenarchitektur

Wie oben erwähnt ist Enterprise JavaBeans eine Komponentenarchitektur. Die Einsatzgebiete und Ausprägungen einer Komponentenarchitektur können sehr vielseitig sein. EJB repräsentiert dabei eine ganz bestimmte Variante: eine Komponentenarchitektur für verteilte, serverseitige und transaktionsorientierte Komponenten. Enterprise-Beans sind also Komponenten, die vielen Clients auf einem Server ihre Dienste zur Verfügung stellen. Ohne eine Rahmenarchitektur, die die Komponenten in eine Art von Laufzeitumgebung einbettet und ihnen die notwendigen Dienste zur Verfügung stellt, müsste jede Komponente, die über ein Netzwerk verfügbar sein soll, ein eigener Server sein. Das würde die Entwicklung einer derartigen Komponente wesentlich erschweren und beim Einsatz mehrerer Komponenten auf einem Rechner für eine unnötige Belastung der Ressourcen sorgen. Auch die Wiederverwendbarkeit einer Komponente könnte so ins Wanken geraten, da Server häufig auf die darunter liegende Plattform abgestimmt werden müssen. Eine Komponentenarchitektur wie die Enterprise JavaBeans ermöglicht den Einsatz von Komponenten für verteilte Anwendungen, ohne dass die Komponente selbst wesentlich davon betroffen ist (vgl. Abbildung 2.5).

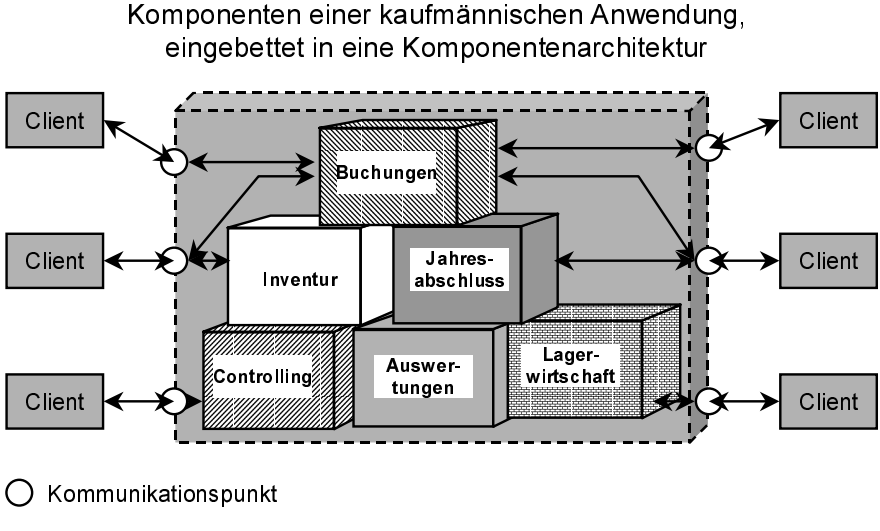

Abbildung 2.5: Beispiel für eine Komponentenarchitektur

[Griffel, 1998] zeigt eine Auflistung von Anforderungen, die eine Komponentenarchitektur erfüllen sollte (zitiert aus [I-Kinetics, 1997]).

- *Unabhängigkeit von der Umgebung:* Komponenten sollten ohne Rücksicht auf Programmiersprachen, Betriebssysteme, Netztechnologien etc. eingesetzt werden können bzw. mit diesen zusammenarbeiten.
- *Ortstransparenz:* Für den Benutzer von Komponenten sollte es keine Rolle spielen, ob die Komponente in seinem lokalen Adressraum oder im Adressraum eines anderen, entfernten Computers ihre Dienste anbietet. Entsprechende Mechanismen für die transparente Nutzung von lokalen oder entfernten Komponenten sollten durch die Komponentenarchitektur zur Verfügung gestellt werden.
- *Trennung von Schnittstelle und Implementierung:* Die Spezifikation einer Komponente sollte vollständig unabhängig von ihrer Implementierung erfolgen können.
- *Selbstbeschreibende Schnittstellen:* Um eine lose Kopplung von Komponenten zur Laufzeit erreichen zu können, sollte eine Komponente in der Lage sein, über ihre Fähigkeiten und Einstiegspunkte Auskunft zu geben.
- *Problemlose sofortige Nutzbarkeit (Plug&Play):* Eine Komponente sollte ohne Umschweife auf jeder Plattform nutzbar sein (was eine binäre Unabhängigkeit des Komponentencodes impliziert).
- *Integrations- und Kompositionsfähigkeit:* Eine Komponente sollte in der Kombination mit anderen Komponenten zur Schaffung einer neuen, gültigen Komponente beitragen können.

Enterprise JavaBeans ist aber nicht nur eine Komponentenarchitektur. Die Spezifikation definiert ein systemtechnisch orientiertes *Komponentenmodell* (für den Begriff des Komponentenmodells vergleiche ebenfalls [Griffel, 1998]). Es erlaubt den Einsatz verschiedener Typen von Enterprise-Beans. Es definiert Protokolle für die Verwaltung der Komponenten, für die Kooperation und Kommunikation der Komponenten untereinander und für die Benutzung durch den Client.

JavaBeans vs. Enterprise JavaBeans

Eine *JavaBean* ist eine wiederverwendbare Software-Komponente, die mit einem Builder-Tool visuell manipuliert werden kann. [Sun Microsystems, 1997]

JavaBeans ist ein Komponentenmodell der Firma Sun für Java. Eine JavaBean ist im Wesentlichen eine Java-Klasse, die den in der JavaBeans-Spezifikation beschriebenen Regeln folgt. Die wichtigsten Merkmale einer Bean sind ihre öffentliche Schnittstelle, die Möglichkeit, auf ihre Beschaffenheit hin analysiert zu werden, die Anpassbarkeit an individuelle Bedürfnisse und die Fähigkeit zur Persistenz (via Objektserialisierung). Die öffentliche Schnittstelle besteht aus den Eigenschaften (Properties) einer Bean, den Methoden, die sie anderen zu benutzen erlaubt, und den Events, die sie auslöst oder empfängt. Eine Bean kann eine sichtbare (z. B. ein Button) oder eine unsichtbare Komponente (z. B. ein Netzwerkdienst) sein. Das folgendes Beispiel zeigt eine gültige JavaBean:

```
public class AValidBean implements AEventListener {

    private int aProperty;
    private Vector beanListeners;

    public AValidBean()
    {
        aProperty     = -1;
        beanListeners = new Vector();
    }

    public void setAProperty(int value)
    {
        aProperty = value;
    }

    public int getAProperty()
    {
        return aProperty;
    }

    public void addBEventListener(BEventListener listener)
    {
       beanListeners.addElement(listener);
    }

    public void removeBEventListener(BEventListener listener)
    {
        beanListener.remove(listener);
    }

    private void fireBEvent() {
        BEventListener l;
        for(int i = 0; I < beanListener.size(); i++) {
            l = (BEventListener)beanListener.elementAt(i);
            l.notify(new BEvent(this));
        }
    }

    //Implementation of AEventListener-Interface

    public void notify(AEvent event)
    {
        //processing the event
    }
}
```

Listing 2.1: Beispiel einer JavaBean

Diese Bean-Klasse ist von keiner Klasse abgeleitet, sie implementiert keine Standardschnittstellen und ist trotzdem eine gültige JavaBean (nur sichtbare JavaBeans haben die Auflage, dass sie von `java.awt.Component` abgeleitet sein müssen). Sie folgt lediglich den in der Spezifikation festgelegten Namenskoventionen. Sie hat die Eigenschaft *aProperty*, die über die Methoden *setAProperty* und *getAProperty* manipuliert bzw. ausgelesen werden kann. Sie kann dadurch, dass sie das Interface *AEventListener* implementiert, auf den Event *AEvent* reagieren. Sie löst den Event *BEvent* aus, für den sich andere Beans als Interessenten über die Methoden *addBEventListener* registrieren und über *removeBEventListener* deregistrieren können. Über den Austausch von Events lassen sich Beans dynamisch miteinander koppeln, da sie sich zur Laufzeit für bestimmte Events registrieren und deregistrieren können. Die Kopplung über Events ist auch eine lose Kopplung, da durch die entsprechenden Listener-Interfaces vom eigentlichen Typ der Bean abstrahiert wird.

Durch die Namenskonvention *<Typ> get<Eigenschaft>(), void set<Eigenschaft>(<Typ>), implements <Eventtyp>Listener, void add<EventTyp>Listener() und void remove<EventTypeListener>()* etc. kann beispielsweise ein Builder-Tool die Bean mit Hilfe der Java-Reflection-API in Bezug auf ihre Eigenschaften und die Möglichkeit, sie an Events zu binden, analysieren (Introspection). Das Tool kann den Anwender so in die Lage versetzen, die Bean visuell zu manipulieren. Die JavaBeans-Spezifikation konzentriert sich also im Wesentlichen auf die Beschreibung der Programmierschnittstellen für

- das Erkennen und Benutzen von Eigenschaften der JavaBeans,
- die Anpassung von JavaBeans an individuelle Gegebenheiten,
- das Registrieren für und Senden von Events zwischen einzelnen JavaBeans und
- die Persistenz von JavaBeans-Komponenten.

Die Spezifikation der Enterprise JavaBeans legt dagegen den Schwerpunkt auf den Aspekt der Verteilung und auf transaktionsorientierte Geschäftsvorfälle. JavaBeans-Objekte haben keinen verteilten Charakter. Die EJB-Spezifikation beschreibt ein Service-Framework für serverseitige Komponenten. Enterprise-Beans sind niemals sichtbare Serverkomponenten. Nach der Diskussion von Eigenschaften und Events einer Enterprise-Bean sucht man in der EJB-Spezifikation vergebens, da sie hauptsächlich die Programmierschnittstellen und die Eigenschaften des Frameworks beschreibt. Natürlich können Server auf Basis herkömmlicher JavaBeans entwickelt werden. Dann müsste allerdings das Framework, das den Komponenten entsprechende Serverdienste anbietet und für die Verteilung sorgt, selbst entwickelt werden. Denkbar wäre jedoch eine Kombination aus unsichtbaren JavaBeans und Enterprise-Beans, bei der eine Enterprise-Bean eine bestimmte Schnittstelle im EJB-Server anbietet und die Aufrufe an JavaBeans delegiert (z.B über das Auslösen von Java-Bean-Events).

Man sollte nicht versuchen, allzu viele Gemeinsamkeiten zwischen beiden Modellen zu finden, da trotz der augenscheinlichen Ähnlichkeit in der Namensgebung die Schwerpunkte bei beiden Modellen stark unterschiedlich gesetzt sind. JavaBeans und Enterprise JavaBeans sind allerdings nicht als konträre, sondern eher als komplementäre Konzepte zu betrachten.

Die in diesem Kapitel unter den Schlagworten *Enterprise*, *Java* und *Beans* diskutierten Gesichtspunkte werden im weiteren Verlauf dieses Buches konkretisiert. Wir werden in Kapitel 3, »Architektur der Enterprise JavaBeans«, untersuchen, inwieweit die EJB-Spezifikation den in diesem Kapitel diskutierten Gesichtspunkten Rechnung trägt.

3 Die Architektur der Enterprise JavaBeans

3.1 Überblick

In Kapitel 2 wurde es bereits angedeutet: Enterprise JavaBeans (EJB) ist ein Bestandteil der Java-2-Plattform, Enterprise Edition (zu Details siehe [J2EE, 1999]). In diesem Modell übernimmt EJB den Teil der serverseitigen Anwendungslogik, die in Form von Komponenten, den Enterprise-Beans, zur Verfügung steht. Dieses Kapitel stellt die Architektur der Enterprise JavaBeans vor.

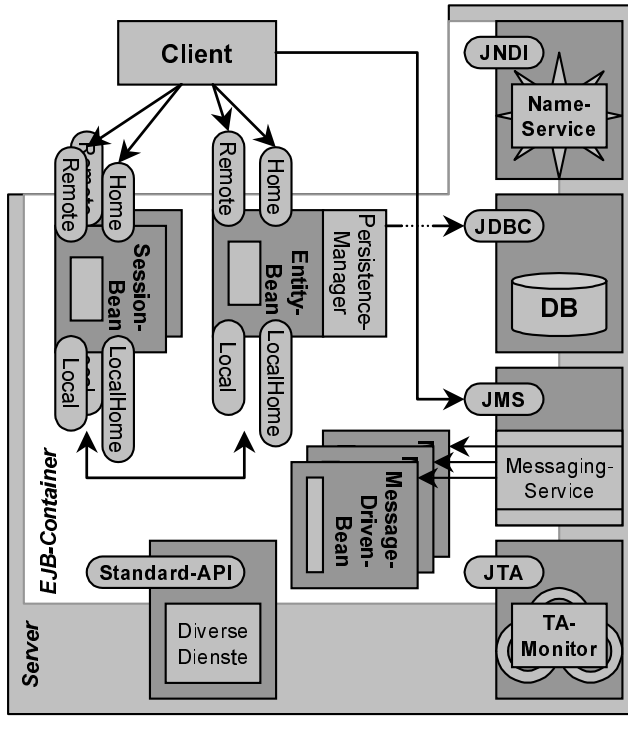

Abbildung 3.1: Gesamtüberblick über die EJB-Architektur

Abbildung 3.1 zeigt *Enterprise-Beans* (in ihren Ausprägungen als Entity-, Message-Driven- und Session-Beans) als zentrale Elemente. Sie enthalten die Anwendungslogik, die von den *Client*-Programmen genutzt wird. Die Enterprise-Beans existieren in einem *EJB-Container*, der ihnen eine Laufzeitumgebung zur Verfügung stellt (z.B. die Ansprechbarkeit durch Client-Programme über das *Home*- und *Remote*-Interface, die Möglichkeit zur Kommunikation untereinander über das *Local-Home*- und das *Local*-Interface, Lebenszyklusmanagement etc.). In den EJB-Container sind über Standard-Programmierschnittstellen *Dienste* eingebunden, die der Bean zur Verfügung stehen (z.B. der Zugriff auf Datenbanken über JDBC, der Zugriff auf einen Transaktionsdienst über JTA und der Zugriff auf einen Messaging-Service über JMS). Der EJB-Container ist (unter Umständen neben anderen Containern) in einem *Server* installiert.

Im Folgenden erläutern wir die Details der einzelnen Architekturbestandteile und weitere Zusammenhänge.

3.2 Der Server

Der Server ist die Basiskomponente der EJB-Architektur. An dieser Stelle wird absichtlich nicht von einem *EJB-Server* gesprochen. Eigentlich müsste es *J2EE-Server* heißen. Die Strategie von SUN in Hinblick auf Enterprise-Anwendungen im Rahmen der J2EE-Plattform bezieht Enterprise JavaBeans wesentlich stärker in das gesamte Portfolio von Java-basierten Programmierschnittstellen und Produkten ein, als es noch bei der Version 1.0 der Spezifikation der Enterprise JavaBeans der Fall war.

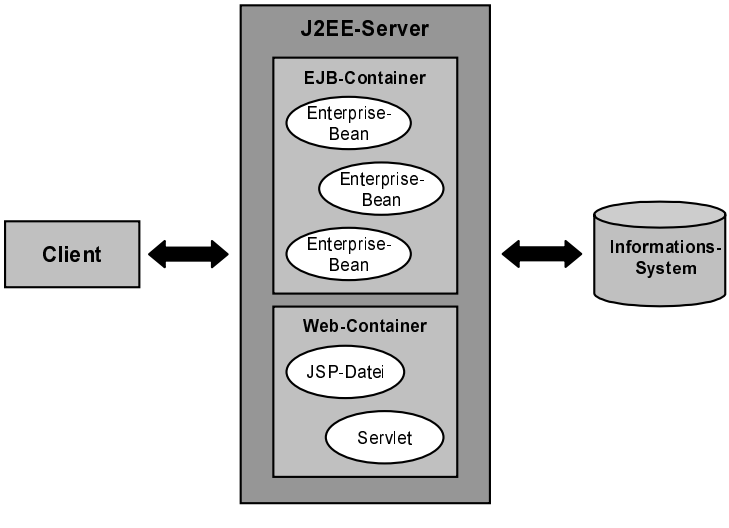

Abbildung 3.2: EJB im Kontext der Java 2, Enterprise Edition

Die Spezifikation von Enterprise JavaBeans in der Version 2.0 definiert keinerlei Anforderungen an den Server (ebenso wenig wie die vorherigen Versionen 1.0 und 1.1). Der Grund dafür ist vermutlich die stärkere Einbindung in die Java-2-Plattform, Enterprise Edition.

Ein J2EE-konformer Server ist eine Laufzeitumgebung für verschiedene Container (von denen einer oder mehrere EJB-Container sein können). Jeder Container stellt dabei wiederum eine Laufzeitumgebung für eine bestimmte Art von Komponente zur Verfügung. Die Hersteller von Java-Applikationsservern tendieren immer mehr dazu, die J2EE-Plattform zu unterstützen. Kaum ein Hersteller bietet einen reinen EJB-Server an. Viele Anbieter von Datenbanken, Transaktionsmonitoren oder CORBA-ORBs unterstützen inzwischen die Enterprise JavaBeans.

Der Serverkomponente fällt im Umfeld der J2EE-Plattform (und damit indirekt in der EJB-Architektur) die Aufgabe zu, die grundlegende Funktionalität zur Verfügung zu stellen. Dazu gehören beispielsweise:

- das Thread- und Prozessmanagement (damit mehrere Container parallel auf dem Server ihre Dienste anbieten können),
- die Unterstützung von Clustering und Lastverteilung (d.h. die Fähigkeit, mehrere Server im Verbund zu betreiben und die Anfragen der Clients je nach Auslastung zu verteilen, um bestmögliche Antwortzeiten zu gewährleisten),
- die Ausfallsicherheit,
- ein Namens- und Verzeichnisdienst (um Komponenten auffinden zu können)
- eine Zugriffsmöglichkeit auf und das Pooling von Betriebssystemressourcen (z.B. Netzwerk-Sockets für den Betrieb eines Webcontainers).

Die Schnittstelle zwischen dem Server und den Containern ist dabei sehr stark vom Hersteller abhängig. Weder die Spezifikation der Enterprise JavaBeans noch die Spezifikation der Java-2-Plattform, Enterprise Edition, definieren hierfür ein Protokoll. Die Spezifikation der Enterprise JavaBeans in Version 2.0 unterstellt, dass der Hersteller des Servers und der Hersteller des Containers identisch sind.

3.3 Der EJB-Container

Der EJB-Container ist eine Laufzeitumgebung für Enterprise-Bean-Komponenten. So wie der EJB-Container auf den Server als Laufzeitumgebung und Dienstanbieter angewiesen ist, ist eine Bean von ihrem EJB-Container abhängig. Er stellt ihr eine Laufzeitumgebung und Dienste zur Verfügung. Derartige Dienste werden der Bean über Standard-Programmierschnittstellen zur Verfügung gestellt. Die Spezifikation in der Version 2.0 verpflichtet den EJB-Container dazu, einer Bean *mindestens* folgende Programmierschnittstellen zugänglich zu machen:

- das API (Application Programming Interface) der Java-2-Plattform, Standard-Edition in der Version 1.3 ,
- das API der Spezifikation der Enterprise JavaBeans 2.0,
- das API des JNDI 1.2 (Java Naming and Directory Interface),
- das UserTransaction-API aus JTA 1.0.1 (Java Transaction API),
- das API der JDBC-2.0-Erweiterung (Java Database Connectivity),
- das API des JMS 1.0.2 (Java Messaging Service),
- das API von Java Mail 1.1 (für das Verschicken von E-Mails),
- das API von JAXP 1.0 (Java XML Parser).

Einem Anbieter eines Java-Applikationsservers steht es frei, zusätzliche Dienste über Standardschnittstellen anzubieten. Einige Hersteller bieten beispielsweise eine generische, herstellereigene Serviceschnittstelle an, über die selbstentwickelte Dienste (z.B. ein Logging-Service oder eine Benutzerverwaltung) angeboten werden können. Benutzt eine Enterprise-Bean einen derartigen proprietären Service, so ist sie nicht mehr ohne weiteres in allen Containern einsetzbar.

Der EJB-Container stellt den Enterprise-Beans zum einen eine Laufzeitumgebung zur Verfügung, zum anderen bietet er den Enterprise-Beans zur Laufzeit über die oben genannten (statischen) Programmierschnittstellen bestimmte Dienste an. Aus beiden Bereichen (dem Bereich der Laufzeitumgebung sowie dem Bereich der angebotenen Dienste) wollen wir die wichtigsten Aspekte im Detail betrachten.

Kontrolle des Lebenszyklus einer Bean (Laufzeitumgebung)

Der EJB-Container ist für die Kontrolle des Lebenszyklus einer Bean zuständig. Der EJB-Container erzeugt Bean-Instanzen (z.B. wenn ihn ein Client dazu auffordert), versetzt sie über Callback-Methoden in verschiedene Zustände und verwaltet sie in Pools für den Fall, dass sie gerade nicht benötigt werden. Die Bean-Instanzen werden vom EJB-Container auch wieder gelöscht. Die Zustände und die jeweiligen Methoden, die zum Zustandsübergang führen, hängen vom Typ der Bean ab. Sie werden in den entsprechenden Kapiteln (Kapitel 4, Session-Beans, Kapitel 5, Entity-Beans, und Kapitel 6, Message-Driven Beans) im Detail behandelt.

Instanzenpooling, Aktivierung und Passivierung (Laufzeitumgebung)

Ein System, das im Umfeld unternehmenskritischer Anwendungen positioniert ist, muss mit großen Belastungen zurechtkommen. Es muss in der Lage sein, viele Clients bedienen zu können, ohne dass diese lange Antwortzeiten in Kauf nehmen müssen. Je

größer die Anzahl der angebundenen Clients ist, desto höher ist in der Regel die Anzahl der im Server erzeugten Objekte.

Damit einerseits die Anzahl der Bean-Instanzen nicht ins Unermessliche wächst und andererseits nicht ständig Bean-Instanzen erzeugt und wieder zerstört werden müssen, hält ein EJB-Container eine gewisse Anzahl von Bean-Instanzen in einem Pool vorrätig. Solange sich eine Bean im Pool befindet, ist sie im Zustand *Pooled* und gewissermaßen deaktiviert. Bei Bedarf (in der Regel im Fall einer Clientanfrage) wird die nächstbeste Bean-Instanz des passenden Typs aus dem Pool entnommen. Sie wird reaktiviert und in den Zustand *Ready* versetzt.

Ein derartiges Instanzen-Pooling ist bei Datenbankverbindungen gängige Praxis. Ebenso hat sich dieses Vorgehen beim Management von Thread-Objekten bewährt (etwa bei der Serverentwicklung). Durch das Pooling werden in der Regel weniger Bean-Instanzen benötigt, als Clientverbindungen existieren. Durch die Vermeidung permanenter Objekterzeugung und -zerstörung erhöht sich die Performanz des Systems. Das Instanzen-Pooling steht in engem Zusammenhang mit dem Lebenszyklusmanagement einer Bean. Die Spezifikation verpflichtet den EJB-Container im Übrigen nicht dazu, das Instanzen-Pooling zu unterstützen.

Eine weitere Möglichkeit des EJB-Containers, Systemressourcen zu schonen, besteht darin, Enterprise-Bean-Instanzen, die derzeit nicht benötigt werden, z.B. über Objektserialisierung persistent zu machen und sie aus dem Speicher zu entfernen (*Passivierung*). Bei Bedarf können diese Instanzen wieder deserialisiert und im Speicher zur Verfügung gestellt werden (*Aktivierung*). Durch die temporäre Auslagerung von Bean-Instanzen (bei denen beispielsweise ein Zeitzähler abgelaufen ist) auf sekundäre Medien kann die Ressourcenbelastung des Servers reduziert werden.

Ob Bean-Instanzen gepoolt oder aktiviert bzw. passiviert werden können, hängt von ihrem Typ ab. Bei der Beschreibung der Bean-Typen im weiteren Verlauf dieses Buchs wird die notwendige Unterscheidung getroffen werden.

Verteilung (Laufzeitumgebung)

Der EJB-Container sorgt dafür, dass die Enterprise-Beans von Clientprogrammen aus benutzt werden können, die in der Regel nicht im selben Prozess laufen. Der Client weiß nicht, auf welchem Server sich die Enterprise-Bean, die er gerade benutzt, befindet. Auch wenn eine Enterprise-Bean eine andere Enterprise-Bean benutzt, ist sie ein Client. Der Ort, an dem die Enterprise-Bean exisitert, ist für den Client transparent. Die Benutzung einer auf einem anderen Rechner existierenden Enterprise-Bean unterscheidet sich für den Client nicht wesentlich von der Benutzung von Objekten, die sich im gleichen Adressraum befinden.

Für die verteilte Kommunikation zwischen den beteiligten Parteien wird Java RMI (Remote Method Invocation) benutzt. Zum Zweck der Interoperabilität unter Applikationsservern verschiedener Hersteller schreibt die EJB-Spezifikation für einen spezifikationskorformen EJB-Container die Unterstützung für das Kommunikationsprotokoll der CORBA-Spezifikation, IIOP, vor. Die verteilte Kommunikation findet daher über das Protokoll RMI-IIOP (RMI over IIOP) statt. In jedem Fall muss sich der Entwickler einer Enterprise-Bean nicht darum kümmern, dass sie von außen angesprochen werden kann. Diese Aufgabe fällt einzig und allein dem EJB-Container zu.

Dieses Buch setzt voraus, dass der Leser bereits mit der Technik der Remote Method Invocation vertraut ist (gegebenenfalls sind in [Kredel et al., 1999] und [Eckel, 1998] entsprechende Informationen zum Thema RMI zu finden). Dem Leser sollten insbesondere die Konzepte des *Stub* und des *Skeleton* geläufig sein.

Seit der Version 2.0 der Spezifikation der Enterprise JavaBeans besteht für den Client die Möglichkeit mit einer Enterprise-Bean über das so genannte *Local*-Interface zu kommunizieren. In vielen Anwendungen besteht die Notewendigkeit, dass Enterprise-Beans untereinander kommunizieren, die alle im selben EJB-Container installiert sind (Aggregation von Kompoenten). In der Version 1.1 der EJB-Spezifikation bedeutete das einen Remote-Aufruf an einer Komponente, die sich im selben Adressraum befindet. Der Overhead des RMI-Protokolls ist in diesem Fall überflüssig und führt zu Einbußen in der Performanz. Der Vorteil bei der Verwendung von Local-Interfaces ist, dass RMI dabei vollständig außen vor gelassen wird. Local-Interfaces können sinnigerweise nur dann verwendet werden, wenn sich Client und Enterprise-Bean im gleichen Adressraum einer virtuellen Java-Maschine befinden. Der Ort, an dem sich die Enterprise-Bean befindet, ist damit für den Client nicht mehr transparent. Die Ortstransparenz gilt nur für EJB-Komponenten, die über das Remote-Interface angesprochen werden. Außerdem ändert sich die Semantik eines Methodenaufrufs an einer Komponente bei der Verwendung von Local-Interfaces. Bei Aufrufen am Local-Interface werden Aufrufparameter per *Call-by-Reference* Semantik übergeben, während bei Aufrufen am Remote-Interface die *Call-by-Value* Semantik benutzt wird.

Die einschlägigen Kapitel zu den unterschiedlichen Bean-Typen werden sich intensiv mit dem Unterschied zwischen dem *Remote*- und dem *Local*-Interface auseinandersetzen und die Sicht des Clients auf eine Enterprise-Bean detailliert erläutern.

Namens- und Verzeichnisdienst (Dienst)

Damit ein Client eine Bean finden kann, ist er auf einen Namensdienst angewiesen. Ein Namensdienst bietet die Möglichkeit, Referenzen auf entfernte Objekte unter einem bestimmten Namen, der frei vergeben werden kann, an einem definierten Platz zu hinterlegen (*Binding*). Zum anderen bietet er die Möglichkeit, an den Namensdienst gebundene Objekte über deren Namen wiederzufinden (*Lookup*).

Ein Verzeichnisdienst ist leistungsfähiger als ein Namensdienst. Er unterstützt nicht nur das Binden von Referenzen an einen Namen, er kann verteilte Objekte und andere Ressourcen (wie z.B. Drucker, Dateien, Anwendungsserver etc.) in hierarchischen Strukturen verwalten und bietet weitergehende Möglichkeiten zur Administration. Durch einen Verzeichnisdienst können einem Client zu einer Referenz auf ein Remote-Objekt zusätzliche, beschreibende Informationen zur Verfügung gestellt werden.

Die Schnittstelle, über die der Namens- und Verzeichnisdienst angesprochen wird, ist JNDI (Java Naming and Directory Interface). Auch die Bean kann über den Namens- und Verzeichnisdienst Informationen bekommen. Der EJB-Container stellt der Bean-Instanz beispielsweise Informationen zur Verfügung, die bei der Installation der Komponente festgelegt wurden (so genannte Environment-Einträge). Dadurch besteht die Möglichkeit, das Verhalten von Enterprise-Beans durch Parametrisierung von außen zu beeinflussen. Die Bean hat auch die Möglichkeit, über den Namens- und Verzeichnisdienst auf bestimmte Ressourcen, wie z.B. Datenbankverbindungen oder einen Messaging-Dienst, zuzugreifen.

Persistenz (Dienst)

Der EJB-Container stellt den Beans über den Namens- und Verzeichnisdienst die Möglichkeit zur Verfügung, auf Datenbankverbindungen zuzugreifen. Die Beans können damit selbst dafür Sorge tragen, dass ihr Zustand persistent gemacht wird. Die Spezifikation der Enterprise JavaBeans sieht jedoch einen Mechanismus vor, mit dessen Hilfe der Zustand bestimmter Enterprise-Bean-Typen automatisch persistent gemacht werden kann (wir werden in Kapitel 5, Entity-Beans, auf diesen Mechanismus zurückkommen).

Bei der automatischen, durch den EJB-Container gesteuerten Speicherung des Zustands von Enterprise-Beans werden die Daten in der Regel in einer Datenbank persistent gemacht. Es sind andere EJB-Container denkbar, die Persistenzmechanismen anbieten, die die Daten in anderen Speichersystemen ablegen (z.B. im Dateisystem oder in elektronischen Archiven). Es besteht auch die Möglichkeit, EJB-Container zu entwickeln, die die Schnittstellen anderer Anwendungssysteme benutzen, um dort Daten abzulegen oder auszulesen. So können beispielsweise die Daten alter Mainframesysteme über spezielle EJB-Container in komponentenorientierte Systeme eingebunden werden (der Container dient dann gewissermaßen als Wrapper zu den Altsystemen).

Entscheidend jedoch ist die Tatsache, dass es im Fall der automatischen Persistenz einer Enterprise-Bean für die Bean unerheblich ist, wo die Daten abgelegt werden. Der EJB-Container trägt dafür Sorge, dass die Daten gespeichert werden und sich zu jedem Zeitpunkt in einem konsistenten Zustand befinden. So kann eine bestimmte Bean in verschiedenen EJB-Containern installiert werden, die jeweils unterschiedliche Spei-

chersysteme als Persistenzmedium unterstützen. Für die Enterprise-Bean bleibt die Persistenz transparent. Sie weiß nicht, wo ihre Daten gespeichert werden oder woher die Daten kommen, mit denen sie vom EJB-Container initialisiert wird.

Transaktionen (Dienst und Laufzeitumgebung)

Transaktionen sind eine bewährte Technik, um die Entwicklung verteilter Anwendungen zu erleichtern. Transaktionen unterstützen den Applikationsentwickler unter anderem bei der Behandlung von Fehlersituationen, die durch den gleichzeitigen Zugriff mehrerer Benutzer auf bestimmte Daten auftreten können.

Wenn der Entwickler Transaktionen benutzt, so unterteilt er die auszuführenden Aktionen in Einheiten (Transaktionen). Der Transaktionsmonitor (der EJB-Container) sorgt dafür, dass die einzelnen Aktionen einer Transaktion alle erfolgreich durchgeführt werden. Falls eine Aktion fehlschlägt, werden die bislang erfolgreich durchgeführten Einzelaktionen wieder rückgängig gemacht.

Die Unterstützung von Transaktionen ist ein wesentlicher Bestandteil der Spezifikation der Enterprise JavaBeans. In verteilten Systemen, in denen mehrere Benutzer gleichzeitig in vielen Einzelaktionen mit den gleichen Daten arbeiten (die unter Umständen auf mehrere Backend-Systeme verteilt sind), ist ein Transaktionsdienst auf der Ebene des Applikationsservers unverzichtbar. Es ist die Aufgabe des EJB-Containers, dafür zu sorgen, dass die notwendigen Protokolle (z. B. das Two-Phase-Commit-Protokoll zwischen einem Transaktionsmonitor und einem Datenbanksystem, die Context-Propagation und ein verteiltes Two-Phase-Commit) für die Behandlung von Transaktionen zur Verfügung stehen. Die Spezifikation der Enterprise JavaBeans unterstützt flache Transaktionen, d.h. Transaktionen können nicht geschachtelt werden.

Der Entwickler einer Enterprise-Bean kann wählen, wie er Transaktionen benutzen möchte. Er kann zum einen Transaktionen explizit benutzen, indem er direkt über JTA mit dem Transaktionsdienst des EJB-Containers kommuniziert. Zum anderen kann er sich für deklarative Transaktionen entscheiden. In diesem Fall wird bei der Installation einer Enterprise-Bean im EJB-Container angegeben, welche Methoden innerhalb welcher Transaktion ablaufen sollen. Der EJB-Container interveniert beim Aufruf dieser Methoden und sorgt dafür, dass sie innerhalb des entsprechenden Transaktionskontextes aufgerufen werden.

Im Fall der deklarativen Transaktionen muss sich der Bean-Entwickler nicht um die Handhabung der Transaktionen kümmern. Die Bean kann bei der Installation in EJB-Container A beispielsweise mit einem völlig anderen transaktionalen Verhalten installiert werden als bei der Installation in EJB-Container B. Die Bean selbst bleibt in jedem Fall davon unberührt, da der Container für die Sicherstellung des gewünschten transaktionalen Verhaltens verantwortlich zeichnet. Umfangreiche Details zu Transaktionen im EJB-Umfeld bietet Kapitel 7, Transaktionen.

Messaging (Dienst)

Mit der Version 2.0 der Spezifikation der Enterprise JavaBeans wird der EJB-Container dazu verpflichtet, einen Messaging-Dienst über die JMS-API (Java Message Service) einzubinden. Durch die Definition eines neuen Bean-Typs, der Message-Driven-Bean, wird der Messaging-Dienst auf signifikante Weise in den EJB-Container integriert. Die Entwicklung von Anwendungen auf Basis von Enterprise JavaBeans gewinnt damit zwei zusätzliche Dimensionen: Asynchronität und parallele Verarbeitung.

Grundsätzlich ermöglicht ein Messaging-System den asynchronen Austausch von Nachrichten zwischen zwei oder mehreren Clients. Anders als bei einem klassischen Client-Server-System ist die Architektur eines Messaging-Systems auf eine lose Koppelung gleichgestellter Partner ausgelegt. Jeder Client des Messaging-Systems kann Nachrichten asynchron senden und/oder empfangen. Der Sender einer Nachricht bleibt dabei weitgehend anonym, ebenso wie der bzw. die Empfänger. Messaging Systeme sind auch unter dem Namen *Message-oriented Middleware* (MOM) bekannt.

Neben der Verwendung von Message-Driven-Beans kann der Messaging-Dienst natürlich auch für den asynchronen Nachrichtenaustausch zwischen beliebigen Parteien benutzt werden. Enterprise-Beans können dabei ebenso Nachrichten verschicken wie Clients. Durch die Verwendung von Messaging lassen sich beispielsweise Prozesse voneinander entkoppeln oder auch Schnittstellen zu anderen Systemen schaffen.

Kapitel 6 wird sich eingehender mit dem Java Message Service und Message-Driven-Beans auseinandersetzen.

Sicherheit (Laufzeitumgebung)

Die Spezifikation verpflichtet den EJB-Container, den Enterprise-Beans eine Infrastruktur für das Sicherheitsmanagement als Bestandteil der Laufzeitumgebung zur Verfügung zu stellen. Es ist die Aufgabe des Systemadministrators und desjenigen, der die Enterprise-Beans installiert, die Sicherheitspolitik (Security Policy) festzulegen. Für die Umsetzung dieser Sicherheitspolitik ist einmal mehr der EJB-Container zuständig. Das Ziel dabei ist (ähnlich wie bei der containergesteuerten, automatischen Persistenz und den deklarativen Transaktionen), die Sicherheitsmechanismen für die Enterprise-Beans transparent zu machen, damit sie in möglichst vielen Systemen einsetzbar sind.

Würde die Sicherheitsstrategie in der Bean implementiert, wäre es problematisch, die gleiche Bean bei strengen und weniger strengen Anforderungen an die Sicherheit einzusetzen. Sehr viel sinnvoller dagegen ist die Verlagerung der Sicherheitsmechanismen in die Laufzeitumgebung der Komponenten. Somit sind sie gut wiederverwendbar und die Sicherheitspolitik kann von außen an die entsprechenden Umstände angepasst werden. Die Spezifikation der Enterprise JavaBeans spricht sich ausdrücklich dafür aus, dass sich im Code einer Bean möglichst keine sicherheitsrelevante Logik befinden soll.

Es besteht die Möglichkeit, Benutzerrollen zu definieren. In jeder Enterprise-Bean können einer bestimmten Rolle verschiedene Rechte erteilt werden. Diese Zuteilung findet, ebenso wie die Festlegung der Benutzerrollen,

- zum Zeitpunkt der Bean-Installation oder
- zu dem Zeitpunkt, zu dem mehrere Beans zu Aggregaten kombiniert werden,

statt. Die Rechte beziehen sich im Wesentlichen darauf, ob der Benutzer bestimmte Methoden einer Enterprise-Bean aufrufen darf oder nicht. Zur Laufzeit stellt der EJB-Container sicher, dass ein Clientaufruf einer Bean-Methode durchgeführt werden darf. Dazu vergleicht er die Rolle des Clients mit den Rechten der jeweiligen Enterprise-Bean-Methode.

Zusätzlich neben den soeben beschriebenen Sicherheitsmechanismen wird ein EJB-Container in der Regel folgende Sicherheitsleistungen anbieten:

- Authentifizierung des Benutzers durch eine Benutzerkennung und ein Passwort,
- Sichere Kommunikation (z.B durch den Einsatz von Secure-Socket-Layern).

Kapitel 8 wird sich noch detaillierter mit dem Thema Sicherheit im Zusammenhang mit Enterprise JavaBeans auseinandersetzen.

Abschließend lässt sich festhalten, dass der EJB-Container die zentrale Instanz im Komponentenmodell der Enterprise JavaBeans ist. Er stellt den Enterprise-Beans (den Komponenten) eine komfortable Laufzeitumgebung auf einem sehr hohen Abstraktionsniveau zur Verfügung und macht ihnen diverse Dienste über Standardschnittstellen zugänglich.

3.4 Der Persistence-Manager

Der Persistence-Manager ist dasjenige Bauteil in der Architektur der Enterprise JavaBeans, welches die automatische Persistenz bestimmter Komponenten ermöglicht. Er wurde mit der Version 2.0 der EJB-Spezifikation eingeführt, um eine bessere Trennung der physikalischen Datenspeicherung vom Objektmodell zu erreichen. Das Ziel dabei war, die Portierbarkeitkeit von persistenten EJB-Komponenten auf Applikationsserver anderer Hersteller zu verbessern. Dazu wurden verbesserte Möglichkeiten zur Abbildung einer persistenten Komponente auf das Speichermedium, eine Möglichkeit zum Aufbau deklarativer Beziehungen zwischen persistenten Komponenten und eine abstrakte Abfragesprache eingeführt. Bei der Version 1.1 war man im Fall der automatischen, container-gesteuerten Persistenz oft auf die Verwendung von proprietären Tools (sog. OR-Mapping-Tools) oder auf die Verwendung proprietärer Erweiterungen des EJB-Containers angewiesen, wodurch die Komponenten bezüglich ihrer Portabilität stark eingeschränkt waren.

Die Steuerung der Persistenz übernimmt nach wie vor der EJB-Container, d.h. er bestimmt, *wann* die Daten einer Komponente geladen bzw. gespeichert werden. Der EJB-Container bestimmt auch, ob im Falle des Scheiterns einer Aktion ein erfolgreich durchgeführter Speichervorgang wieder rückgängig gemacht werden muss (Transaktion). Der Persistence-Manager ist dagegen verantwortlich dafür, *wo* und *wie* die persistenten Daten gespeichert werden. Er übernimmt die Kommunikation mit dem Speichermedium (z. B. Datenbank). Die Abbildung der persistenten Daten einer Enterprise-Bean auf das Speichermedium (z. B. die Abbildung auf eine oder mehrere Datenbanktabellen) wird bei der Installation einer Komponente festgelegt. Der Persistence-Manager spielt keine Rolle, wenn sich die Enterprise-Bean selbst um Persistenz kümmert oder wenn die Komponente keine persistenten Daten besitzt.

In den meisten Fällen wird eine Datenbank zur Speicherung der Daten zum Einsatz kommen. Datenbanken verschiedener Hersteller sind trotz des ANSI-SQL-Standards untereinander nicht zu hundert Prozent kompatibel. Sie benutzen beispielsweise unterschiedliche Schlüsselwörter in der Syntax der Abfragesprache. Meist sind bestimmte Funktionen einer Datenbank, über die sie sich von anderen Herstellern abgrenzt, nur durch proprietäre Erweiterungen der Standard-Abfragesprache SQL nutzbar. Auch diese Schwierigkeiten soll der Persistence-Manager abfangen. Je nach eingesetzter Datenbank kann ein spezialisierter Persistence-Manager verwendet werden, der mit den Besonderheiten des Datenbanksystems umgehen kann.

Eine weitere Verantwortlichkeit des Persistence-Managers ist die Formulierung von Suchanfragen. Mit der Kenntnis über die Abbildung der Daten und über die Besonderheiten des verwendeten Speichermediums kann er abstrakte Suchanfragen in konkrete Suchanfragen übersetzen. Für die Formulierung von abstrakten Suchanfragen zum Auffinden von EJB-Komponenten stellt die Spezifikation der Enterprise JavaBeans eine Abfragesprache mit dem Namen EJB-QL (Enterprise JavaBeans Query Language) zur Verfügung.

Der Persistence-Manager und die Abfragesprache EJB-QL werden ausführlich in Kapitel 5 behandelt.

3.5 Enterprise-Beans

Enterprise-Beans sind die serverseitigen Komponenten, die in der Komponentenarchitektur der Enterprise JavaBeans zum Einsatz kommen. Sie implementieren die Anwendungslogik, auf die Clientprogramme zurückgreifen. Die Funktionalität des Servers und des EJB-Containers sorgt lediglich dafür, dass Beans benutzt werden können. Enterprise-Beans werden in einem EJB-Container installiert, der ihnen zur Laufzeit eine Umgebung bietet, in der sie existieren können. Enterprise-Beans greifen *implizit*

- bei containergesteuerter Persistenz,
- bei deklarativen Transaktionen,
- beim Empfangen von asynchronen Nachrichten oder
- bei der Sicherheit

bzw. *explizit*

- bei der Verwendung expliziter Transaktionen,
- bei Bean-gesteuerter Persistenz oder
- beim Versenden von asynchronen Nachrichten

auf die Dienste zurück, die ihnen der EJB-Container anbietet.

3.5.1 Typen von Enterprise-Beans

Enterprise-Beans gibt es in drei unterschiedlichen Ausprägungen, die sich mehr oder weniger stark voneinander unterscheiden: Entity-, Message-Driven- und Session-Beans. Tabelle 3.1 erläutert die wesentlichen Unterscheidungsmerkmale zwischen diesen drei Typen von Enterprise-Beans:

	Session-Bean	**Message-Driven-Bean**	**Entity-Bean**
Aufgabe der Bean	Repräsentiert einen serverseitigen Dienst, der Aufgaben für einen Client ausführt.	Repräsentiert serverseitige Geschäftslogik für die Verarbeitung asynchroner Nachrichten.	Repräsentiert ein Geschäftsobjekt, dessen Daten sich in einem dauerhaften Speicher befinden.
Zugriff auf die Bean	Eine Session-Bean ist für den Client eine private Ressource. Sie steht ihm exklusiv zur Verfügung.	Eine Message-Driven-Bean ist für den Client nicht direkt zugänglich. Eine Kommunikation mit ihr erfolgt ausschließlich über das Verschicken von Nachrichten über einen bestimmten Kanal des Messaging-Dienstes.	Die Entity-Bean ist eine zentrale Ressource; die Bean-Instanz wird von mehreren Clients gleichzeitig benutzt und ihre Daten stehen allen Clients zur Verfügung.

Tabelle 3.1: Maßgebliche Unterscheidungskriterien zwischen Session-, Message-Driven- und Entity-Beans (vgl. [EJB Developer Guide, 1999])

	Session-Bean	Message-Driven-Bean	Entity-Bean
Persistenz der Bean	Nicht persistent; wenn der verbundene Client oder der Server terminiert werden, ist die Bean nicht mehr verfügbar.	Nicht persistent. Wenn der Server terminiert wird, ist die Bean nicht mehr verfügbar. Die Nachrichten, die noch nicht an die Bean zugestellt wurden, sind gegebenenfalls persistent (Näheres dazu in Kapitel 6)	Persistent; wenn die verbundenen Clients oder der Server terminiert werden, befindet sich der Zustand der Entity-Bean auf einem persistenten Speichermedium; die Bean kann so zu einem späteren Zeitpunkt wiederhergestellt werden.

Tabelle 3.1: Maßgebliche Unterscheidungskriterien zwischen Session-, Message-Driven- und Entity-Beans (vgl. [EJB Developer Guide, 1999] (Fortsetzung))

Session-Beans modellieren für gewöhnlich Abläufe oder Vorgänge. Dabei handelt es sich beispielsweise um das Anlegen eines neuen Kunden in einem Warenwirtschaftssystem, die Durchführung einer Buchung in einem Buchungssystem oder das Erstellen eines Produktionsplans, der auf offenen Bestellungen basiert. Session-Beans kann man als den verlängerten Arm des Clients auf dem Server betrachten. Diese Betrachtungsweise wird durch die Tatsache unterstützt, dass eine Session-Bean eine private Ressource eines bestimmten Clients ist.

Entity-Beans hingegen repräsentieren Dinge des wirklichen Lebens, die mit bestimmten Daten assoziiert werden, wie z.B. ein Kunde, ein Buchungskonto oder ein Produkt. Eine Instanz eines bestimmten Entity-Bean-Typs kann von mehreren Clients gleichzeitig benutzt werden. Session-Beans operieren üblicherweise auf den Daten, die durch Entity-Beans repräsentiert werden.

Message-Driven-Beans sind Empfänger asynchroner Nachrichten. Als Mittler zwischen dem Versender einer Nachricht und der Message-Driven-Bean kommt ein Messaging-Dienst zum Einsatz. Entity- und Session-Beans werden direkt über das *Remote*- oder das *Local*-Interface angesprochen. Aufrufe an Entity- oder Session-Beans sind synchron, d.h. die Ausführung des Clients bleibt solange blockiert, bis die Methode der Enterprise-Bean abgearbeitet ist. Nachdem der Methodenaufruf zurückgekehrt ist, kann der Client seine Verarbeitung fortsetzen. Message-Driven-Beans können vom Cient nur (indirekt) durch das Schicken einer Nachricht über einen bestimmten Kanal des Messaging-Dienstes angesprochen werden. Ein bestimmter Typ einer Message-Driven-Bean empfängt alle Nachrichten, die über einen bestimmten Kanal des Messaging-Dienstes geschickt werden. Die Kommunikation über einen Messaging-Dienst ist asynchron. Das heißt, die Ausführung des Clients kann unmittelbar nach Absenden der Nachricht fortfahren. Sie bleibt nicht blockiert, bis die Nachricht zugestellt und verarbeitet ist. Der Container kann für die Verarbeitung von Nachrichten eines bestimmten Kanals mehrere Instanzen eines bestimmten Message-Driven-Bean-

Typs einsetzen. Somit ist in diesem Fall parallele Verarbeitung möglich. Message-Driven-Beans haben keinen Zustand zwischen der Verarbeitung mehrerer Nachrichten. Außerdem haben sie dem Client gegenüber keine Identität. In gewisser Weise ähneln sie damit den *stateless* Session-Beans (siehe unten). Message-Driven-Beans können für die Verarbeitung einer Nachricht Session- oder Entity-Beans sowie alle Dienste, die der Container anbietet, benutzen.

Bei Session-Beans lässt sich eine weitere Unterscheidung treffen, nämlich ob die Session-Bean zustandslos (*stateless*) oder zustandsbehaftet (*stateful*) ist. Zustandslose Session-Beans speichern von einem Methodenaufruf zum nächsten keine Daten. Die Methoden einer zustandslosen Session-Bean arbeiten nur mit den Daten, die ihr als Parameter übergeben werden. Zustandslose Session-Beans des gleichen Typs besitzen alle die gleiche Identität. Da sie keinen Zustand haben, besteht weder die Notwendigkeit noch die Möglichkeit, sie zu unterscheiden.

Zustandsbehaftete Session-Beans dagegen speichern Daten über mehrere Methodenaufrufe hinweg. Aufrufe von Methoden an zustandsbehafteten Session-Beans können den Zustand der Bean verändern. Der Zustand geht verloren, wenn der Client die Bean nicht mehr benutzt oder der Server heruntergefahren wird. Zustandsbehaftete Session-Beans des gleichen Typs haben zur Laufzeit unterschiedliche Identitäten. Der EJB-Container muss sie unterscheiden können, da sie für ihre Clients jeweils unterschiedliche Zustände verwalten.

Einer Session-Bean wird vom EJB-Container ihre Identität zugewiesen. Im Gegensatz zu Entity-Beans ist die Identität einer Session-Bean nach außen hin nicht sichtbar. Da Clients immer mit einer für sie exklusiven Instanz einer Session-Bean arbeiten, ist dafür auch keine Notwendigkeit gegeben.

Entity-Beans lassen sich dadurch unterscheiden, ob sie selbst dafür verantwortlich sind, dass ihre Daten persistent gemacht werden, oder ob der EJB-Container diese Aufgabe übernimmt. Im ersten Fall spricht man von *bean-managed Persistence*, im zweiten Fall von *container-managed Persistence*.

Entity-Beans des gleichen Typs haben zur Laufzeit unterschiedliche Identitäten. Eine Entity-Bean eines bestimmten Typs wird zur Laufzeit durch ihren Primärschlüssel identifiziert, der ihr vom EJB-Container zugeteilt wird. Dadurch wird sie an bestimmte Daten gebunden, die sie in ihrer Aktivitätsphase repräsentiert. Die Identität einer Entity-Bean ist nach außen hin sichtbar.

Die Bean-Typen spielen beim Ressourcenmanagement des EJB-Containers eine Rolle. Bei Entity-Beans, Message-Driven-Beans und zustandslosen Session-Beans kann der Container das Pooling, bei zustandsbehafteten Session-Beans kann er die Passivierung bzw. Aktivierung (Serialisierung bzw. Deserialisierung auf ein sekundäres Speichermedium) einsetzen.

Die Schnittstelle zwischen einer Entity-Bean und dem EJB-Container ist der so genannte *Kontext* (*javax.ejb.EJBContext*). Dieses Interface wird für die drei Bean-Typen nochmals spezialisiert (zu *javax.ejb.EntityContext*, *javax.ejb.MessageDrivenContext* und *javax.ejb.SessionContext*). Über den Kontext, der der Bean vom EJB-Container übergeben wird, kann die Bean mit dem Container kommunizieren. Der Kontext bleibt mit einer Bean während ihrer gesamten Lebensdauer verbunden. Der EJB-Container verwaltet über den Kontext die Identität einer Enterprise-Bean. Über eine Änderung des Kontextes kann der EJB-Container die Identität einer Bean verändern.

Die Kapitel 4, 5 und 6 setzen sich ausführlich mit den technischen Details von Session-, MessageDriven und Entity-Beans auseinander. Abschnitt 9.2 beschäftigt sich eingehend mit der unterschiedlichen Semantik der Bean-Typen.

3.5.2 Bestandteile einer Enterprise-Bean

Zu einer Enterprise-Bean-Komponente gehören folgende Bestandteile:

- das Remote- und das (Remote-) Home-Interface **oder**
- das Local- und das *Local-Home*-Interface (bei Entity- und Session-Beans),
- die Bean-Klasse (bei Entity-, Message-Driven und Session-Beans),
- der Primärschlüssel bzw. die Primärschlüsselklasse (bei Entity-Beans),
- der Deployment-Deskriptor (bei Entity-, Message-Driven und Session-Beans).

Man spricht vom *Remote-Client-View*, wenn eine Enterprise-Bean über das Remote-Interface ansprechbar ist. Benutzt eine Enterprise-Bean das Local-Interface, spricht man vom *Local-Client-View*. Grundsätzlich kann eine Enterprise-Bean sowohl den Local- als auch den Remote-Client-View unterstützen. Die Spezifikation rät jedoch dazu, sich für einen der beiden Fälle zu entscheiden.

Die einzelnen Bestandteile einer Bean sollen anhand eines Beispiels vorgestellt werden. Es soll eine Entity-Bean entwickelt werden, die ein Bankkonto repräsentiert. Die Komponente soll es ermöglichen, die Kontonummer des Bankkontos, die Beschreibung des Kontos und den aktuellen Kontostand zu erfragen. Außerdem soll der Kontostand um einen bestimmten Betrag erhöht bzw. vermindert werden können. Die Bankkonto-Bean soll von Remote-Clients angesprochen werden können, also von Clients, die sich außerhalb des Adressraums der Bankkonto-Bean befinden.

Dieses Kapitel konzentriert sich auf die Darstellung der Besonderheiten, die durch die Architektur bedingt sind. Wir werden bei diesem Beispiel nicht auf die Besonderheiten eines bestimmten Bean-Typs eingehen. Das wird ausführlich in den Kapiteln 4, 5 und 6 getan. Da eine Entity-Bean alle oben aufgeführten Bestandteile aufweist, eignet sie sich für die Einführung am besten. Ferner werden wir bei diesem Beispiel kein Local-Inter-

face verwenden. Da EJB eine verteilte Komponentenarchitektur ist, ist die Verwendung des Remote-Interface der Standardfall. Die Verwendung des Local-Interface ist analog zur Verwendung des Remote-Interface und wird im Detail in den Kapiteln 4 und 5 behandelt.

Remote-Interface

Das Remote-Interface definiert diejenigen Methoden, die von einer Bean nach außen hin angeboten werden. Die Methoden des Remote-Interface spiegeln demnach die Funktionalität wider, die von der Komponente erwartet bzw. gefordert wird. Das Remote-Interface muss von *javax.ejb.EJBObject* abgeleitet werden, das wiederum von *java.rmi.Remote* abgeleitet ist. Alle Methoden des Remote-Interfaces müssen die Ausnahme *java.rmi.RemoteException* deklarieren.

```
package ejb.bankaccount;

import java.rmi.RemoteException;
import javax.ejb.EJBObject;

public interface BankAccount extends EJBObject {
    //Erfragen der Kontonummer
    public String getAccNumber()
        throws RemoteException;
    //Erfragen der Kontobezeichnung
    public String getAccDescription()
        throws RemoteException;
    //Erfragen des Kontostandes
    public float getBalance()
        throws RemoteException;
    //Erhöhen des Kontostandes
    public void increaseBalance(float amount)
        throws RemoteException;
    //Vermindern des Kontostandes
    public void decreaseBalance(float amount)
        throws RemoteException;
}
```

Listing 3.1: Remote-Interface von BankAccount

Home-Interface

Das Home-Interface muss von *javax.ejb.EJBHome* abgeleitet sein (in diesem Interface befindet sich die Methode zum Löschen einer Bean; sie muss nicht gesondert deklariert werden). *EJBHome* seinerseits ist ebenfalls von *java.rmi.Remote* abgeleitet. Auch beim Home-Interface deklarieren alle Methoden das Auslösen einer Ausnahme vom Typ *java.rmi.RemoteExeption*. Ebenso wie beim Remote-Interface deuten alle Sachverhalte auf den verteilten Charakter und die Einbettung in das EJB-Framework hin.

```
package ejb.bankaccount;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;
import javax.ejb.FinderException;

public interface BankAccountHome extends EJBHome {

    //Erzeugen eines Kontos
    public BankAccount create(String accNo,
                              String accDescription,
                              float  initialBalance)
        throws CreateException, RemoteException;

    //Finden eines bestimmten Kontos
    public BankAccount findByPrimaryKey(String accPK)
        throws FinderException, RemoteException;

}
```

Listing 3.2: Home-Interface von BankAccount

Bean-Klasse

Die Bean-Klasse implementiert die Methoden, die im Home- und im Remote-Interface deklariert wurden (mit Ausnahme der *findByPrimaryKey*-Methode), ohne diese beiden Interfaces im Sinne des Schlüsselwortes *implements* tatsächlich einzubinden. Die Signaturen der Methoden des Remote-Interfaces müssen mit den entsprechenden Methoden in der Bean-Klasse übereinstimmen. Die Bean-Klasse muss in Abhängigkeit von ihrem Typ ein Interface implementieren, und zwar *javax.ejb.EntityBean*, *javax.ejb.MessageDrivenBean* oder *javax.ejb.SessionBean*. Die Bean implementiert weder ihr Home- noch ihr Remote-Interface. Nur im Falle einer Entity-Bean mit container-gesteuerter, automatischer Persistenz ist die Klasse *abstrakt*. Die Klassen von Session-, Message-Driven- und Entity-Beans, die sich selbst um ihre Persistenz kümmern, sind *konkrete* Klassen.

```
package ejb.bankaccount;

import javax.ejb.CreateException;
import javax.ejb.EntityBean;
import javax.ejb.EntityContext;
import javax.ejb.RemoveException;

public abstract class BankAccountBean implements EntityBean {

    private EntityContext theContext;

    public BankAccountBean() {
    }
```

```
//Die create-Methode des Home-Interface

public String ejbCreate(String accNo,
                        String accDescription,
                        float  initialBalance)
    throws CreateException
{
    setAccountNumber(accNo);
    setAccountDescription(accDescription);
    setAccountBalance(initialBalance);
    return null;
}

public void ejbPostCreate(String accNo,
                          String accDescription,
                          float  initialBalance)
    throws CreateException
{
}

//Abstrakte getter-/setter-Methoden

public abstract String getAccountNumber();
public abstract void   setAccountNumber(String acn);

public abstract String getAccountDescription();
public abstract void   setAccountDescription(String acd);

public abstract float  getAccountBalance();
public abstract void   setAccountBalance(float acb);

//Die Methoden des Remote-Interface

public String getAccNumber() {
    return getAccountNumber();
}

public String getAccDescription() {
    return getAccountDescription();
}

public float getBalance() {
    return getAccountBalance();
}

public void increaseBalance(float amount) {
    float acb = getAccountBalance();
    acb += amount;
    setAccountBalance(acb);
}

public void decreaseBalance(float amount) {
```

```
            float acb = getAccountBalance();
            acb -= amount;
            setAccountBalance(acb);
        }

        //Die Methoden des javax.ejb.EntityBean-Interface

        public void setEntityContext(EntityContext ctx) {
            theContext = ctx;
        }

        public void unsetEntityContext() {
            theContext = null;
        }

        public void ejbRemove()
            throws RemoveException
        {}

        public void ejbActivate() {
        }

        public void ejbPassivate() {
        }

        public void ejbLoad() {
        }

        public void ejbStore() {
        }
    }
```

Listing 3.3: Bean-Klasse von BankAccount

Der Primärschlüssel (Primärschlüsselklasse)

Der Primärschlüssel ist nur bei Entity-Beans relevant. Er dient dazu, eine Entität eines bestimmten Typs eindeutig zu identifizieren. Ähnlich dem Primärschlüssel einer Datenbanktabelle enthält er diejenigen Attribute, die für die eindeutige Identifikation notwendig sind. Über den Primärschlüssel kann eine bestimmte Entität gefunden werden, die dann vom EJB-Container mit einer Entity-Bean-Instanz des passenden Typs assoziiert wird. Über den Primärschlüssel wird die Identität einer Entity-Bean nach außen sichtbar. Die Primärschlüsselklasse ist für Session- und Message-Driven-Beans nicht von Bedeutung, da deren Identität niemals nach außen hin sichtbar wird. Die Spezifikation unterscheidet zwischen zwei Arten von Primärschlüsseln:

- Primärschlüssel, die sich auf ein Feld der Entity-Bean-Klasse beziehen,
- Primärschlüssel, die sich auf mehrere Felder der Entity-Bean-Klasse beziehen.

Ein Primärschlüssel, der sich auf nur ein Feld der Entity-Bean-Klasse bezieht, kann durch eine Standard-Java-Klasse (z.B. *java.lang.String*, *java.lang.Integer* etc.) repräsentiert werden. In unserem Beispiel ist die Klasse *java.lang.String* die Primärschlüsselklasse, da die eindeutige Identifizierung eines Kontos über die (alphanumerische) Kontonummer möglich ist.

Ein Primärschlüssel, der sich auf mehrere Felder der Entity-Bean-Klasse bezieht, wird in der Regel durch eine eigens dafür entwickelte Klasse repräsentiert. Eine solche Klasse muss eine öffentliche (public) Klasse sein und sie muss einen öffentlichen (public) Konstruktor ohne Argumente haben. Die Felder der Primärschlüsselklasse, welche den Primärschlüssel der Entity-Bean repräsentieren, müssen namentlich denen der Entity-Bean-Klasse entsprechen. Außerdem müssen diese Felder ebenfalls öffentlich (public) sein. Die Klasse muss RMI-IIOP kompatibel (serialisierbar) sein und sie muss die Methoden *equals()* und *hashCode()* implementieren. Listing 3.4 zeigt ein Beispiel einer solchen Primärschlüsselklasse für eine Konto-Bean, die zur eindeutigen Identifikation neben der Kontonummer auch die Nummer des Mandanten benötigt (mandantenfähiges System).

```
package ejb.custom;

public class CustomAccountPK implements java.io.Serializable {

    public String clientNumber;
    public String accountNumber;

    public CustomAccountPK() {
    }

    public int hashCode() {
        return clientNumber.hashCode() ^
                accountNumber.hashCode();
    }

    public boolean equals(Object obj) {
        if(!(obj instanceof CustomAccountPK)) {
            return false;
        }
        CustomAccountPK pk = (CustomAccountPK)obj;
        return (clientNumber.equals(pk.clientNumber)
             && accountNumber.equals(pk.accountNumber));
    }

    public String toString() {
        return clientNumber + ":" + accountNumber;
    }
}
```

Listing 3.4: Beispiel einer Primärschlüsselklasse für mehrteilige Primärschlüssel

Der Deployment-Deskriptor

Der Deployment-Deskriptor ist eine Datei im XML-Format (Details zu XML sind in [Behme et al., 1998] zu finden) und beschreibt eine oder mehrere Beans bzw. wie mehrere Beans zu Aggregaten zusammenzufassen sind. Im Deployment-Deskriptor werden all diejenigen Informationen abgelegt, die nicht im Code einer Bean zu finden sind. Dabei handelt es sich im Wesentlichen um deklarative Informationen. Diese Informationen sind insbesondere für diejenigen Personen wichtig, die mehrere Enterprise-Beans zu Anwendungen zusammenfassen bzw. Enterprise-Beans in einem EJB-Container installieren. Dem EJB-Container wird über den Deployment-Deskriptor mitgeteilt, wie er die Komponente(n) zur Laufzeit zu behandeln hat.

Der Deployment-Deskriptor enthält zum einen Informationen über die Struktur einer Enterprise-Bean und ihre externen Abhängigkeiten (z.B. zu anderen Beans oder zu bestimmten Ressourcen wie Verbindungen zu einer Datenbank). Zum anderen enthält er Informationen darüber, wie sich die Komponente zur Laufzeit verhalten soll bzw. wie sie mit anderen Komponenten zu komplexeren Bausteinen kombiniert werden kann. Zur Veranschaulichung soll ein für unsere Beispiel-Bean passender, vollständiger Deployment-Deskriptor gezeigt werden (für eine vollständige Beschreibung des Deployment-Deskriptors vergleiche [Sun Microsystems, 2001]) :

```
<?xml version="1.0" ?>
<!DOCTYPE ejb-jar PUBLIC
 "-//Sun Microsystems, Inc.//DTD EnterpriseJavaBeans 2.0//EN"
 "http://java.sun.com/dtd/ejb-jar_2_0.dtd">
<ejb-jar>
  <description>
    Dieser Deployment-Deskriptor enthält Informationen
    über die Entity-Bean BankAccount.
  </description>
  <enterprise-beans>
    <entity>
      <!-- Name der Enterprise-Bean -->
      <ejb-name>BankAccount</ejb-name>
      <!-- Klasse des Home-Interfaces -->
      <home>ejb.bankaccount.BankAccountHome</home>
      <!-- Klasse des Remote-Interfaces -->
      <remote>ejb.bankaccount.BankAccount</remote>
      <!-- Klasse der Enterprise-Bean -->
      <ejb-class>ejb.bankaccount.BankAccountBean</ejb-class>
      <!-- Art der Persistenz -->
      <persistence-type>Container</persistence-type>
      <!-- Klasse des Primärschlüssels -->
      <prim-key-class>java.lang.String</prim-key-class>
      <!-- Gibt an, ob die Implementierung der Enterprise-
       Bean reentrant ist, oder nicht -->
      <reentrant>False</reentrant>
```

```
      <!-- Die EJB-Version, für die diese Enterprise-
       Bean entwickelt wurde -->
      <cmp-version>2.x</cmp-version>
      <!-- Name des Persistenzschemas -->
      <abstract-schema-name>AccountBean
      </abstract-schema-name>
      <!-- Liste der persistenten Attribute der
       Enterprise-Bean -->
      <cmp-field>
        <description>Die Kontonummer</description>
        <field-name>accountNumber</field-name>
      </cmp-field>
      <cmp-field>
        <description>Die Kontobeschreibung</description>
        <field-name>accountDescription</field-name>
      </cmp-field>
      <cmp-field>
        <description>Der Kontostand</description>
        <field-name>accountBalance</field-name>
      </cmp-field>
      <!-- Primärschlüssel-Feld -->
      <primkey-field>accountNumber</primkey-field>
    </entity>
  </enterprise-beans>
  <assembly-descriptor>
    <!-- Definition der Benutzerrolle 'Banker' -->
    <security-role>
      <description> Die Rolle Bankangestellter
      </description>
      <role-name>Banker</role-name>
    </security-role>
    <!-- Definition der Zugriffsrechte auf Methodenebene -->
    <method-permission>
      <role-name>banker</role-name>
      <method>
        <ejb-name>BankAccount</ejb-name>
        <method-name>*</method-name>
      </method>
    </method-permission>
    <!-- Definition des transaktionalen Verhaltens
      pro Methode der Enterprise-Bean -->
    <container-transaction>
      <method>
        <ejb-name>BankAccount</ejb-name>
        <method-name>increaseBalance</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
    <container-transaction>
      <method>
        <ejb-name>BankAccount</ejb-name>
```

```
        <method-name>decreaseBalance</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>
```

Listing 3.5: Deployment-Deskriptor von BankAccount

Da es sich in unserem Beispiel um eine Entity-Bean mit Container-Managed-Persistence handelt, müssen noch Angaben für den Persistence-Manager gemacht werden. Er muss wissen, welche Felder der Bean auf welche Spalten in welcher Tabelle abgebildet werden sollen. Zudem benötigt er Angaben darüber, in welcher Datenbank die Daten gespeichert werden sollen. Die Spezifikation schreibt nicht vor, in welcher Form diese Angaben zu machen sind. Sie schreibt jedoch vor, dass der Hersteller Tools liefern muss, mit deren Hilfe derartige Angaben gemacht werden können. Damit ist auch klar, dass sich diese Tools von Hersteller zu Hersteller unterscheiden, ebenso wie das Format, in dem die Angaben abgelegt werden. Im einfachsten Fall könnten diese Angaben in einer Datei im XML-Format gemacht werden. Listing 3.6 zeigt ein fiktives Beispiel. Im weiteren Verlauf dieses Kapitels werden wir diese Angaben als *Persistence-Deskriptor* bezeichnen.

```
<?xml version="1.0" ?>
<abstract-schema>
  <name>AccountBean</name>
  <data-source>
    <name>test-db</name>
    <type>oracle</type>
  </data-source>
  <table-name>account</table-name>
  <field-mapping>
    <bean-field>accountNumber</bean-field>
    <column>acno</column>
  </field-mapping>
  <field-mapping>
    <bean-field>accountDescription</bean-field>
    <column>acdesc</column>
  </field-mapping>
  <field-mapping>
    <bean-field>accountBalance</bean-field>
    <column>acbal</column>
  </field-mapping>
</abstract-schema>
```

Listing 3.6: Fiktives Beispiel für Mapping-Angaben für den Persistence-Manager

Sind alle Bestandteile vorhanden, so werden je nach Bean-Typ Home- und Remote-Interface(s), die Bean-Klasse(n), die Primärschlüsselklasse(n) und der Deployment-Deskriptor (der die Beschreibung mehrerer Enterprise-Bean-Komponenten enthalten

kann) in eine Datei im jar-Format verpackt. Die Abkürzung jar steht dabei für Java-Archiv und entspricht dem gängigen ZIP-Format. Damit sind die Bestandteile einer Enterprise-Bean komplett und als Komponente(n) in einer jar-Datei verpackt (weitere Details zum Verpacken einer Komponente in einer jar-Datei sind in [Sun Microsystems, 2001] zu finden).

Abschließend soll die Aufstellung in Tabelle 3.2 nochmals verdeutlichen, welche Bestandteile bei welchem Bean-Typ relevant sind.

	Entity-Bean		**Session-Bean**	**Message-Driven-Bean**
	container-managed	**bean-managed**		
Remote- bzw. Local-Interface				
LocalHome-bzw. Home-Interface				
Konkrete Bean-Klasse				
Abstrakte Bean-Klasse				
Deployment-Deskriptor				
Persistence-Deskriptor				

Tabelle 3.2: Überblick über die Bestandteile unterschiedlicher Bean-Typen

3.6 Wie alles zusammenspielt

Wir wollen davon ausgehen, dass die Komponente *Bankkonto* nicht in ein Aggregat eingeht, sondern direkt in einem EJB-Container installiert wird, um von Clientprogrammen benutzt zu werden. Die Installation erfolgt mit Hilfe von entsprechenden Tools. Die Spezifikation verpflichtet den Hersteller eines EJB-Containers und des Persistence-Managers, diese Tools zu liefern. Wie die Tools aussehen und wie sie zu bedienen sind, ist dem Hersteller überlassen. In der Regel werden sie die Erfassung aller relevanten Angaben für die Installation einer Komponente über eine grafische Benutzeroberfläche unterstützen. Entscheidend ist jedoch das Ergebnis des toolunterstützten Installationsvorgangs. Er liefert die fehlenden Bindeglieder in der EJB-Architektur: die Implementierung des Home- und des Remote-Interface (bzw. des Local-Home- und des Local-Interface) der Enterprise-Bean und im Falle einer container-managed Entity-Bean die Implementierung der konkreten Bean-Klasse. Abbildung 3.3 stellt diese Zusammenhänge dar.

Wie bereits erwähnt wurde, verwendet eine Enterprise-Bean entweder Local- oder Remote-Interfaces. Die Implementierungsklasse des Home bzw. des Local-Home-Interface wird üblicherweise als *EJBHome*, die des Local- bzw. Remote-Interface üb-

licherweise als *EJBObject* bezeichnet. *EJBHome* und *EJBObject* werden durch die Tools des Containerherstellers aus den Bestandteilen einer Bean generiert.

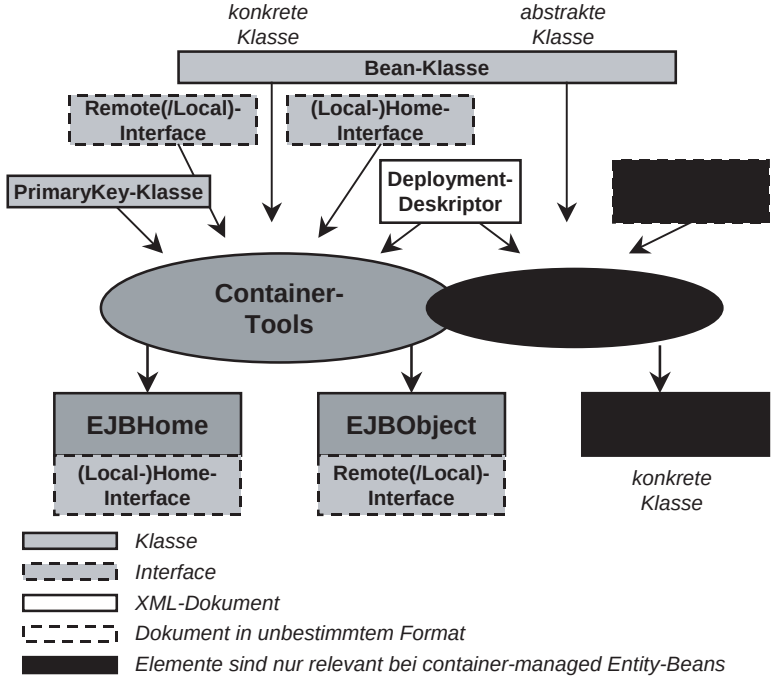

Abbildung 3.3: Generierung fehlender Architekturbestandteile

Falls es sich um eine Entity-Bean mit container-managed-Persistence handelt, ist es die Aufgabe des Persistence-Managers, eine konkrete Klasse zu generieren. Diese leitet von der abstrakten Bean-Klasse ab und stellt den für die Persistenz notwenigen Code bereit. Für die Generierung werden die zusätzlich gemachten Angaben über die Datenbank und die Tabelle(n) aus dem Persistence-Deskriptor sowie die persistenten Felder der Komponenten benutzt (vergleiche dazu auch das Beispiel in Listing 3.6).

Das *EJBHome*-Objekt dient zur Laufzeit als eine Art Objekt-Factory, das *EJBObject*-Objekt als eine Art funktionale Hülle (*Wrapper*) für die jeweilige Enterprise-Bean. Sie sind der verlängerte Arm der Laufzeitumgebung des EJB-Containers für bestimmte Bean-Typen. Verwendet die Enterprise-Bean Remote-Interfaces sind sie beide Remote-Objekte im Sinne der Java-RMI. Verwendet die Enterprise-Bean Local-Interfaces, sind sie herkömmliche Java-Objekte. In beiden Fällen ruft der Client die Methoden an den *EJBHome*- und *EJBObject*-Objekten auf, niemals direkt an der Bean-Instanz. *EJBHome* und *EJBObject* delegieren die entsprechenden Methodenaufrufe (nach bzw. vor bestimmten containerbezogenen Routinen) an die Bean-Instanz. Durch diese Indirektion hat der EJB-Contai-

ner die Möglichkeit, seine Implementierung für die deklarativen Angaben im Deployment-Deskriptor einzubringen. Der Inhalt des Deployment-Deskriptors beeinflusst den generierten Code der *EJBHome*- und der *EJBOject*-Klasse maßgeblich.

Die *EJBHome*-Klasse enthält den Code für das Erzeugen, Finden und das Löschen von Bean-Instanzen. Oftmals wird auch Code für das Ressourcenmanagement in die *EJBHome*-Klasse generiert. Die *EJBObject*-Klasse implementiert das transaktionale Verhalten, die Sicherheitsüberprüfungen und gegebenenfalls die Logik für die containergesteuerte Persistenz. Die Implementierung dieser Klassen ist in hohem Maße herstellerabhängig. Die Spezifikation macht dem Hersteller bezüglich der Implementierung von *EJBHome* und *EJBObject* keine zwingenden Vorschriften.

Keine Instanz außer den *EJBHome*- und *EJBObject*-Objekten kann mit der Bean-Instanz kooperieren. Sie ist durch die Containerklassen vollständig abgeschirmt. Auch die Kommunikation zwischen Beans findet immer über die Containerklassen *EJBHome* und *EJBObject* statt.

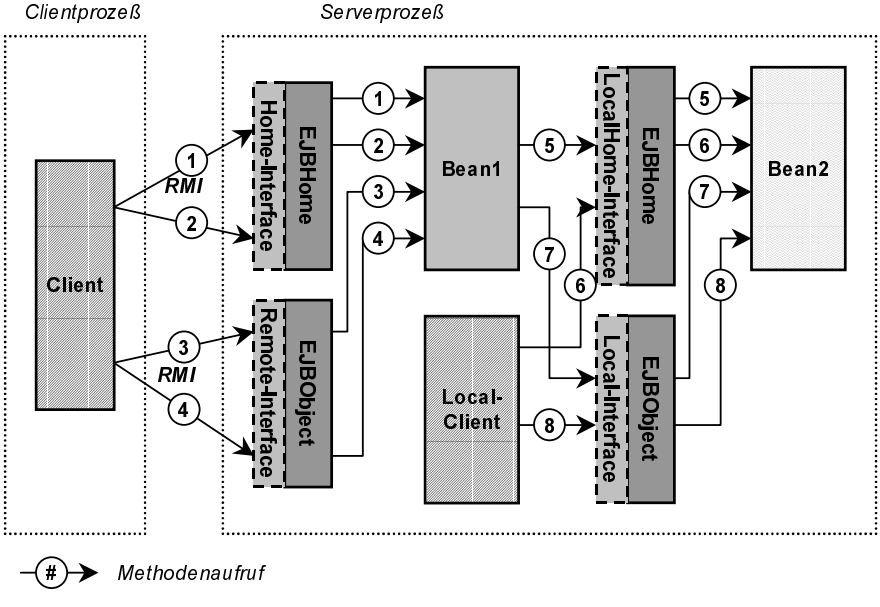

Abbildung 3.4: EJBHome und EJBObject zur Laufzeit

Aus diesen Umständen heraus klären sich die Fragen, die eventuell im vorangegangenen Abschnitt unbeantwortet geblieben sind. Das Home- und das Remote-Interface müssen nicht implementiert werden, da die Implementierungsklassen von den Containertools generiert werden (Gleiches gilt, falls ein Local-Home- und ein Local-Interface

verwendet werden). Die Enterprise-Bean ist kein Remote-Objekt, da sie gar nicht von außen angesprochen werden soll. Sie soll nur über *EJBHome* und *EJBObject* angesprochen werden können. Der EJB-Container hätte sonst keine Interventionsmöglichkeit, um seinen Aufgaben nachkommen zu können. Deswegen sind die Interfaces *javax. ejb.EJBHome* und *javax.ejb.EJBObject* (die Basisinterfaces des Home- und des Remote-Interface) auch von *java.rmi.Remote* abgeleitet.

Auch der Umstand, weshalb die Signaturen der Bean-Methoden und der Methoden im Home- und Remote-Interface übereinstimmen müssen, wird klar. Implementiert werden die Interfaces durch die *EJBHome*- und *EJBObject*-Klassen. Sie delegieren die Aufrufe an den Interface-Methoden auf die entsprechenden Methoden der Bean-Klasse. Der dafür notwendige Code wird generiert. Passen die Signaturen der in den Interfaces deklarierten Methoden nicht zu den entsprechenden Bean-Methoden oder fehlen diese Methoden gar in der Bean-Klasse, so wird dies bei der Generierung von *EJBHome* und *EJBObject* durch die Container-Tools beanstandet oder führt zu Laufzeitfehlern.

EJBHome und *EJBObject* sind es letztlich auch, die sicherstellen, dass im Falle von Session-Beans jeder Client mit einer für ihn exklusiven Instanz arbeiten kann, während sich im Fall von Entity-Beans mehrere Clients die gleiche Instanz teilen können.

3.7 Die Sicht des Clients

Wenn ein Client nun die im EJB-Container installierte Bankkonto-Bean benutzen möchte, so muss er die Bean zunächst finden. Dazu benutzt er einen Namens- und Verzeichnisdienst. Dieser wird über die JNDI-Schnittstelle angesprochen. Ebenso wie die Enterprise-Bean benutzt der Client den Namens- und Verzeichnisdienst des EJB-Containers bzw. des Servers, in dem die Enterprise-Bean installiert ist. Der EJB-Container ist dafür verantwortlich, die Bean unter einem bestimmten Namen über den Namens- und Verzeichnisdienst zugänglich zu machen. In vielen Fällen wird dafür der im Deployment-Deskriptor angegebene Name der Bean verwendet. Das Feld des Deployment-Deskriptors, in dem der Name der Enterprise-Bean eingetragen wird, heißt *ejb-name* (und war in unserem Beispiel *BankAccount*). Wir wollen davon ausgehen, dass der EJB-Container diesen Namen für die Veröffentlichung benutzt. Folgendes Codefragment zeigt, wie ein Client die Bankkonto-Bean findet:

```
//Je nach Hersteller des Containers bzw. Servers
//müssen im Environment entsprechende Einstellungen
//vorgenommen werden, um den richtigen Context
//zu erzeugen.
final String BANK_ACCOUNT =
                    "java:comp/env/ejb/BankAccount";
//Erzeugen des Kontextes für den Zugang zum Namensdienst
InitialContext ctx = new InitialContext();
```

```
//Auffinden der Bean BankAccount
Object o = ctx.lookup(BANK_ACCOUNT);
//Typumwandlung; Details hierzu in Kapitel 4.3
BankAccountHome bh = (BankAccountHome)
   PortableRemoteObject.narrow(o,BankAccountHome.class);
```

Listing 3.7: Finden des Home-Interface mittels JNDI

Der EJB-Container stellt der Enterprise-Bean über den Namens- und Verzeichnisdienst eine Instanz des Client-Stubs der Implementierungsklasse des Home-Interface (*EJB-Home*) zur Verfügung.

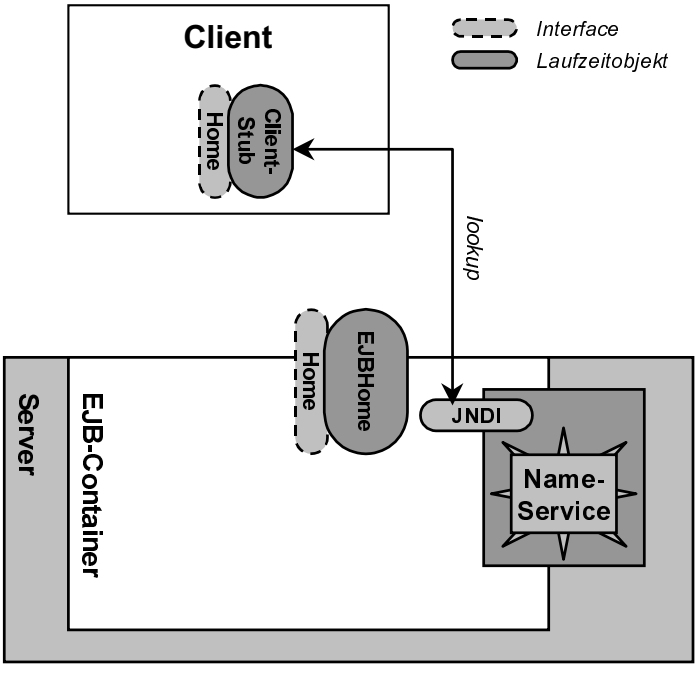

Abbildung 3.5: Finden einer Enterprise-Bean über JNDI

Über die Methoden des Home-Interface (und die des Interface *javax.ejb.EJBHome*) kann der Client den Lebenszyklus der Bean steuern. Beispielsweise kann er mit diesem Codefragment ein neues Konto erzeugen:

```
BankAccount ba = bh.create("0815", "Beispielkonto", 0.0);
```

Das *EJBHome*-Objekt erzeugt eine neue Bean-Instanz (oder nimmt eine aus dem Pool), erzeugt einen neuen Datensatz in der Datenbank und erzeugt eine *EJBObject*-Instanz. Aus den Parametern der *create*-Methode wird ein Primärschlüsselobjekt erzeugt und

mit der Bean-Instanz (über den Kontext der Bean) assoziiert. Damit bekommt die Bean-Instanz ihre Identität. Als Ergebnis der Operation wird dem Client der Stub der *EJBObject*-Instanz zur Verfügung gestellt. Der Stub repräsentiert dem Client gegenüber das Remote-Interface der Bean.

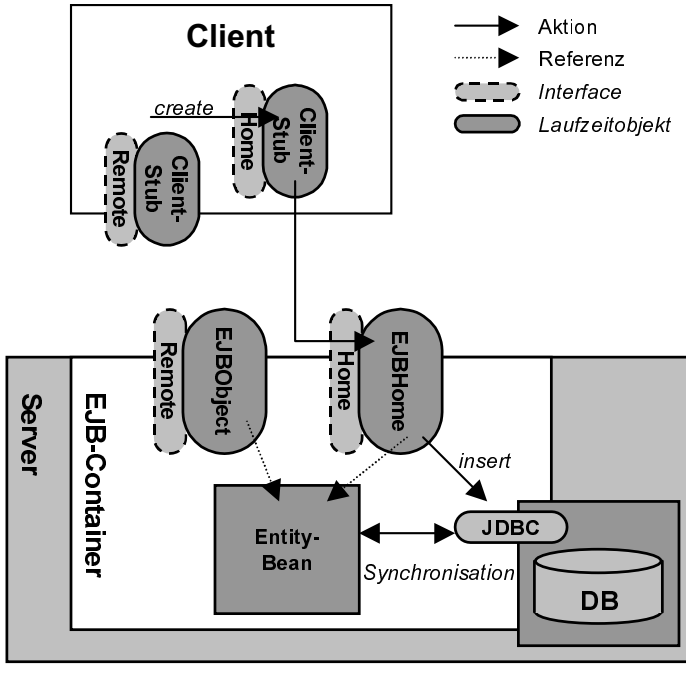

Abbildung 3.6: Erzeugen einer neuen Bean über das Home-Interface

Von diesem Zeitpunkt an kann der Client die Funktionalität der Bean nutzen, indem er die Methoden des Remote-Interface aufruft. Er kann beispielsweise den Kontostand um einen Betrag von hundert Einheiten erhöhen:

```
ba.increaseBalance(100.0);
```

Der Aufruf der Methode *increaseBalance(float)* am Client-Stub geht an die *EJBObject*-Instanz auf dem Server über. Von dort aus wird sie wiederum an die Bean-Instanz delegiert, was letztlich die Änderung der Daten unter Einbeziehung des Transaktionsmonitors zur Folge hat (für diese Methode wurde im Deployment-Deskriptor festgelegt, dass sie in einer Transaktion stattfinden muss).

Für den Client unterscheidet sich die Benutzung einer Enterprise-Bean von einem herkömmlichen, im selben Prozess befindlichen Java-Objekt nicht wesentlich, obwohl er mit einer transaktionsgesicherten Komponente auf einem entfernten Rechner kommu-

niziert. Was im Code des Clients mit einer einzeiligen Anweisung beginnt, kann auf dem Server sehr komplexe Aktionen auslösen.

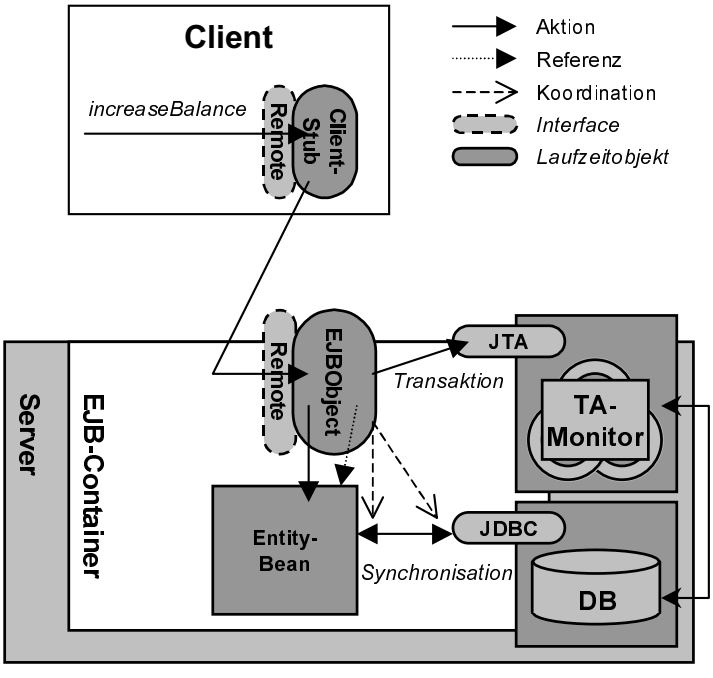

Abbildung 3.7: Erzeugen einer neuen Bean über das Home-Interface

Die Vorgehensweise bei der Verwendung einer Enterprise-Bean über das Local-Interface ist aus Sicht des Clients analog zu der Verwendung einer Enterprise-Bean über das Remote-Interface. Ebenso analog sind die Abläufe in den *EJBHome*- und *EJBObject*-Klassen.

Die in diesem Abschnitt dargestellten Sachverhalte treffen im Wesentlichen auf Session- und Entity-Beans zu. Message-Driven-Beans unterscheiden sich von den anderen beiden Typen (wie bereits erwähnt) dadurch, dass sie nicht direkt von einem Client angesprochen werden können. Sie können nur indirekt über das asynchrone Versenden einer Nachricht angesprochen werden. Eine Message-Driven-Bean ist weniger komplex wie eine Session- oder Entity-Bean und auch für den Container leichter zu handhaben, da der Messaging-Dienst den Großteil der Arbeit übernimmt. Kapitel 6 wird sich im Detail damit beschäftigen und die Unterschiede zu den anderen beiden Bean-Typen deutlich machen.

3.8 Was eine Enterprise-Bean nicht darf

Die Spezifikation der Enterprise JavaBeans ist gegenüber dem Entwickler einer Enterprise-Bean in Bezug auf die Nutzung bestimmter Schnittstellen sehr restriktiv. Die wichtigsten Einschränkungen sollen in diesem Abschnitt vorgestellt werden. Interessant ist vor allem die Frage nach dem Sinn derartiger Verbote. Wir werden diese Frage ergründen, nachdem wir die wichtigsten Restriktionen zunächst kurz vorstellen (die komplette Liste von Einschränkungen finden Sie in [Sun Microsystems, 2001]).

- Eine Enterprise-Bean darf keine statischen Variablen benutzen. Statische Konstanten sind dagegen erlaubt.
- Eine Enterprise-Bean darf keine Thread-Synchronisationsmechanismen benutzen.
- Eine Enterprise-Bean darf keine Funktionalität des AWT (Abstract Windowing Toolkit) benutzen, um Ausgaben über grafische Benutzeroberflächen zu machen oder Eingaben von der Tastatur zu lesen.
- Eine Enterprise-Bean darf keine Klassen aus *java.io* benutzen, um auf Dateien oder Verzeichnisse im Dateisystem zuzugreifen.
- Eine Enterprise-Bean darf nicht an einem Netzwerk-Socket horchen, sie darf keine Verbindungen an einem Netzwerk-Socket akzeptieren und sie darf keinen Socket für Multicast benutzen.
- Eine Enterprise-Bean darf nicht durch Introspection oder Reflection versuchen, an Informationen von Klassen und Instanzen zu gelangen, die ihr durch die Java-Sicherheitspolitik verborgen bleiben sollen.
- Eine Enterprise-Bean darf keinen Class-Loader erzeugen; sie darf keinen Class-Loader benutzen; sie darf den Kontext des Class-Loaders nicht verändern; sie darf keinen Security-Manager setzen; sie darf keinen neuen Security-Manager erzeugen; sie darf die Java Virtuelle Maschine nicht anhalten und sie darf den Standardeingabe-, den Standardausgabe- und den Standardfehlerstream nicht verändern.
- Eine Enterprise-Bean darf keine Objekte der Klassen *Policy*, *Security*, *Provider*, *Signer* oder *Identity* aus dem *java.security*-Package benutzen oder versuchen, deren Werte zu verändern.
- Eine Enterprise-Bean darf keine Socket-Factory setzen, die durch die Klassen *ServerSocket* und *Socket* benutzt wird. Gleiches gilt für die Stream-Handler-Factory der *URL*-Klasse.
- Eine Enterprise-Bean darf keine Threads benutzen. Sie darf sie weder starten noch stoppen.
- Eine Enterprise-Bean darf nicht direkt Dateideskriptoren lesen oder schreiben.

- Eine Enterprise-Bean darf keine nativen Bibliotheken laden.
- Eine Enterprise-Bean darf niemals *this* als Argument bei einem Methodenaufruf übergeben oder *this* als Rückgabewert für einen Methodenaufruf liefern.

Eine Enterprise-Bean zu programmieren heißt, serverseitige Logik auf einem relativ hohen Abstraktionsniveau zu entwickeln. Die Spezifikation der Enterprise JavaBeans beschreibt eine Komponentenarchitekur für verteilte Anwendungen. Durch das Komponentenmodell will die Spezifikation klare Trennlinien zwischen bestimmten Verantwortlichkeiten ziehen. Der Server und der EJB-Container sind für die systemnahe Funktionalität zuständig. Die Bean nutzt diese Infrastruktur als Laufzeitumgebung, muss sich also um eine systemnahe Funktionalität nicht kümmern. Die Aufgabe der Bean ist es, sich auf die Logik für unternehmensbezogene Abläufe zu konzentrieren. Um diese Aufgabe erfüllen zu können, soll sie die Dienste nutzen, die ihr durch den EJB-Container zur Verfügung gestellt werden, keine anderen. Durch die Restriktionen, die die Spezifikation den Enterprise-Beans auferlegt, sollen Konflikte zwischen dem EJB-Container und den Enterprise-Beans vermieden werden.

3.9 EJB-Rollenverteilung

Die Spezifikation der Enterprise JavaBeans teilt die Entwicklungsverantwortlichkeiten auf verschiedene Rollen auf. Dem liegt der Gedanke zugrunde, eine gewisse Abstraktion im EJB-Modell zu erreichen, um durch die Zuteilung von konkreten Aufgaben an bestimmte Expertengruppen Diversifizierung zu fördern und Synergieeffekte zu erzielen. Oder anders ausgedrückt: Jeder soll das entwickeln, was er am besten kann. Abbildung 3.8 zeigt einen Überblick über die verschiedenen Rollen und deren Zusammenspiel.

Server-Provider

Der Server-Provider ist für die Bereitstellung der grundlegenden Serverfunktionen verantwortlich. Er hat dafür zu sorgen, dass unterschiedlichen Containern (v.a. EJB-Containern) eine stabile Laufzeitumgebung zur Verfügung steht. Dazu gehören – wie in dem vorangegangenen Kapitel bereits erwähnt – unter anderem die Netzwerkanbindung, das Thread- und Prozessmanagement, die Skalierung (Clusterung) sowie das Ressourcenmanagement.

Container-Provider

Der Container-Provider setzt auf den Schnittstellen des Server-Providers auf und bietet den Komponenten (Enterprise-Beans) eine komfortable Laufzeitumgebung. Die Implementierung des Container-Providers muss den in diesem Kapitel vorgestellten Sachverhalten entsprechen. Er hat dafür zu sorgen, dass die Zugriffe auf eine Enter-

prise-Bean ausschließlich über den Container erfolgen, ebenso wie die Kommunikation der Enterprise-Bean mit ihrer Umwelt. Der Container-Provider stellt primär Möglichkeiten für die Persistenz der Komponenten, Mechanismen für die Abwicklung transaktionsorientierter Vorgänge, Security-Features, Ressource-Pooling und Unterstützung für die Versionierung von Komponenten zur Verfügung (wobei letzteres Feature durch die Spezifikation nicht näher definiert wird). Um Beans im Container installieren zu können, muss der Container-Provider Tools zur Verfügung stellen, die es dem Bean-Deployer (siehe unten) ermöglichen, den notwendigen Schnittstellencode zu erzeugen (EJBHome und EJBObject). Für den Systemadministrator muss der Container-Provider Tools für das Monitoring des Containers und der Beans zur Verfügung stellen.

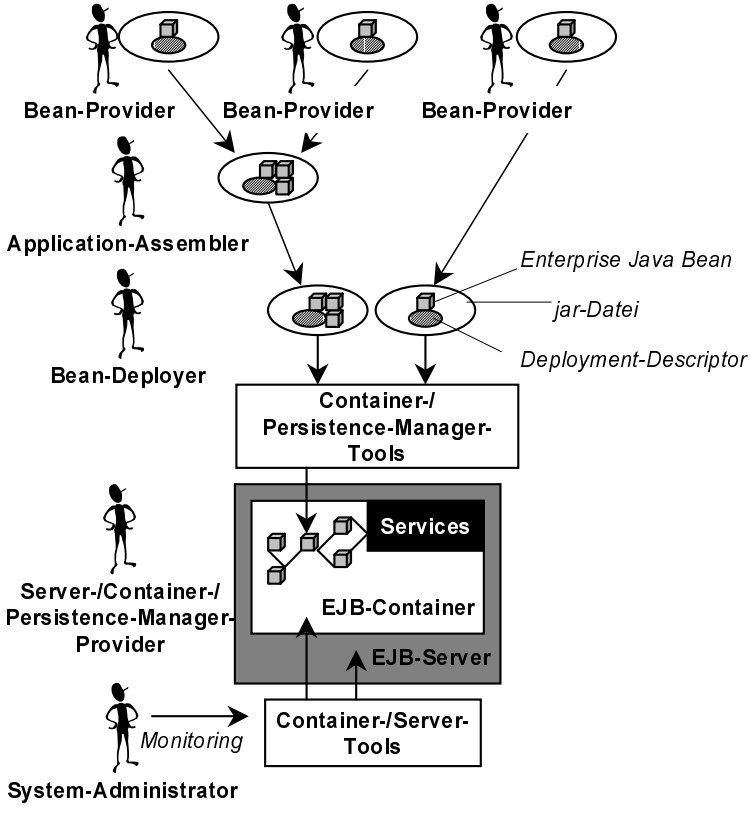

Abbildung 3.8: EJB-Rollenverteilung

Persistence-Manager-Provider

Der Persistence-Manager-Provider muss Tools zur Verfügung stellen, mit deren Hilfe der für die Persistenz einer container-managed Entity-Bean notwendige Code generiert werden kann. Die Tools werden bei der Installation einer Enterprise-Bean in einem bestimmten Container benutzt. Der Persistence-Manager-Provider ist typischerweise ein Spezialist im Bereich von Datenbanken. Er ist dafür zuständig, dass

- der Zustand einer Entity-Bean mit container-managed Persistence korrekt in der Datenbank gespeichert wird,
- die referenzielle Integrität der Entity-Bean mit anderen Entity-Beans, mit der sie in Beziehung steht, gewährleistet ist.

Da der EJB-Container die Persistenz einer Entity-Bean steuert (d.h., er bestimmt z.B., wann der Zustand gespeichert werden soll), ist eine Schnittstelle zwischen Container und Persistence-Manager erforderlich. Die Spezifikation überlässt es dem Container- und dem Persistence-Manager-Provider, eine solche Schnittstelle zu definieren.

Bean-Provider

Diese Rolle ist denjenigen Entwicklern zugedacht, die die eigentliche Geschäftslogik implementieren. Sie verpacken ihr Anwendungswissen in Komponenten, durch die dieses Wissen wiederverwendbar wird. Aufbauend auf den Produkten des Server- und Container-Providers wird der Bean-Entwickler vollständig von der Entwicklung grundlegender Servertechniken (Multithreading, Netzwerkanbindung, Transaktionen etc.) entlastet. Dadurch kann er sich voll und ganz seiner eigentlichen Aufgabe widmen. Zusammen mit seiner Komponente liefert der Bean-Provider den Deployment-Deskriptor, der Informationen über die Komponente selbst sowie über ihre externen Abhängigkeiten enthält (z.B. auf welche Serverdienste die Komponente angewiesen ist). Der Deployment-Deskriptor (ein ausführliches Beispiel wurde in Abschnitt 3.4.2 vorgestellt) liefert alle Informationen, die der Bean-Deployer (siehe unten) braucht, um die Komponente in einem Server zu installieren. Auch der Application-Assembler (siehe unten) benötigt diese Informationen, um Anwendungen oder Teilmodule einer Anwendung aus verschiedenen Komponenten zusammenstellen zu können. Das Ergebnis der Arbeit eines Bean-Providers ist eine Datei im jar-Format (jar steht für Java-Archiv), die die Bean-Klasse(n) und die Interfaces der Bean(s) sowie den Deployment-Deskriptor enthält.

Application-Assembler

Letzten Endes ist der Application-Assembler dafür verantwortlich, die zur Verfügung stehenden Funktionalitäten der im Container installierten Beans zu Anwendungen zu verknüpfen. Hierzu kann die Entwicklung der Client-Applikationen und die damit

verbundene Steuerung des Nachrichtenaustauschs mit den Enterprise-Beans gehören. Der Application-Assembler kann aber ebenso mehrere (kleinere) Komponenten, die von Bean-Providern geliefert wurden, zu einer neuen (größeren) Komponente zusammenfassen. Der Application-Assembler kann dieses Konglomerat durch eigene Enterprise-Beans ergänzen, deren Aufgabe beispielsweise in der Koppelung der anderen Enterprise-Beans besteht. Die dadurch entstehenden »Superkomponenten" repräsentieren bereits Teilaspekte einer konkreten Anwendung.

Seine Arbeit dokumentiert der Application-Assembler ebenso wie der Bean-Provider im Deployment-Deskriptor. Damit erkennt der Bean-Deployer (vgl. nächsten Abschnitt), wie er die Komponenten zu installieren hat, damit aus dem Zusammenspiel einzelner Bausteine eine komplette Anwendung entsteht. Daneben kann der Application-Assembler für den Bean-Deployer auch Anweisungen und Informationen bezüglich der Benutzerschnittstelle oder der Abhängigkeiten zu Nicht-EJB-Komponenten im Deployment-Deskriptor vorsehen.

Bean-Deployer

Die für die Lösung einer bestimmten Geschäftsproblematik ausgewählten Komponenten in einem Container zu installieren, ist die Aufgabe des Bean-Deployers. Dazu gehört vor allem die Bedienung der Tools des Container- und des Persistence-Manager-Providers sowie die richtige Parametrisierung der zu installierenden Komponenten. Diese Tätigkeit setzt ein sehr umfangreiches Wissen über den Anwendungs- und Systemkontext sowie über die internen Verknüpfungen des Systems voraus. Insbesondere muss der Deployer die im Deployment-Deskriptor definierten externen Abhängigkeiten auflösen (indem er z. B. die Existenz geforderter Dienste oder verwendeter Enterprise-Beans sicherstellt) und die im Deployment-Deskriptor enthaltenen Anweisungen des Application-Assemblers berücksichtigen.

Systemadministrator

Der Systemadministrator überwacht den EJB-Server mit Hilfe der vom Hersteller gelieferten Tools zur Laufzeit und sorgt für die zum Betrieb eines EJB-Servers notwendige Infrastruktur (z. B. für ein funktionstüchtiges Netzwerk).

Die EJB-Rollenverteilung stellt ein idealtypisches Szenario dar und die Umsetzung der EJB-Spezifikation in der Praxis wird zeigen, ob es in dieser Form eingehalten werden kann. So werden heute Server und Container in einem Produkt angeboten. Die Java-2-Plattform, Enterprise Edition (J2EE), sieht für einen Applikationsserver ohnehin vor, dass er verschiedene Container anbieten soll (mindestens einen EJB-Container und einen Web-Container). Die Spezifikation der Enterprise JavaBeans geht mittlerweile davon aus, dass Server und Container immer vom selben Hersteller angeboten werden und untereinander nicht ausgetauscht werden. Gleiches ist für den Persistence-Mana-

ger anzunehmen. Solange die Spezifikation kein Protokoll zwischen EJB-Container und Persistence-Manager definiert, werden beide in einem Produkt angeboten werden.

Dieses Rollenspiel passt (in der Theorie) jedoch hervorragend zum Ansatz komponentenorientierter Software. Zumindest wenn man davon ausgeht, dass es sich jeweils um verschiedene Personen handelt. So weiß ein Container-Provider (der Container ist schließlich auch eine Art Komponente) bei der Entwicklung nicht, welche Beans später in diesem Container eingesetzt werden. Ebensowenig weiß ein Bean-Provider zum Zeitpunkt der Entwicklung, für welche Anwendungszwecke seine Komponente später einmal benutzt wird. Er verpackt lediglich eine in sich geschlossene Funktionalität in eine Komponente mit dem Ziel der optimalen Wiederverwendbarkeit. Sofern durch den Bean-Deployer die richtigen Komponenten installiert und parametrisiert wurden, ist es für den Application-Assembler ein Leichtes, die angebotene Funktionalität zu Anwendungen zu verbinden bzw. aus den verfügbaren Komponenten Teilmodule zu bilden. Die Arbeit des Application-Assemblers ist im Übrigen zeitlich nicht an die Arbeit des Bean-Deployers gebunden. Der Application-Assembler kann durch den Deployer bereits installierte Komponenten benutzen oder kann dem Deployer von ihm zusammengestellte »Superkomponenten« zur Installation übergeben.

Die Rollen der Server- und Container-Provider sind den Systemspezialisten zugedacht, die die stabile Basis liefern. Für einen Anwendungsentwickler (wie er etwa im einleitenden Kapitel *Motivation* beschrieben wurde) ergeben sich verschiedene Ansätze. Er könnte zunächst in die Rolle des Bean-Providers schlüpfen, indem er sein Fachwissen in Beans verpackt und in einem zugekauften Applikationsserver, der einen EJB-Container anbietet, installiert. Stattdessen könnte er auch im Katalog eines Softwarehauses blättern und nach Komponenten suchen, die ihm bereits eine Lösung für sein(e) Problem(e) anbieten. Seine Aufgabe würde sich dann darauf beschränken, die gekauften Beans im Server zu installieren und richtig zu parametrisieren (was der Rolle des Bean-Deployers entspricht). Im nächsten Schritt könnte er (oder ein Entwicklerkollege) unter Zuhilfenahme der installierten Beans eine Anwendung schreiben, die die Interaktion mit dem Benutzer und die Kommunikation mit den Komponenten übernimmt (Application-Assembler, beschränkt auf die Entwicklung des Client-Szenarios).

Eines ist für den Anwendungsentwickler (vor allem in der Rolle des Bean-Providers) in jedem Fall klar: Er kann sich voll und ganz auf die Lösung seiner Probleme konzentrieren. Technische Probleme wie die Implementierung eines stabilen, leistungsfähigen und skalierbaren Servers werden ihm durch den Server- und den Container-Provider abgenommen.

Dieses Buch wird sich im weiteren Verlauf im Wesentlichen auf die Rollen des Bean-Providers, des Bean-Deployers und des Application-Assemblers beschränken.

3.10 Sichtweisen

Nachdem die Architektur und die wesentlichen Merkmale und Konzepte der Spezifikation der Enterprise JavaBeans bekannt sind, wollen wir abschließend die unterschiedlichen Blickwinkel auf die Architektur verdeutlichen.

3.10.1 EJB aus der Sicht der Applikationsentwicklung

Es ist interessant, die Enterprise JavaBeans in Hinblick auf die Applikationsentwicklung zu betrachten. Hierzu soll auf das Szenario in Kapitel 1 zurückgegriffen werden. Der dort beschriebene Anwendungsentwickler könnte als Basis für den Prototyp, den er entwickeln muss, Enterprise JavaBeans benutzen. Dazu würde er einen J2EE-kompatiblen Server inklusive EJB-Container eines beliebigen Herstellers benutzen und den Prototyp in Form von Enterprise-Beans entwickeln. Wir wollen diskutieren, ob sich die Randprobleme, die sich durch das Businessproblem ergeben, durch den Einsatz von Enterprise JavaBeans als Basistechnologie neutralisieren lassen. Wir wollen untersuchen, ob sich der Applikationsentwickler voll und ganz auf die Probleme seines Anwendungsbereichs konzentrieren kann. Dazu soll nochmals eine Abbildung aus Kapitel 1 ins Gedächtnis gerufen werden (siehe Abbildung 3.9).

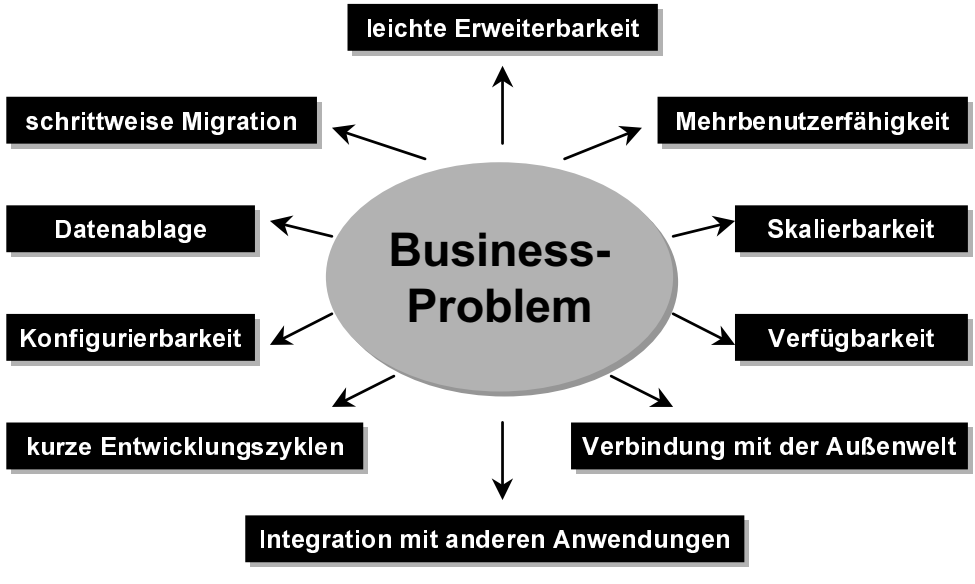

Abbildung 3.9: Randprobleme einer Businessproblematik

Die Anforderung der *Mehrbenutzerfähigkeit* ist durch die client-server-orientierte Architektur ohne Frage gegeben. Auch die Sicherheitseigenschaften und der Transaktionsdienst von Enterprise JavaBeans tragen ihren Teil dazu bei, einen reibungslosen Ablauf im Mehrbenutzerbetrieb sicherzustellen.

Die Fähigkeit zur *Skalierbarkeit* ist dadurch gegeben, dass Anwendungen, die als Enterprise-Beans oder als Aggregate von Beans realisiert sind, auf mehrere Server verteilt werden können. Die Server können beispielsweise die gleiche Datenbank als Backend-System benutzen. So ist die Last der Clientanfragen in Abhängigkeit von der benutzten Anwendung auf mehrere Server verteilt, während sich die Datenbasis an zentraler Stelle befindet. Durch die Ortstransparenz von Enterprise-Beans mit Remote-Interface (die über einen zentralen Namens- und Verzeichnisdienst gewährleistet wird) ist es möglich,

- einzelne Anwendungen eines Applikationssystems auf mehrere Server zu verteilen,
- einzelne Beans von einem Server auf einen anderen zu verlagern,
- zusätzliche Server einzurichten und Beans bzw. Aggregate von Beans dorthin zu verlagern.

Darüber hinaus bieten viele Applikationsserver Lastverteilungsmechanismen an. Hierbei kümmert sich ein Cluster aus Applikationsservern (unabhängig von der Anwendung, die der Client bedient) um die optimale Bedienung der Clientanfragen. Dieses Feature ist jedoch herstellerabhängig und nicht durch die Architektur der Enterprise JavaBeans bedingt.

Auch beim Thema *Verfügbarkeit* kommt die Ortstransparenz der Enterprise-Bean-Komponenten, die durch den zentralen Namens- und Verzeichnisdienst gewährleistet wird, zum Tragen. Fällt ein Applikationsserver aus, können die dort installierten Beans in einem anderen Server installiert werden. Die für die Installation notwendigen Informationen sind bereits in den Deployment-Deskriptoren der Enterprise-Beans enthalten und können in der Regel unverändert übernommen werden. Durch eine Änderung der Konfiguration im Namens- und Verzeichnisdienst können sie den Clients auf einem anderen Server zur Verfügung gestellt werden, ohne dass die Konfiguration des Clients geändert werden muss.

Da Enterprise JavaBeans immer enger in den Kontext der Java-2-Plattform, Enterprise Edition, eingebunden werden, ist die *Verbindung mit der Außenwelt* ein unkritisches Thema. Mit der Unterstützung durch einen Web-Container (vgl. Abbildung 3.2) sind zumindest die technologischen und architektonischen Grundvoraussetzungen gegeben, um eine Verbindung zur Außenwelt zu schaffen. Im letzten Kapitel werden wir uns diesem Thema etwas detaillierter widmen.

Sind alle Anwendungen eines Applikationssystems auf der Basis von Enterprise JavaBeans realisiert, so ist die *Integrierbarkeit der Anwendungen* untereinander sehr hoch. Der Austausch von Daten innerhalb der Anwendung ist in diesem Fall einfach (noch dazu wenn eine zentrale Datenbank benutzt wird). Enterprise JavaBeans kann auch dazu benutzt werden, um Anwendungen verschiedener Systeme zu integrieren. Über den Messaging-Dienst, den der EJB-Container seit Version 2.0 integrieren muss, ist sogar eine dynamische Kopplung von Anwendungen und Systemen möglich. Ein Messaging-Dienst stellt verschiedene Kommunikationskanäle zur Verfügung, über die eine n-zu-n-Kommunikation mit dynamisch wechselnden Partnern möglich ist.

Durch den Einsatz spezialisierter EJB-Container, die statt eines Datenbanksystems ein anderes Applikationssystem als Persistenzsystem benutzen, können diese Systeme in EJB-basierte Systeme integriert werden. Unter Umständen kann man sich auch vorstellen, Enterprise-Beans als Wrapper und Schnittstelle zu einfachen Systemen einzusetzen (sofern das im Rahmen der Programmierrestriktionen möglich ist).

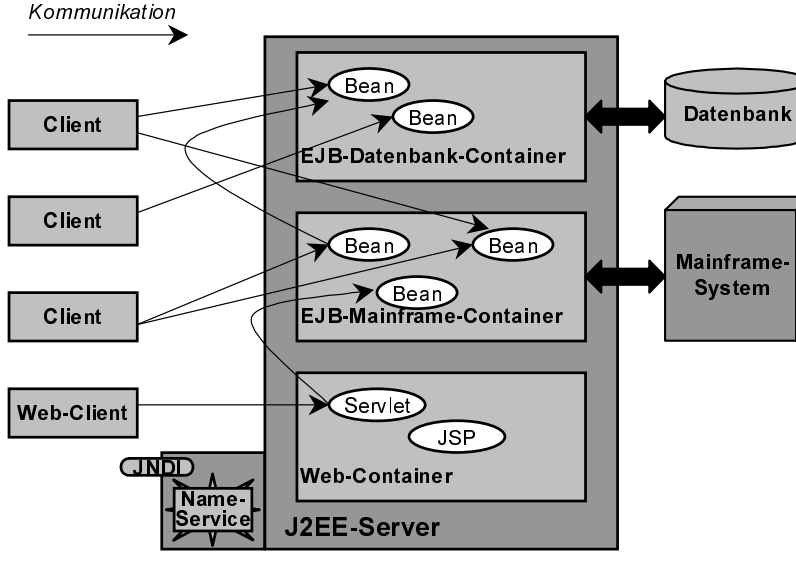

Abbildung 3.10: Enterprise-Java-Bean-Szenario

Anwendungen setzen sich aus mehreren Enterprise-Beans zusammen, von denen jede für sich eine gewisse Funktionalität kapselt. Diese durch das Komponentenparadigma vorgegebene Granularität ermöglicht es, die Entwicklung eines Applikationssystems nach vielen Gesichtspunkten zu strukturieren. Jede Komponente kann ein Teilprojekt sein. Der Umfang solcher Teilprojekte ist im Vergleich zum Gesamtprojekt wesentlich überschaubarer.

Die Schnittstellen nach außen sind pro Komponente (und damit pro Teilprojekt) fest definiert. Dadurch wird tendenziell ein Beitrag zur *Verkürzung der Entwicklungszyklen* geleistet. Nicht zuletzt trägt die Architektur der Enterprise JavaBeans aktiv dazu bei, da dem Anwendungsentwickler (Bean-Entwickler) die Entwicklung der systemtechnischen Funktionalität durch den Applikationsserver abgenommen wird. Durch das Vorhandensein einer einheitlichen Plattform kann diese systemtechnische Funktionalität wiederverwendet werden und der Bean-Entwickler kann sich ganz auf die Lösung seiner anwendungsbezogenen Problemstellungen konzentrieren.

Die *Konfigurierbarkeit* eines EJB-basierten Systems ist im Wesentlichen durch den Deployment-Deskriptor gegeben. Durch ihn lassen sich Beans zum Zeitpunkt der Installation in einem Server in vielfältiger Weise anpassen. Auch das Verhalten einer Enterprise-Bean zur Laufzeit kann über die Einstellungen im Deployment-Deskriptor beeinflusst werden. Über das Bean-Environment können Parameter gesetzt werden, die die Bean zur Laufzeit auswerten kann.

Ein EJB-basiertes System kann auch durch den Austausch von Bean-Klassen konfiguriert werden. Die neuen Bean-Klassen bieten zwar die gleiche Schnittstelle an, d.h. sie benutzen die gleichen Home- und Remote-Interfaces wie die alten Bean-Klassen, weisen aber eine andere Implementierung auf. Dem Client gegenüber treten die Beans mit der geänderten Implementierung unter dem gleichen Namen im JNDI auf wie die alten. Für den Client spielt der Austausch der Implementierungen keine Rolle, sofern der Vertrag (die versprochene Funktionalität der Schnittstelle) eingehalten wird.

Im Falle der container-managed Persistence bietet Enterprise JavaBeans ein Konzept zur transparenten, automatischen *Datenablage*. Die Enterprise-Bean überlässt die Datenspeicherung dem EJB-Container und dem Persistence-Manager. Sie muss sich nicht mit den Techniken zur Datenspeicherung auseinandersetzen. Eine solche Bean kann problemlos in anderen EJB-Containern eingesetzt werden.

Durch den Einsatz spezialisierter EJB-Container (wie in Abbildung 3.10 dargestellt) ist auch eine *schrittweise Migration* von Altsystemen möglich. Die Datenbestände sowie die Systeme selbst können weiterbenutzt werden. Für die Bean ist der EJB-Container damit nicht nur Laufzeitumgebung und Dienstanbieter, sondern zugleich die Schnittstelle zu einem Altsystem (die damit für die Bean weitgehend transparent ist). Nach der kompletten Übernahme der Datenbestände (z.B. in ein relationales Datenbanksystem) können die bereits entwickelten Beans durch die Installation in einem anderen EJB-Container weiterverwendet werden.

EJB-basierte Systeme in ihrer *Funktionalität zu erweitern* ist durch die Installation neuer Enterprise-Beans problemlos möglich. Neue Beans können unter Umständen auf die Funktionalität bereits vorhandener Beans zurückgreifen.

Aus der Sicht des Applikationsentwicklers in dem in Kapitel 1 beschriebenen Szenario dürften sich alle auftretenden Randprobleme durch den Einsatz von Enterprise JavaBeans neutralisieren lassen. Für die Lösung seines Problems wäre EJB der richtige Ansatz.

3.10.2 EJB aus der Sicht des Komponentenparadigmas

Ein weiterer Gesichtspunkt, unter dem Enterprise JavaBeans betrachtet werden kann, ist die Sicht des Komponentenparadigmas. In den vorangegangenen Kapiteln wurden einige grundlegende Eckpunkte aus diesem Bereich vorgestellt. Vor diesem Hintergrund stellen sich zwei Fragen:

- Sind Enterprise-Beans echte Komponenten?
- Erfüllt Enterprise JavaBeans die Anforderungen an eine Komponentenarchitektur?

Um diese Fragen zu erörtern, sollen die Eigenschaften von Enterprise-Beans und der EJB-Architektur an den in Kapitel 2 vorgestellten Voraussetzungen gemessen werden.

Enterprise-Beans sind hinsichtlich der in Kapitel 2.3 vorgestellten Definition echte Komponenten. Das wesentliche Merkmal der standardisierten Schnittstelle ist in Form des Home- und des Remote- bzw. des Local-Home und des Local-Interface gegeben. Standardisiert ist die Schnittstelle durch die Vorgaben des Komponentenmodells der Enterprise-JavaBeans-Spezifikation. Die Schnittstelle stellt eine Art Vertrag dar, den sich die Enterprise-Bean einzuhalten verpflichtet. Diese Schnittstelle ist von der Implementierung unabhängig. Eine Bean kann in einem Stück erstellt und gepflegt werden. Ob sie eine in sich geschlossene Aufgabe repräsentiert, hängt von ihrem Entwickler ab. Dieses Wesensmerkmal einer Komponente kann durch eine Spezifikation nicht erzwungen werden, sollte aber dem Entwickler einer Enterprise-Bean als Maßgabe dienen.

Um festzustellen, ob die Architektur der Enterprise JavaBeans den Anforderungen des Komponentenparadigmas gerecht wird, wollen wir EJB an den wesentlichen Merkmalen, die ebenfalls in Abschnitt 2.3 bereits vorgestellt wurden, messen:

- *Unabhängigkeit von der Umgebung:* Enterprise JavaBeans basiert vollständig auf der Programmiersprache Java. Enterprise-Beans können nur in einer Java-Umgebung eingesetzt werden. Sie können jedoch über CORBA aus anderen Umgebungen heraus benutzt werden. Die Spezifikation in der Version 2.0 sieht RMI-IIOP als Kommunikationsprotokoll vor, was die Kooperationsfähigkeit mit CORBA sicherstellen soll.
- *Ortstransparenz:* Der Aufenthaltsort einer Komponente mit Remote-Interface ist bei Enterprise JavaBeans vollständig transparent, da die Enterprise-Beans über den Namens- und Verzeichnisdienst gefunden und über RMI auch auf entfernten Rech-

nern angesprochen werden können. Die Spezifikation verpflichtet einen Hersteller auch dazu, einen Namens- und Verzeichnisdienst in die Systemumgebung mit einzubeziehen und betont den verteilten Charakter von Enterprise-Beans. Die Ortstransparenz ist bei Enterprise-Beans mit Local-Interface nicht gegeben.

- *Trennung von Schnittstelle und Implementierung:* Diese Eigenschaft ist bei Enterprise JavaBeans durch die Konzepte des Home- und des Remote- bzw. des Local-Home- und des Local-Interface gegeben.

- *Selbstbeschreibende Schnittstellen:* Bei Enterprise JavaBeans ist es möglich, zur Laufzeit Informationen über die Remote-Schnittstelle(n) einer Komponente zu erfragen. Das Home-Objekt liefert über die Methode *getEJBMetaData()* ein Objekt des Typs *javax.ejb.EJBMetaData* (für entsprechende Details vgl. [Sun Microsystems, 2001]). Dieses Objekt gibt unter anderem Auskunft über das Home- und Remote-Interface einer Enterprise-Bean. Mit Hilfe der Java-Reflection-API (vergleiche [Eckel, 1998]) ist es möglich, die Interfaces zur Laufzeit zu untersuchen und dynamische Methodenaufrufe an Enterprise-Beans zu programmieren. Die Kombination aus dem Namens- und Verzeichnisdienst (JNDI), der Metadaten-Schnittstelle (*javax.ejb.EJBMetaData*) und der Java-Reflection-API bietet vergleichbare Möglichkeiten wie das Interface-Repository bei CORBA (vergleiche dazu [Orfali et al., 1998]). Es ist die Aufgabe des EJB-Containers aus den Angaben des Deployment-Deskriptors für eine Enterprise-Bean eine entsprechende Implementierungsklasse für die Metadaten-Schnittstelle zu generieren.

- *Problemlose sofortige Nutzbarkeit (Plug&Play):* Zum Installationszeitpunkt einer Enterprise-Bean müssen Implementierungsklassen für das Home- und Remote-Interface generiert und kompiliert werden. Durch diese Klassen wird eine Enterprise-Bean-Komponente (ohne dass sie dabei selbst verändert werden muss) für den EJB-Container eines bestimmten Herstellers benutzbar. Alle für den Installationsvorgang notwendigen Informationen sind in der Bean-Klasse selbst und im Deployment-Deskriptor vorhanden (der ein Bestandteil der Komponente ist). Die binäre Unabhängigkeit des Komponentencodes einer Enterprise-Bean ist insofern gegeben, als die Komponenten dem Binärstandard entsprechen, der durch die Programmiersprache Java definiert wird. Diese Umstände machen eine Enterprise-Bean-Komponente ohne Probleme in jedem EJB-Container benutzbar.

- *Integrations- und Kompositionsfähigkeit:* Diesem Punkt wird durch eine bestimmte Rolle im Vorgehensmodell, dem *Application-Assembler* (siehe Abschnitt 3.8, EJB-Rollenverteilung), und einen besonderen Abschnitt im Deployment-Deskriptor Rechnung getragen. Die Komposition von Enterprise-Beans zu Aggregaten ist ausdrücklich vorgesehen.

Enterprise JavaBeans kann mit gutem Gewissen als Komponentenarchitektur bezeichnet werden. Man darf gespannt darauf sein, welche Erweiterungen und Verbesserungen zukünftige Versionen bieten werden.

3.10.3 EJB aus Sicht der Unternehmung

Ein letzter Gesichtspunkt wäre die Betrachtung von Enterprise JavaBeans aus Sicht der Unternehmung. In Kapitel 2 wurden Kriterien definiert, die eine Basistechnologie aus dieser Sicht erfüllen sollte: Wirtschaftlichkeit, Sicherheit und Bedarfsorientierung. Nun ist die objektive Bewertung einer Technologie anhand solch globaler Messgrößen schwer. Grundsätzlich lässt sich jedoch festhalten, dass Enterprise JavaBeans aus Gründen der Wirtschaftlichkeit, so wie Kapitel 2 sie definiert, durchaus für Unternehmen interessant sind – insbesondere für diejenigen Unternehmen, die Anwendungsentwicklung betreiben (sei es für unternehmensinterne oder -externe Kunden).

Aus den obigen Abschnitten gehen die Vorteile einer solchen Technologie bereits hervor. Bei der Entwicklung von unternehmensbezogener Logik in Enterprise-Beans ist der Entwickler vollständig von systemtechnischen Problemenstellungen entlastet. Die technologische Basis wird von Spezialisten in diesem Gebiet entwickelt und entspricht einem (Quasi-)Standard. Die eigene Entwicklung kann zielorientierter auf die Probleme der jeweiligen Unternehmung ausgerichtet werden. Die komponentenorientierte Sichtweise ermöglicht eine höhere Granularität und macht dadurch die Entwicklung überschaubarer und besser beherrschbar. Die EJB-Technologie bietet alle Voraussetzungen, damit die Anwendungssysteme mit dem Unternehmen wachsen können (sowohl in Bezug auf erweiterte Funktionalität als auch auf eine wachsende Anzahl von Mitarbeitern).

In puncto Sicherheit wird die Unternehmung nicht nur durch die Sicherheitsmechanismen der Spezifikation und der Programmiersprache Java unterstützt. Da Enterprise JavaBeans in gewisser Weise einen Standard definiert, der bereits von vielen Herstellern unterstützt wird, begibt man sich in relativ geringe Abhängigkeiten. Wünschenswert wäre jedoch die offizielle Standardisierung der Spezifikation der Enterprise JavaBeans durch ein Normierungsgremium. Die Investition in diese Technologie dürfte sich dennoch auf lange Zeit hin bezahlt machen.

Die Spezifikation definiert viele Sachverhalte, die der Ausfallsicherheit von EJB-basierten Systemen zugute kommen. Ein Beispiel hierfür sind die strengen Restriktionen bei der Bean-Entwicklung. Sie sollen unter anderem helfen, instabile Zustände des Systems zu vermeiden. Ein anderes Beispiel ist die Trennung von Systemfunktionalität und unternehmensbezogener Funktionalität. Förderlich für die Zuverlässigkeit und Sicherheit ist, dass die Systemfunktionalität von einem spezialisierten Hersteller entwickelt wird. Ein breiter Einsatz des Systems bei vielen Kunden mit einem verantwortlichen Hersteller sollte schneller zu hoher Qualität führen als eine Eigenentwicklung.

Was die Bedarfsorientierung angeht, so zielt die Spezifikation genau auf solche Unternehmen, die für sich selbst oder für andere Unternehmen Anwendungen entwickeln. In der Regel sind die Entwickler solcher Unternehmen Spezialisten in bestimmten Anwendungsbereichen, nicht aber Spezialisten im Bereich der Systementwicklung. Oft endet dies in Anwendungen, die zwar die fachlichen Probleme abdecken, aber durch Mängel im Bereich der technologischen Basis für Unzufriedenheit sorgen. Mit dem Modell der Enterprise JavaBeans wird genau dieser Problematik begegnet. Der Bedarf nach einer stabilen Basisarchitektur, die ein komfortables Komponentenmodell bietet, in das die Anwendungslogik eingebettet werden kann, dürfte in vielen Bereichen und in vielen Unternehmen gegeben sein. Gerade weil die in Unternehmen eingesetzten Applikationen immer komplexer werden, besteht zunehmender Bedarf an stabilen, sicheren und flexiblen Basisarchitekturen.

4 Session-Beans

4.1 Einleitung

Eine Session-Bean ist ein Geschäftsobjekt, da sie typischerweise die Logik eines Geschäftsprozesses realisiert. In der Praxis könnte die Funktionalität einer Session-Bean das Aufstellen einer Bilanz in einem Finanzbuchhaltungssystem, die Durchführung einer Überweisung einer Banken-Software oder die Verwaltung eines Materiallagers sein.

Logisch wird eine Session-Bean als serverseitige Erweiterung des Clientprogramms betrachtet. Ein Client hat über den EJB-Container den exklusiven Zugriff auf eine bestimmte Session-Bean-Instanz und kann deren Funktionalität nutzen. Dabei umfasst der Begriff »Client« andere Beans auf dem Server und Clientprogramme auf einem Clientrechner. Für die Realisierung ihrer Funktionalität kann die Session-Bean andere Enterprise-Beans und die Services, die der EJB-Container zur Verfügung stellt, verwenden.

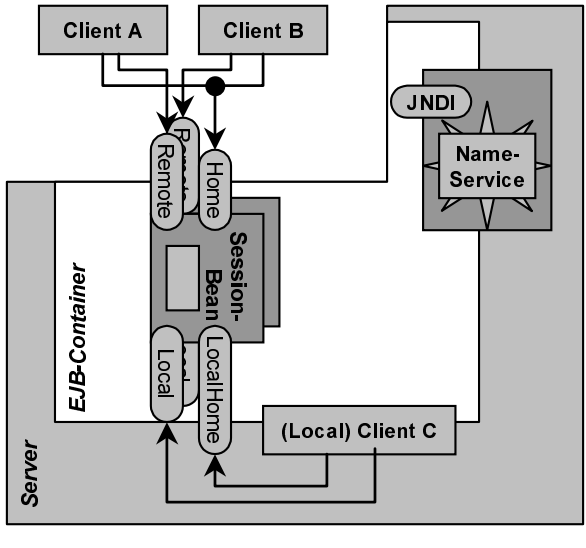

Abbildung 4.1: Überblick über Session-Beans

Während ein Client eine Session-Bean verwendet, kann er unterschiedliche Aktionen ausführen, die zu Zustandsänderungen, d.h. zu Änderungen der Daten von Enterprise-Beans führen. Man unterscheidet persistente (dauerhafte) und transiente (nicht dauerhafte) Zustandsänderungen:

- Eine persistente Zustandsänderung bleibt über mehrere Sitzungen des Clients hinweg bestehen. Veränderte Daten werden beispielsweise in einer Datenbank gespeichert. Session-Beans selbst kennen keinen persistenten Zustand, sondern verwenden Entity-Beans für die Speicherung persistenter Daten (siehe Kapitel 5, Entity-Beans) oder greifen beispielsweise direkt auf eine Datenbank zu.
- Eine transiente Zustandsänderung besteht nur für die Dauer einer Session. Der transiente Zustand beschreibt somit den Zustand der Session bzw. der Konversation zwischen Client und Server. Deshalb spricht man hier auch von einem *Conversational-State* (Zustand der Konversation). Der transiente Zustand wird teilweise vom Client und teilweise von der Session-Bean gespeichert. Deshalb wird die Session-Bean auch als logische Erweiterung des Clients betrachtet.

Es werden zwei Arten von Session-Beans unterschieden: Es gibt zustandslose (*stateless*) und zustandsbehaftete (*stateful*) Session-Beans. Im Gegensatz zu den zustandsbehafteten Session-Beans haben die zustandslosen Session-Beans keinen eigenen Conversational-State. Das bedeutet, dass keine Anwendungsmethode einer zustandslosen Session-Bean den Zustand der Bean-Instanz verändert.

Die beiden Arten sind stets voneinander zu trennen, da der EJB-Container sie bei der Verwaltung der Bean-Instanzen unterschiedlich behandelt. Dies wirkt sich auf die Programmierung der Methoden für das Zustandsmanagement der Bean-Instanzen aus, die der EJB-Container aufruft, um die Bean-Instanz zur Laufzeit über einen Zustandswechel zu informieren. In Abschnitt 4.2.4, *Lebenszyklus einer Session-Bean-Instanz*, werden wir auf die genauen Unterschiede eingehen.

Methoden von Session-Beans können in einer Transaktion ausgeführt werden. Dies ist davon abhängig, ob die Session-Bean explizite Transaktionen verwendet oder ob der EJB-Container durch die Konfiguration angewiesen wird, ein entsprechendes transaktionales Verhalten sicherzustellen. Eine Transaktion stellt sicher, dass die jeweilige Methode entweder erfolgreich ausgeführt oder vollständig rückgängig gemacht wird. So kommt es im Fehlerfall nicht zu inkonsistenten Zuständen. Da die Transaktionsthematik sehr komplex ist, wird sie vollständig in Kapitel 7, *Transaktionen*, behandelt.

4.2 Konzepte

4.2.1 Conversational-State

Eine Session-Bean bietet ihrem Client Dienstleistungen in Form von Methoden an. Der Client ruft im Regelfall mehrere Methoden auf, um sein Ziel zu erreichen. Ein Methodenaufruf kann schreibend auf einen verfügbaren Service zugreifen, beispielsweise die Daten in einer Datenbank verändern. Die veränderten Daten stehen allen Clients zur Verfügung und beeinflussen das Ergebnis der folgenden Methodenaufrufe bei unterschiedlichen Beans.

Ein Methodenaufruf kann auch den Zustand der Session-Bean-Instanz (d.h. ihre Instanzvariablen) selbst ändern. Die Session-Bean-Instanz ist dann zustandsbehaftet. Der Zustand einer Session-Bean-Instanz kann nur durch Methodenaufrufe an der jeweiligen Instanz beeinflusst werden.

Dieser Zustand einer Instanz einer zustandsbehafteten Session-Bean wird Conversational-State genannt. Die EJB-Spezifikation definiert den Conversational-State als den Zustand aller Attribute einer Session-Bean-Instanz einschließlich der transitiven Hülle über alle Objekte, die über Java-Referenzen von der Session-Bean-Instanz aus erreichbar sind. Der Conversational-State wird also durch alle Attribute und verwendeten Ressourcen der Session-Bean gebildet. In der Praxis besteht der Conversational-State beispielsweise aus Attributwerten, offenen Socket-Verbindungen, Datenbankverbindungen, Referenzen auf andere Beans und ähnliche Ressourcen.

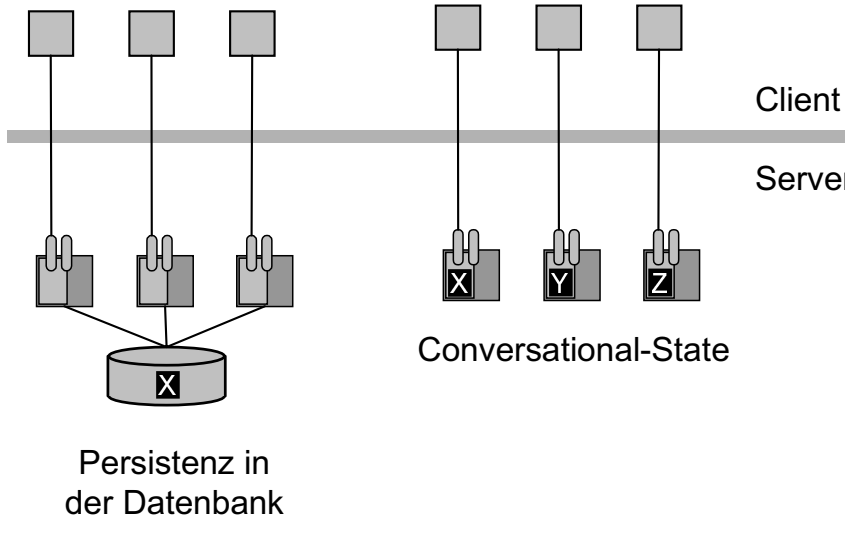

Abbildung 4.2: Conversational-State

Der Conversational-State bleibt nur für die Dauer einer Sitzung eines Clients erhalten und steht exklusiv nur diesem einen Client zur Verfügung. Aus diesem Grund wird die Session-Bean als logische, serverseitige Erweiterung des Clients verstanden.

Wenn eine Session-Bean-Instanz einen eigenen Zustand für genau einen Client speichert, dann wird auch für jeden Client eine eigene Bean-Instanz benötigt. Damit die Anzahl der Instanzen von zustandsbehafteten Session-Beans am Server nicht beliebig wächst, verwendet der EJB-Container eine Strategie zur zeitweisen Auslagerung der Instanzen aus dem Arbeitsspeicher. Die Auslagerung wird *Passivierung* genannt, das Zurückladen *Aktivierung*.

Bei der Passivierung schreibt der EJB-Container den Zustand einer zustandsbehafteten Session-Bean auf ein sekundäres Speichermedium (z. B. auf die Festplatte). Dafür wird die *Serialisierung* von Java-Objekten verwendet, die ein Bestandteil der Sprachdefinition von Java selbst ist. Dieser Mechanismus verwandelt Java-Objekte in einen Byte-Strom, der beispielsweise einfach in eine Datei geschrieben werden kann. Der umgekehrte Mechanismus heißt *Deserialisierung* und wird bei der Aktivierung eingesetzt. Dabei wird der Byte-Strom wieder in Java-Objekte im Arbeitsspeicher umgewandelt.

Mit Hilfe dieser Mechanismen (die im Übrigen nur auf Session-Beans mit einem Conversational-State angewendet werden) kann der EJB-Container mit einer nach oben hin begrenzten Anzahl von Session-Bean-Instanzen arbeiten, um die verfügbaren Ressourcen zu schonen. Werden mehr Bean-Instanzen benötigt, so können momentan nicht verwendete Instanzen ausgelagert werden. Wird die ausgelagerte Instanz wieder benötigt, so muss dafür zuerst eine andere Bean-Instanz aus dem Speicher ausgelagert werden. Dann wird die benötigte Bean-Instanz wieder in den Arbeitsspeicher geholt. Der EJB-Container verwendet hier also ein klassisches Caching-Verfahren. Als Strategie bietet sich die Auslagerung der am längsten nicht gebrauchten Bean-Instanz an (LRU). Die maximale Anzahl der Bean-Instanzen, mit denen der EJB-Container arbeitet, kann in der Regel konfiguriert werden.

Dieser Mechanismus birgt jedoch ein Problem: Nicht alle Java-Objekte können einfach serialisiert werden. Beispielsweise kann eine Socket- oder Datenbankverbindung nicht einfach auf Platte geschrieben werden. Die Sprachdefinition von Java verlangt von allen Java-Klassen, deren Objekte serialisierbar sein sollen, das Interface *java.io.Serializable* zu implementieren. Folgerichtig implementieren in der Java-Klassenbibliothek auch nur die Klassen das Interface, deren Objekte serialisierbar sind. Die EJB-Spezifikation definiert genauer, welche Bedingungen die Objekte des Conversational-State einer Session-Bean erfüllen müssen:

- Serialisierbares Objekt
- Referenz auf NULL
- Referenz auf ein Remote-Interface

- Referenz auf ein Home-Interface
- Referenz auf ein Local-Interface
- Referenz auf ein Local-Home-Interface
- Referenz auf den *SessionContext*
- Referenz auf den JNDI-Namensdienst der Bean-Umgebung
- Referenz auf das Interface *UserTransaction*
- Referenz auf ein Resource-Factory (siehe unten)

Eine Session-Bean-Instanz wird immer vor ihrer Auslagerung auf ein sekundäres Speichermedium durch den Aufruf der Methode *ejbPassivate* benachrichtigt. Die Session-Bean-Instanz muss dann die oben genannten Bedingungen herstellen. Das bedeutet beispielsweise, dass Socket- und Datenbankverbindungen geschlossen werden müssen. Um Probleme bei der Serialisierung zu vermeiden, sollten alle Referenzen auf zusätzliche Ressourcen *transient* deklariert werden. Wird die Session-Bean-Instanz wieder in den Arbeitsspeicher geholt, so wird sie erneut durch den Aufruf der Methode *ejbActivate* benachrichtigt. Damit kann sie beispielsweise die Socket- und Datenbankverbindungen wieder aufbauen.

4.2.2 Zustandslose und zustandsbehaftete Session-Beans

Session-Beans können einen eigenen Conversational-State haben oder zustandslos sein. Aufgrund dieser Eigenschaften werden zwei Arten von Session-Beans unterschieden:

- zustandslose Session-Beans (*stateless Session Beans*)
- zustandsbehaftete Session-Beans (*stateful Session Beans*)

Wenn eine zustandsbehaftete Session-Bean neu erzeugt wird, muss ihr Zustand initialisiert werden. Dazu bietet sie eine oder mehrere *create*-Methoden an, denen als Parameter die nötigen Daten vom Client übergeben werden. Eine zustandslose Session-Bean muss keinen Zustand initialisieren. Deshalb hat sie immer nur eine *create*-Methode ohne Parameter. Für jede Session-Bean-Klasse wird im Deployment-Deskriptor definiert, zu welcher Art sie gehört und welche Methoden sie anbietet.

Die Unterscheidung der beiden Arten hilft dem EJB-Container bei der Verwaltung der Bean-Instanzen. Um den Ressourcenverbrauch gering und die Performanz hoch zu halten, verwendet der EJB-Container Strategien für die Verwaltung, die das Erzeugen und Verwerfen von Java-Objekten soweit wie möglich vermeiden.

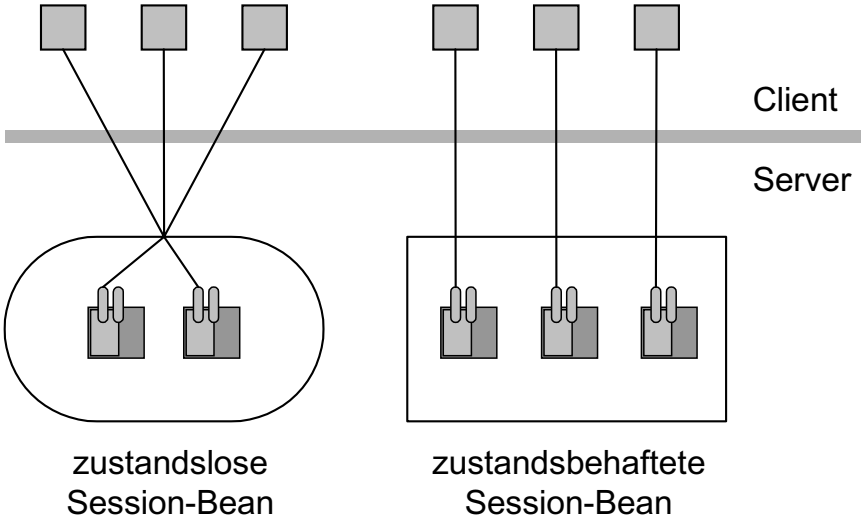

Abbildung 4.3: Zustandslose und zustandsbehaftete Session-Beans

Die Instanz einer zustandslosen Session-Bean kann vom EJB-Container nach jedem Methodenaufruf für einen anderen Client verwendet werden. Da diese keinen eigenen Conversational-State besitzt, können die aufeinanderfolgenden Methodenaufrufe eines Clients auch von unterschiedlichen Instanzen bearbeitet werden. Die Instanz einer zustandslosen Session-Bean steht somit einem Client nur für die Dauer eines Methodenaufrufs exklusiv zur Verfügung. Der EJB-Container verteilt die Methodenaufrufe auf einen Pool mit einer festen Anzahl von Bean-Instanzen. Die Anzahl der benötigten Bean-Instanzen ist dadurch viel kleiner als die Anzahl der parallelen Client-Sessions.

Bei zustandsbehafteten Session-Beans hat der EJB-Container ein anderes Problem zu lösen. Damit der Conversational-State für einen Client während einer Sitzung erhalten bleibt und nur diesem exklusiv zur Verfügung steht, benötigt jeder Client eine eigene Session-Bean-Instanz. Damit steht jede Instanz einer zustandsbehafteten Session-Bean-Instanz für die Dauer einer Session immer nur einem Client zur Verfügung.

Damit trotzdem die Anzahl der Bean-Instanzen auch bei sehr vielen parallelen Clients begrenzt bleibt, wird eine besondere Strategie benötigt. Bei der Aktivierung bzw. Passivierung kann eine Session-Bean aus dem Arbeitsspeicher ausgelagert und später wieder zurückgeladen werden (vergleiche dazu auch die Ausführungen des vorangegangenen Abschnitts).

Der entscheidende Unterschied ist somit, dass die Aktivierung und Passivierung nur für die zustandsbehafteten Session-Beans benötigt werden. Dieser Unterschied war wichtig genug, um beim EJB-Konzept eine Unterteilung der Session-Beans in zwei Arten einzuführen.

Der EJB-Container kann dadurch wesentlich effizienter mit zustandslosen Session-Beans umgehen, als mit zustandsbehafteten Session-Beans. Die Serialisierung und Deserialisierung von Java-Objekten ist eine relativ teure Operation, d.h. sie verbraucht viel Rechenzeit. Deshalb zahlt sich der Einsatz von zustandslosen Session-Beans besonders dann aus, wenn viele parallele Clients zu bedienen sind. Der Vergleich mit Webservern bietet sich an: Einer der Gründe für den Erfolg des Word Wide Web ist, dass ein Webserver mit statischen Webseiten zustandslos arbeiten kann und damit sehr viele Clients parallel bedienen kann.

Wegen ihres höheren Ressourcenverbrauchs werden zustandsbehaftete Session-Beans als *schwergewichtige Beans* bezeichnet, während zustandslose Session-Beans als *leichtgewichtig* angesehen werden.

Zustandslose Session-Beans sind jedoch nur für bestimmte Anwendungsfälle geeignet. Tabelle 4.1 fasst die Unterschiede zwischen zustandslosen und zustandsbehafteten Session-Beans zusammen.

Zustandslose Session-Beans	**Zustandsbehaftete Session-Beans**
haben keinen Conversational-State	haben Conversational-State
sind leichtgewichtig	sind schwergewichtig
werden nicht aktiviert und passiviert	Werden aktiviert und passiviert
Instanz steht einem Client für die Dauer eines Methodenaufrufs zur Verfügung	Instanz steht einem Client für die Dauer einer Sitzung zur Verfügung

Tabelle 4.1: Vergleich zustandsloser und zustandsbehafteter Session-Beans

4.2.3 Die Sicht des Clients auf Session-Beans

Der Client hat eine sehr einfache Sicht auf Session-Beans. Der EJB-Container verbirgt die Mechanismen der Verwaltung von Bean-Instanzen vor dem Client. Der Umgang des Clients mit einer Session-Bean ist im Wesentlichen davon abhängig, ob sich der Client im selben Java-Prozess befindet wie die Session-Bean, oder nicht.

Local- vs. Remote-Client-View

Der **Remote-Client-View** ist die Sicht des Clients auf eine Session-Bean, die sich in einem anderen Java-Prozess befindet als der Client selbst. Der Remote-Client-View ist der Standardfall der Sicht des Clients auf eine Session-Bean. Den Fall des Local-Client-Views (siehe nächster Abschnitt) gibt es erst seit der Version 2.0 der EJB-Spezifikation. Den Remote-Client-View benutzen z.B. Client-Anwendungen, die auf Client-Rechnern installiert sind und die Enterprise-Beans eines Applikationsservers über das Netzwerk benutzen. Den Remote-Client-View benutzen auch Enterprise-Beans, die sich zur Erfüllung ihrer Aufgabe anderer Enterprise-Beans bedienen, die in einem anderen, entfernten Applikationsserver installiert sind.

Mit der Methode *create* des *Home-Objekts* (vgl. dazu auch Kapitel 3) kann der Client neue Session-Beans einer bestimmten Klasse erzeugen. Die Session-Bean steht ihm exklusiv zur Verfügung, bis er sie durch den Aufruf der Methode *remove* am *Remote-Objekt* (vgl. dazu auch Kapitel 3) wieder löscht. Solange die Session-Bean existiert, kann der Client ihre Methoden über das Remote-Objekt aufrufen. Die Parameter und die Rückgabewerte der Methoden, die der Client über das Home- und das Remote-Objekt aufruft, werden call-by-value (als Kopie) übergeben. Diese Semantik eines Methodenaufrufs wird durch Java-RMI bestimmt, welche die EJB-Spezifikation dem Remote-Client-View zugrunde legt.

Der **Local-Client-View** ist die Sicht des Clients auf eine Session-Bean, die sich im gleichen Java-Prozess befindet wie der Client selbst. Dabei kann es sich um Enterprise-Beans handeln, die zur Erfüllung ihrer Aufgabe andere Enterprise-Beans benutzen, die im gleichen Applikationsserver installiert sind. Ein anderes Beispiel sind serverseitige JMS-Clients oder Servlets, die ihre Dienste im gleichen Java-Prozess (sprich im gleichen Applikationsserver) verrichten wie die jeweilige Session-Bean.

Mit der Methode *create* des Local-*Home-Objekts* (vgl. dazu auch Kapitel 3) kann der Client neue Session-Beans einer bestimmten Klasse erzeugen. Die Session-Bean steht ihm exklusiv zur Verfügung, bis er sie durch den Aufruf der Methode *remove* am *Local-Objekt* (vgl. dazu auch Kapitel 3) wieder löscht. Solange die Session-Bean existiert, kann der Client ihre Methoden über das Local-Objekt aufrufen. Die Parameter und die Rückgabewerte der Methoden, die der Client über das Local-Home- und das Local-Objekt aufruft, werden call-by-reference (als Referenz) übergeben. Diese Semantik eines Methodenaufrufs wird durch die Programmiersprache Java bestimmt. Dies ist aus Sicht des Clients der wesentliche Unterschied zum Remote-Client-View.

Der Bean-Entwickler legt fest, ob die Session-Bean über den Local- oder den Remote-Client-View benutzt werden kann. Laut Spezifikation sollte eine Enterprise-Bean entweder den Local- oder den Remote-Client-View unterstützen. Theoretisch wäre es möglich, dass die Enterprise-Bean sowohl den Local- als auch den Remote-Client-View unterstützt. Ein bestimmter Client einer bestimmten Session-Bean programmiert in jedem Fall entweder gegen das Local- oder das Remote-Interface der Session-Bean. Befindet sich der Client nicht im gleichen Java-Prozess wie die Enterprise-Bean, muss er den Remote-Client-View benutzen. Bietet die Session-Bean in diesem Fall nur einen Local-Client-View an, kann er die Session-Bean nicht benutzen. Befindet sich der Client im gleichen Java-Prozess wie die Session-Bean, wird er den Local-Client-View verwenden (sofern die Session-Bean einen anbietet), da die Laufzeiteffizienz wesentlich höher ist als über den Remote-Client-View (RMI-Overhead entfällt). Wenn die Session-Bean nur den Remote-Client-View unterstützt und der Client befindet sich im selben Prozess wie die Session-Bean, kann er sie trotzdem verwenden.

Zustandslose und zustandsbehaftete Session-Beans

Beim Umgang mit stateful oder stateless Session-Beans muss der Client keine Unterscheidung treffen. Der Client weiß unter Umständen nicht einmal, ob er es mit einer stateless oder einer stateful Session-Bean zu tun hat.

Lebenszyklus

Der Client muss vier Zustände unterscheiden. Diese ergeben sich aus der Tatsache, dass der Client nicht direkt mit der Bean-Instanz arbeitet. Der Client arbeitet mit Proxy-Objekten des EJB-Containers (Local-Home- und Local-Objekt bzw. Home- und Remote-Objekt), welche die entsprechenden Aufrufe an die eigentliche Bean-Instanz delegieren.

1. *Nicht existent*: Der Client hat keine Referenz auf das Remote- bzw. Local-Objekt. Aus seiner Perspektive existiert deshalb auch keine Session-Bean-Instanz am Server.
2. *Existent und referenziert*: Der Client besitzt eine gültige Referenz auf das Remote- bzw. Local-Objekt und kann damit indirekt auf die Session-Bean-Instanz zugreifen.
3. *Nicht existent, aber referenziert*: Dieser Zustand ist für die Fehlerbehandlung wichtig. Nachdem der Client die Session-Bean-Instanz mit *remove* gelöscht hat, stellt das Remote- bzw. Local-Objekt keine gültige Referenz mehr dar. Wenn der Client trotzdem eine Methode am Remote- bzw. Local-Objekt aufruft, wird eine Exception ausgelöst, da der EJB-Container keine Bean-Instanz für diesen Aufruf bereithält. Der EJB-Container löscht auch Bean-Instanzen, die längere Zeit nicht verwendet wurden (*Timeout*) oder bei denen ein Systemfehler aufgetreten ist. Der Client kommt dann ohne eigenes Dazutun in diesen Zustand.
4. *Existent, aber nicht referenziert*: Der Client löscht seine Referenz auf das Home- bzw. Local-Home-Objekt, ohne zuvor die Methode *remove* aufzurufen. Nach einer definierten Zeitspanne wird der EJB-Container die Bean-Instanz löschen (*Timeout*).

4.2.4 Lebenszyklus einer Session-Bean-Instanz

Wie bereits erwähnt, nimmt eine Session-Bean-Instanz während ihrer Existenz unterschiedliche Zustände an. Die Übergänge von einem Zustand in einen anderen werden durch den Client (z.B. bei *create-* und *remove*-Methodenaufrufen) und den EJB-Container (durch die Verwaltung seiner Ressourcen) angestoßen. Gesteuert werden die Zustandsübergänge ausschließlich durch den EJB-Container (indem er die Callback-Methoden aufruft).

Damit der EJB-Container die Zustandswechsel einer Session-Bean-Instanz steuern kann, muss die Bean-Klasse das Interface *javax.ejb.SessionBean* implementieren. Das Interface stellt eine Art Vertrag zwischen dem EJB-Container und der Session-Bean dar.

Der EJB-Container ist verpflichtet, die Session-Bean-Instanz über einen Zustandswechsel durch den Aufruf der entsprechenden Methoden im Interface zu informieren. Die Bean-Instanz muss dafür sorgen, dass ihr Zustand für den folgenden Zustand geeignet ist.

Die Lebenszyklen von zustandslosen und zustandsbehafteten Session-Beans sind unterschiedlich und werden im Folgenden ausführlich dargestellt. Ob die Session-Bean den Remote- oder den Local-Client-View unterstützt, ist für den Lebenszyklus dagegen unerheblich. Das detaillierte Verständnis des Lebenszyklus einer Session-Bean-Instanz ist eine unbedingte Voraussetzung für die Entwicklung.

Zustandslose Session-Beans

Eine Instanz einer zustandslosen Session-Bean kennt nur zwei unterschiedliche Zustände:

- *Nicht existent*: Die Instanz existiert nicht.
- *Pooled Ready*: Die Instanz existiert und steht für Methodenaufrufe zur Verfügung.

Abbildung 4.4 stellt den Lebenszyklus einer zustandslosen Session-Bean dar. Die Abbildung unterscheidet, ob der EJB-Container oder der Client einen Zustandsübergang veranlasst.

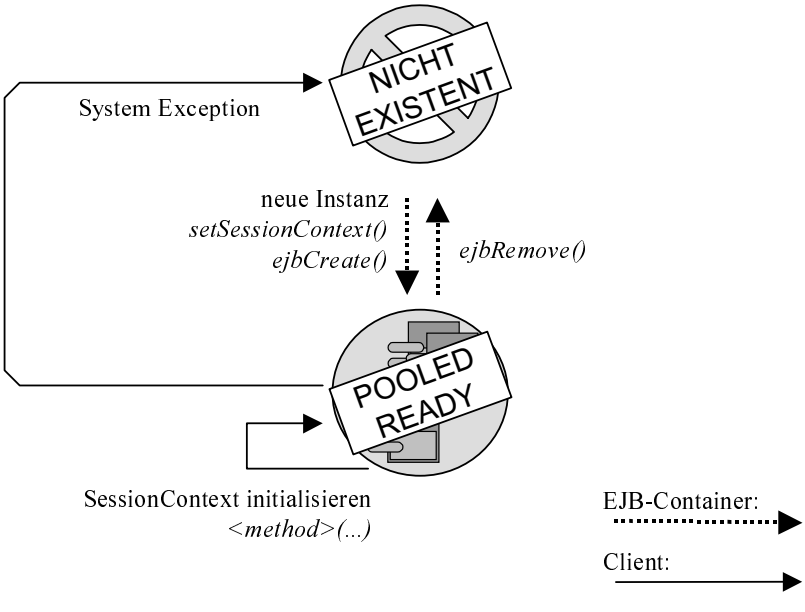

Abbildung 4.4: Lebenszyklus einer zustandslosen Session-Bean-Instanz

Der EJB-Container verwaltet die Bean-Instanzen. Benötigt er eine, so erzeugt er zunächst eine Instanz der entsprechenden Bean-Klasse. Dann ruft er die Methode *setSessionContext* auf und macht der Instanz damit ihren *Session-Kontext* bekannt. Der Session-Kontext ermöglicht der Instanz, auf ihre Umgebung (Identität und Rolle des Clients; Transaktionskontext) und auf den EJB-Container zuzugreifen.

Dann ruft der EJB-Container die Methode *ejbCreate* auf. Danach ist die Bean-Instanz im Zustand *Pooled Ready*. Sie wird vom EJB-Container mit anderen Instanzen ihrer Bean-Klasse in einem Pool bereitgehalten.

Die Methode *ejbCreate* entspricht der Methode *create* im Home- bzw. Local-Home-Interface, mit welcher der Client sich eine neue Instanz beschaffen kann. Der Aufruf des Clients kann jedoch als vollständig unabhängig vom Aufruf des EJB-Containers angesehen werden. Wenn der Client *create* für eine zustandslose Session-Bean aufruft, erhält er als Ergebnis ein Local- oder Remote-Objekt (abhängig davon, welchen Client-View er verwendet), das nur eine Referenz auf die Bean-Klasse und nicht auf eine konkrete Bean-Instanz ist.

Für jeden Aufruf des Clients einer Anwendungsmethode wird eine beliebige Bean-Instanz aus dem Pool benutzt. Damit die Bean-Instanz die Methode ausführen kann, muss für sie zuerst die richtige Umgebung geschaffen werden (z. B. wird ihr durch die Initialisierung des Session-Kontexts der Transaktionskontext und die Identität des Clients bekannt gegeben). Jetzt kann der EJB-Container den Aufruf des Clients an die Bean-Instanz weiterleiten. Nach der Durchführung der Methode wird die Instanz wieder im Pool abgelegt.

Durch diesen Mechanismus ist sichergestellt, dass eine Bean-Instanz immer nur einen Methodenaufruf eines Clients zu einem Zeitpunkt ausführt. Der EJB-Container serialisiert alle Aufrufe für die Bean-Instanzen. Die Implementierung einer Bean muss somit keine Zugriffe durch parallele Threads berücksichtigen.

Bei der Ausführung einer Methode kann es zu einem Fehler kommen. Deklarierte Exceptions der Methode werden einfach an den Client weitergereicht, und die Instanz wird wieder in den Pool aufgenommen. Tritt jedoch eine Ausnahme vom Typ *RuntimeException* auf, die nicht deklariert werden muss, so geht der EJB-Container von einem Systemfehler aus. Auch diese Exception wird an den Client weitergereicht. Die Bean-Instanz wird jedoch nicht wieder in den Pool aufgenommen, sondern wird verworfen.

Der EJB-Container kann die Anzahl der Bean-Instanzen im Pool verringern. Dafür ruft er zuerst die Methode *ejbRemove* an der Bean-Instanz auf, um die Instanz über die bevorstehende Löschung zu unterrichten, und verwirft dann die Instanz. Die Garbage-Collection wird die Instanz zu einem späteren Zeitpunkt löschen.

Zustandsbehaftete Session-Beans

Der Lebenszyklus von zustandsbehafteten Session-Beans ist komplizierter als der von zustandslosen Session-Beans. Es werden vier Zustände unterschieden:

- *Nicht existent*: Die Session-Bean-Instanz existiert nicht.
- *Ready*: Die Session-Bean-Instanz existiert und wurde einem Client zugeteilt. Sie steht für Methodenaufrufe zur Verfügung oder kann vom EJB-Container in diesem Zustand jederzeit passiviert werden.
- *Ready in TA*: Die Session-Bean-Instanz befindet sich in einer Transaktion und wartet auf Methodenaufrufe des Clients (siehe Kapitel 7, *Transaktionen*). Der EJB-Container darf Instanzen in Transaktionen nicht passivieren.
- *Passiviert*: Die Session-Bean-Instanz wurde zeitweilig aus dem Arbeitsspeicher ausgelagert, ist aber noch einem Client zugeteilt. Wenn der Client eine Methode ausführen möchte, muss der EJB-Container die Instanz zuerst wieder aktivieren.

Abbildung 4.5 stellt den Lebenszyklus einer zustandsbehafteten Session-Bean dar. Es wird wieder unterschieden, ob der EJB-Container oder der Client einen Zustandsübergang veranlasst.

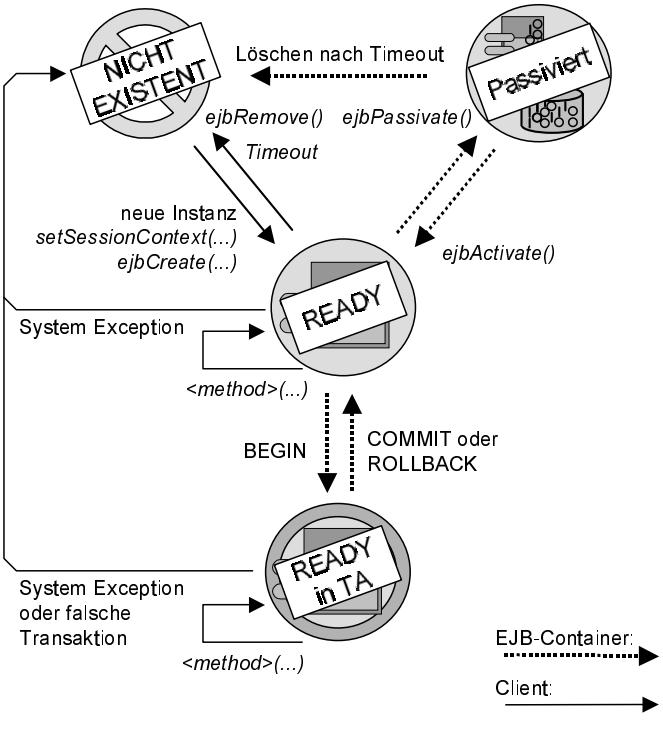

Abbildung 4.5: Lebenszyklus einer zustandsbehafteten Session-Bean-Instanz

Im Gegensatz zu den zustandslosen Session-Beans sieht die EJB-Spezifikation für die zustandsbehafteten Session-Beans keinen Pool vor. Der Lebenszyklus der zustandsbehafteten Session-Beans ist enger an den Gebrauch durch den Client gebunden.

Immer wenn der Client eine Methode *create* am (Local-) Home-Objekt aufruft, um eine neue Session-Bean zu erhalten, stellt der EJB-Container dem Client exklusiv eine neue Bean-Instanz zur Verfügung. Dafür erzeugt der EJB-Container zuerst eine Instanz der entsprechenden Bean-Klasse und macht ihr dann den Session-Kontext bekannt. Dann wird der Methodenaufruf des Clients an die entsprechende *ejbCreate*-Methode der Bean-Instanz weitergeleitet. Mit Hilfe der Parameter initialisiert diese Methode den Zustand der Bean-Instanz. Danach befindet sich die Bean-Instanz im Zustand *Ready*.

Wenn der Client zu einem späteren Zeitpunkt die Session-Bean nicht mehr benötigt und die Methode *remove* am Remote- bzw. am Local-Objekt aufruft, wird die Bean-Instanz vom EJB-Container verworfen. Der EJB-Container ruft die Methode *ejbRemove* der Bean-Instanz auf, um diese von der bevorstehenden Löschung zu unterrichten. Danach wird davon ausgegangen, dass die Bean-Instanz nicht mehr existent ist, obwohl der Garbage-Collector die Bean-Instanz eventuell erst zu einem späteren Zeitpunkt löscht.

Für jeden Client, der eine zustandsbehaftete Session-Bean verwendet, wird eine eigene Bean-Instanz benötigt. Ein EJB-Container muss jedoch die Anzahl der Bean-Instanzen im Arbeitsspeicher begrenzen. Mit der Passivierung und der Aktivierung hat er die Möglichkeit, Bean-Instanzen vorübergehend aus dem Arbeitsspeicher auszulagern. Bevor eine Bean passiviert wird, ruft der EJB-Container die Methode *ejbPassivate* auf, um die Bean über den bevorstehenden Zustandswechsel zu informieren. Bei der Aktivierung wird die Bean-Instanz zuerst wieder in den Arbeitsspeicher geladen und dann durch den Aufruf der Methode *ejbActivate* informiert.

Nachdem der Client eine Session-Bean erhalten hat, kann er über das Remote- bzw. Local-Objekt auf die Bean-Instanz zugreifen. Die Methodenaufrufe werden vom EJB-Container an die Bean-Instanz weitergeleitet. Alle Methodenaufrufe des Clients werden vom EJB-Container serialisiert. Wenn eine *RuntimeException* in einer Methode auftritt, wird wie bei zustandslosen Session-Beans die Bean-Instanz vom EJB-Container verworfen.

Eine Methode kann in einer Transaktion ausgeführt werden. Normalerweise startet der EJB-Container eine eigene Transaktion für jeden Methodenaufruf und beendet sie nach der Ausführung der Methode (dies hängt von der Konfiguration der Bean ab). Es ist jedoch auch möglich, dass mehrere Methodenaufrufe in einer Transaktion zusammengefasst werden. Dieser Fall wird mit dem Zustand *Ready in TA* berücksichtigt. Aus technischen Gründen darf der EJB-Container eine Bean-Instanz nicht passivieren, wenn sie sich in einer Transaktion befindet. Transaktionen werden in Kapitel 7, *Transaktionen*, ausführlich behandelt.

Der EJB-Container muss auch den Fall berücksichtigen, dass die Verbindung zum Client verloren geht. Hier wird mit einer einfachen Heuristik gearbeitet. Wenn der Client innerhalb eines bestimmten Zeitraums keine Methodenaufrufe durchgeführt hat (*Timeout*), wird die Sitzung als beendet angesehen. Der EJB-Container löscht die Bean-Instanz je nach Zustand mit *ejbRemove* und Verwerfen oder durch das Löschen der serialisierten Bean-Instanz.

4.3 Programmierung

4.3.1 Überblick

Abbildung 4.6 gibt einen vollständigen Überblick über die Klassen und Interfaces für den Fall des Remote-Client-Views. Rechts steht die Session-Bean-Klasse, auf der linken Seite befinden sich die Schnittstellen, die auch am Client verfügbar sind und durch das Home- bzw. Remote-Objekt des EJB-Containers implementiert werden (vgl. dazu auch Kapitel 3). Die in der Abbildung durchgestrichenen Methoden können mit Session-Beans nicht sinnvoll eingesetzt werden. Sie sind für Entity-Beans gedacht.

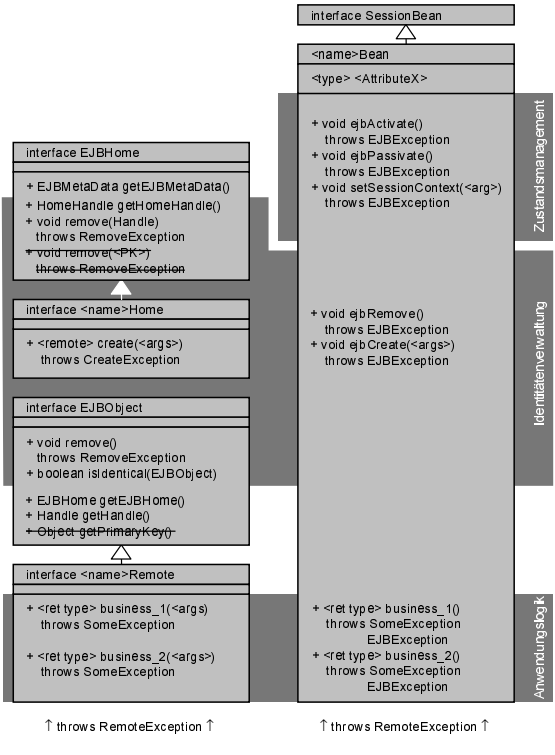

Abbildung 4.6: Interfaces einer Session-Bean mit Remote-Client-View

Die Session-Bean-Klasse implementiert das Interface *javax.ejb.SessionBean*. Der Bean-Entwickler implementiert diese Klasse. In ihr ist die Funktionalität der Session-Bean realisiert. In der Bean-Klasse wird kein eigener Konstruktor und keine *finalize*-Methode definiert. Wenn ein Konstruktor definiert wird, so muss er *public* deklariert sein und darf keine Übergabeparameter besitzen (Default-Konstruktor).

Das Home- und das Remote-Objekt gehören zu den Container-Klassen und werden generiert (vgl. dazu auch Kapitel 3). Der Bean-Entwickler definiert das zugehörige Home- und Remote-Interface. Das Home-Interface der Bean erbt vom Interface *javax.ejb.EJBHome*. Das Remote-Interface erbt vom Interface *javax.ejb.EJBObject*.

Abbildung 4.7 gibt einen vollständigen Überblick über die Klassen und Interfaces für den Fall des Local-Client-Views. Rechts steht die Session-Bean-Klasse, auf der linken Seite befinden sich die Schnittstellen, die vom Client benutzt werden und durch das Local-Home- bzw. Local-Objekt des EJB-Containers implementiert werden (vgl. dazu auch Kapitel 3). Die in der Abbildung durchgestrichenen Methoden können mit Session-Beans nicht sinnvoll eingesetzt werden. Sie sind für Entity-Beans gedacht.

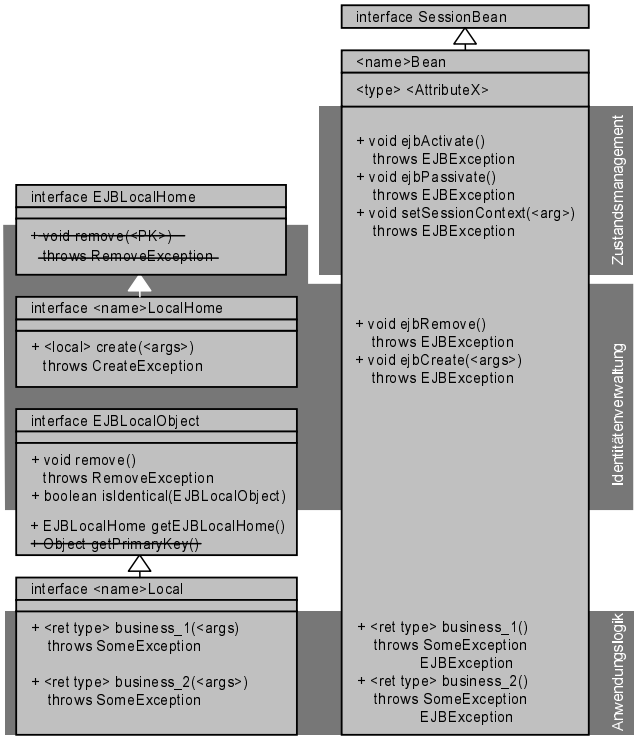

Abbildung 4.7: Interfaces einer Session-Bean mit Local-Client-View

Die Session-Bean-Klasse ist im Fall des Local-Client-Views identisch mit dem des Remote-Client-Views.

Das Local-Home- und das Local-Objekt gehören zu den Container-Klassen und werden generiert (vgl. dazu auch Kapitel 3). Der Bean-Entwickler definiert das zugehörige Local-Home- und Local-Interface. Das Local-Home-Interface der Bean erbt vom Interface *javax.ejb.EJBLocalHome*. Das Local-Interface erbt vom Interface *javax.ejb.EJBLocalObject*.

Eine Session-Bean könnte sowohl den Remote- als auch den Local-Client-View implementieren. Laut Spezifikation sollte eine Enterprise-Bean jedoch entweder den Local- **oder** den Remote-Client-View implementieren, nicht beide.

Alle Methoden aus der Bean-Klasse sowie aus den Home- und Remote-Interfaces lassen sich in drei Gruppen aufteilen:

- Zustandsmanagement
- Identitätenverwaltung
- Anwendungslogik

Die folgenden Abschnitte beschreiben die einzelnen Methoden in diesen Gruppen. Daran anschließend wird der Session-Kontext und die Schnittstellen, die das Clientprogramm verwendet, vorgestellt. Für einen schnellen Einstieg können die folgenden Beschreibungen auch vorerst übersprungen werden. Die Programmbeispiele zu Session-Beans ermöglichen Ihnen den schnellen Einstieg in die praktischen Programmierung.

4.3.2 Zustandsmanagement

In Abbildung 4.6 bzw. Abbildung 4.7 stehen die Methoden für das Zustandsmanagement an oberster Stelle. Diese Methoden werden vom EJB-Container aufgerufen, um die Bean-Instanz über einen Zustandswechsel zu informieren. In den Methoden wird die Logik implementiert, die die Bean-Instanz für den folgenden Zustand vorbereitet.

Sowohl die Aktionen des Clients als auch interne Prozesse des EJB-Containers können den Zustandswechsel auslösen, der zum Aufruf dieser Methoden führt. Der Client erhält jedoch keinen direkten Zugriff auf diese Methoden.

void setSessionContext(SessionContext ctx)

Die Methode *setSessionContext* wird nach dem Erzeugen der Bean-Instanz vom EJB-Container aufgerufen. Der EJB-Container übergibt der Session-Bean ihren *SessionContext*. Die Implementierung dieser Methode hat die Aufgabe, den *SessionContext* zu speichern. Weitere Initialisierungsschritte sollten hier normalerweise nicht vorgenommen werden. Das übergebene Objekt der Klasse *SessionContext* definiert die Umgebung der Bean-Instanz (siehe auch Abschnitt 4.3.5, *Session-Kontext*).

void ejbPassivate()

Diese Methode wird vom EJB-Container aufgerufen, bevor er eine zustandsbehaftete Session-Bean-Instanz passiviert. Zustandslose Session-Beans benötigen diese Methode nicht und lassen die Implementierung leer.

Die Methode wird genutzt, um die Bean-Instanz in einen Zustand zu bringen, der sich für die Passivierung eignet. Die zusätzlichen Ressourcen, welche von der Bean-Instanz genutzt wurden, müssen wieder freigegeben werden. Beispielsweise müssen hier Netzwerk- oder Datenbankverbindungen geschlossen werden. Alle Referenzen auf zusätzliche Ressourcen (andere Enterprise-Beans, Netzwerkverbindungen, Datenbankverbindungen etc.) sollten vom Entwickler der Bean *transient* deklariert werden. Falls dies nicht geschehen ist, müssen diese Referenzen mit *null* initialisiert werden. Abschnitt 4.2.1, *Conversational-State*, gibt die genauen Bedingungen an, die eine Bean für die Passivierung erfüllen muss.

Beachten Sie, dass die Methode *ejbPassivate* nicht in einer Transaktion ausgeführt wird. Der EJB-Container stellt sicher, dass keine Bean-Instanz, die sich in einer Transaktion befindet, passiviert wird. Zum Zeitpunkt der Passivierung enthält der Conversational-State der Bean-Instanz also keine Daten, die mit einer Transaktion geschützt werden müssen. Die Passivierung muss somit auch ohne Transaktion möglich sein.

Tritt in dieser Methode ein Fehler auf, der das Passivieren der Bean-Instanz verhindert, soll die Exception *javax.ejb.EJBException* ausgelöst werden (bei EJB 1.0 wurde noch *java.rmi.RemoteException* verwendet). Der EJB-Container wertet diese Exception als Systemfehler und zerstört die Bean-Instanz.

void ejbActivate()

Die Methode *ejbActivate* ist das Gegenstück zu *ejbPassivate*. Der EJB-Container ruft diese Methode auf, nachdem er eine passivierte Bean-Instanz zurück in den Arbeitsspeicher geladen hat.

Die Methode wird genutzt, um zusätzliche Ressourcen wieder zu reservieren, die bei der Passivierung freigegeben wurden. Beispielsweise können hier Netzwerk- oder Datenbankverbindungen wieder geöffnet werden.

Wie auch bei *ejbPassivate,* steht bei *ejbActivate* keine Transaktion zur Verfügung. Die Fehlerbehandlung ist äquivalent zu *ejbPassivate*.

4.3.3 Verwaltung der Identität

Die Methoden für die Identitätenverwaltung bilden die zweite Gruppe in den Grafiken. Sie ermöglichen das Erzeugen und Löschen von Session-Bean-Identitäten. Diese Methoden finden sich im Local-Home- und Local- bzw. Home- und Remote-Interface wieder und stehen so dem Client zur Verfügung, schon bevor er eine Bean-Instanz kennt.

void ejbCreate(...)

Der Client erhält eine Bean-Instanz durch den Aufruf von *create* im Home- bzw. Local-Home-Interface. Zustandsbehaftete Session-Beans können mehrere *create*-Methoden anbieten, die sich durch ihre Signaturen unterscheiden. Zustandslose Session-Beans haben nur eine *create*-Methode ohne Parameter. Nachdem der EJB-Container eine neue Bean-Instanz erzeugt hat, ruft er deren *ejbCreate*-Methode auf. Im Fall von zustandsbehafteten Session-Beans leitet er den *create*-Aufruf des Clients an die *ejbCreate*-Methode der Bean mit der passenden Signatur weiter.

Der Bean-Entwickler definiert die *create*-Methoden im Home- bzw. Local-Home-Interface und implementiert die *ejbCreate*-Methoden in der Bean-Klasse. Die Signaturen der Methoden in der Bean-Klasse und im Home- bzw. Local-Home-Interface müssen identisch sein. Die Implementierung der *ejbCreate*-Methoden hat die Aufgabe, die Bean-Instanz zu initialisieren. Eine zustandsbehaftete Session-Bean-Instanz initialisiert den Conversational-State mit Hilfe der verwendeten Parameter.

Die Methoden sind nicht Bestandteil des Interface *javax.ejb.SessionBean*, da die Signaturen für jede Bean-Klasse unterschiedlich sind. Jede Methode wird dem EJB-Container im Deployment-Deskriptor bekannt gegeben.

void ejbRemove()

Der Client gibt den Auftrag zum Löschen einer Session-Bean-Identität durch den Aufruf der Methode *remove* im Remote- bzw. Local-Interface. Falls er mit einem *Handle* arbeitet (zu *Handles* siehe Abschnitt 4.3.6, *Client*), kann er auch die Methode *remove(Handle)* im Home-Interface verwenden (nur Remote-Client-View). Die Methode *remove(Object)* im Home-Interface ist Entity-Beans vorbehalten und kann von Session-Beans nicht genutzt werden.

Im Fall von zustandsbehafteten Session-Beans wird der Aufruf des Clients vom EJB-Container an die Methode *ejbRemove* der entsprechenden Bean-Instanz weitergeleitet. Im Fall von zustandslosen Session-Beans wird der EJB-Container nicht direkt durch den Client beauftragt, die Methode *ejbRemove* aufzurufen.

Die Methode *remove* wird in *javax.ejb.EJBObjekt* bzw. *javax.ejb.EJBLocalObjekt* definiert, der Bean-Entwickler implementiert *ejbRemove* in der Bean-Klasse. Die Implementierung von *ejbRemove* gibt reservierte Ressourcen wieder frei. Nach dem Aufruf der Methode wird die Bean-Instanz gelöscht.

Die Methode *ejbRemove* kann vom EJB-Container mit einer Transaktion aufgerufen werden. Das zugehörige Transaktionsattribut im Deployment-Deskriptor definiert hier das Verhalten des EJB-Containers (siehe Kapitel 7, *Transaktionen*).

4.3.4 Anwendungslogik

Die dritte Gruppe in der Abbildung sind die Methoden für die Anwendungslogik, die sich auch im Remote- bzw. Local-Interface wiederfinden. Diese Methoden werden vom Bean-Entwickler in der Bean-Klasse implementiert. Die Definition in der Bean-Klasse hat die gleiche Signatur wie die Definition im Remote- bzw. Local-Interface.

Der EJB-Container kontrolliert alle Aufrufe der Methoden einer Bean. Er leitet die Aufrufe des Clients zu den entsprechenden Methoden der Bean-Klasse weiter. Der Client greift niemals direkt auf die Bean-Instanz zu. Die Transaktionsattribute im Deployment-Deskriptor definieren den Transaktionskontext, in dem die Methoden ausgeführt werden.

Der Bean-Entwickler implementiert in diesen Methoden die eigentliche Logik der Session-Bean. Alle bisher besprochenen Methoden erfüllen hingegen nur Verwaltungsaufgaben.

Tritt in der Anwendungslogik ein nicht behebbarer Fehler auf, muss eine *java.ejb.EJBException* ausgelöst werden. Der EJB-Container bewertet diese als Systemfehler und zerstört die Bean-Instanz.

4.3.5 Der Session-Kontext

Jede Session-Bean-Instanz hat ein Objekt der Klasse *javax.ejb.SessionContext*, das ihr vom EJB-Container in der Methode *setSessionContext* zur Verfügung gestellt wird. Der *Session-Kontext* definiert die Umgebung der Bean-Instanz. Die Session-Bean-Instanz verwendet den übergebenen *Session-Kontext* während ihrer gesamten Lebensdauer. Der EJB-Container kann den *Session-Kontext* einer Bean-Instanz verändern und dieser somit eine veränderte Umgebung zuweisen. Wenn z. B. unterschiedliche Clients der Reihe nach eine Instanz einer zustandslosen Session-Bean verwenden, setzt vor jedem Methodenaufruf der EJB-Container die Identität des Clients im Session-Kontext.

Der Session-Kontext ermöglicht der Bean-Instanz den Zugriff auf das Home-Objekt, die Identität des Clients und die aktive Transaktion. Im Folgenden werden die wichtigsten Methoden aus dem Interface *SessionContext* vorgestellt. Die vollständige Beschreibung können Sie in der EJB-Spezifikation (siehe [Sun Microsystems, 2001]) nachlesen.

EJBHome getEJBHome()

Diese Methode ermöglicht der Bean-Instanz den Zugriff auf ihr eigenes Home-Interface. Der Datentyp des Rückgabewerts muss in das spezielle Home-Interface der Bean-Klasse umgewandelt werden. Die Methode löst eine Ausnahme vom Typ *java.lang.IllegalStateException* aus, wenn die Session-Bean kein Home-Interface (sondern ein Local-Home-Interface) hat.

EJBLocalHome getEJBLocalHome()

Diese Methode ermöglicht der Bean-Instanz den Zugriff auf ihr eigenes Local-Home-Interface. Der Datentyp des Rückgabewerts muss in das spezielle Local-Home-Interface der Bean-Klasse umgewandelt werden. Die Methode löst eine Ausnahme vom Typ *java.lang.IllegalStateException* aus, wenn die Session-Bean kein Local-Home-Interface (sondern ein Home-Interface) hat.

void setRollbackOnly()

Diese Methode dient zur Transaktionssteuerung (siehe auch Kapitel 7, *Transaktionen*). Eine Bean-Instanz verwendet diese Methode im Fehlerfall, um sicherzustellen, dass die aktive Transaktion mit einem Rollback beendet wird.

boolean getRollbackOnly()

Die Methode *getRollbackOnly* ist das lesende Gegenstück zu der Methode *setRollbackOnly*. Mit *getRollbackOnly* kann überprüft werden, ob es noch möglich ist, die aktive Transaktion mit *Commit* zu beenden.

UserTransaction getUserTransaction()

Normalerweise übernimmt die Steuerung der Transaktionen der EJB-Container. Die Methode *getUserTransaction* benötigt eine Bean-Instanz nur, wenn sie die Transaktionssteuerung selbst übernehmen möchte. Der Rückgabewert der Methode ist ein Objekt der Klasse *javax.transaction.UserTransaction*, das Zugriff auf den Transaktionskontext ermöglicht (siehe Kapitel 7, Transaktionen).

Principal getCallerPrincipal()

Die Methode erlaubt es der Bean-Instanz, den Benutzer zu bestimmen, der sich am Client angemeldet hat. Der Rückgabewert ist ein Objekt der Klasse *java.security.Principal* (siehe Kapitel 8, Sicherheit).

boolean isCallerInRole(java.lang.String roleName)

Mit dieser Methode wird überprüft, ob der angemeldete Benutzer eine bestimmte Rolle im Sicherheitskonzept hat. Die möglichen Rollen werden im Deployment-Deskriptor definiert (siehe Kapitel 8, Sicherheit).

4.3.6 Client

Namensdienst als Objektspeicher

Ein Namensdienst verwaltet Name-Wert-Paare in einem Namensraum. Jeder Name ist im Namensraum eindeutig und identifiziert einen Wert. Das EJB-Konzept verwendet

den Namensdienst für die Verwaltung der Home-Objekte und anderer Ressourcen. Das Home-Objekt wird als Wert gespeichert und unter dem Namen der Bean im Namensdienst eingetragen (dies gilt für den Local- und den Remote-Client-View). Jeder Client kann sich dadurch vom Namensdienst das Home-Objekt der Bean geben lassen, wenn er den Namen der Bean kennt. Über das Home-Objekt erhält er dann indirekt Zugriff auf die Bean.

Für den Zugriff von Java auf den Namensdienst steht JNDI (Java Naming and Directory Interface) zur Verfügung. Das Interface wurde von Sun Microsystems definiert und gehört mit der Version Java 1.2 zum Standardsprachumfang.

In der Praxis wurde in viele Applikationsserver ein eigener Namensdienst integriert, der über JNDI angesprochen wird. Der Einsatz eines externen Namensdienstes kann jedoch auch sinnvoll sein. JNDI unterstützt folgende Protokolle:

- RMI Registry (Java Remote Method Invocation)
- LDAP (Lightweight Directory Access Protocol)
- COS (CORBA Services Naming Service)

Um Zugriff auf JNDI zu erhalten, wird die Klasse *javax.naming.InitialContext* verwendet. Diese implementiert das Interface *javax.naming.Context*. Jede Instanz dieser Klasse hat einen vorinitialisierten Namensraum, der mit den Daten aus zwei Quellen initialisiert wird:

- Im Konstruktor kann ein Hashtable-Objekt übergeben werden. Dessen Name-Wert-Paare werden in den Namensraum des *InitialContext* übertragen.
- Die Datei *jndi.properties* im Klassenpfad (*classpath*) enthält ebenfalls Name-Wert-Paare. Auch diese werden übernommen.

Bestimmte Name-Wert-Paare aus dem Hashtable-Objekt, das im Konstruktor übergeben wird, werden nicht einfach in den Namensraum übernommen, sondern dienen zur Initialisierung von JNDI. Das Interface *Context* definiert die nötigen Konstanten, die die verwendeten Namen definieren:

- *INITIAL_CONTEXT_FACTORY* – unter diesem Namen muss immer der Name einer Klasse gespeichert werden, die als Factory (vgl. Design-Pattern *Factory* in [Gamma et al., 1996]) für einen Namensdienst dient. Der hier benötigte Wert muss von jedem Hersteller eines EJB-Containers bzw. eines Namensdienstes angegeben werden.
- *PROVIDER_URL* – hier wird eine URL für den Namensdienst definiert. Die URL ist ein String, der den Netzwerknamen des Servers und den verwendeten Port enthält. Der Hersteller des Namensdienstes muss zur Bildung dieser URL genaue Angaben machen.

- *SECURITY_PRINCIPAL* – hier wird der Benutzername für den Namensdienst definiert. Je nach Konfiguration des Namensdienstes kann es auch möglich sein, ohne Benutzernamen zu arbeiten.
- *SECURITY_CREDENTIALS* – hier wird das zugehörige Passwort zu dem angegebenen Benutzernamen definiert.

Folgende *import*-Anweisungen sind in unserem Fall für die Verwendung von JNDI erforderlich:

```
import javax.naming.Context;
import javax.naming.InitialContext;
```

Listing 4.1: zeigt eine Methode, die ein Clientprogramm für die Initialisierung des *InitialContext* benutzen könnte. Je nach verwendetem Server müssten die fettgedruckten Strings durch die entsprechenden Werte ersetzt werden.

```
final static String ctxUser     = "CtxUser";
final static String ctxPassword = "CtxPassword";
final static String ctxFactory  = "CtxFactory";
final static String ctxUrl      = "CtxURL";

/**
 * Holt und initialisiert InitialContext.
 * Die Methode verwendet die
 * folgenden Variablen:
 *   ctxUser, ctxPassword,
 *   ctxFactory, ctxUrl
 *
 * @return Context - InitialContext Object
 * @exception javax.naming.NamingException
 */
static public Context getInitialContext()
  throws javax.naming.NamingException
{
  Hashtable p = new Hashtable();

  p.put(
    Context.INITIAL_CONTEXT_FACTORY,
    ctxFactory);
  p.put(
    Context.PROVIDER_URL,
    ctxUrl);

  //Dieser Teil ist optional
    p.put(
      Context.SECURITY_PRINCIPAL,
      ctxUser);
    p.put(
      Context.SECURITY_CREDENTIALS,
```

```
        ctxPassword);
    //Dieser Teil ist optional

    return new InitialContext(p);
  }
```

Listing 4.1: getInitialContext – Initialisierung für den JNDI-Zugriff

Zugriff auf eine Session-Bean

Der Client erhält mit Hilfe des Namensdienstes Zugriff auf das Home- bzw. Local-Home-Interface. Beim Aufruf der Methode *lookup* an einem *Context*-Objekt gibt er den eindeutigen JNDI-Namen der Bean als Parameter an und erhält die Implementierung des Home- bzw. Local-Home-Interface als Rückgabewert. Der JNDI-Name einer Enterprise-Bean wird beim Deployment festgelegt. Die Vergabe des JNDI-Namens ist herstellerspezifisch.

Der Rückgabewert der Methode *lookup* hat den Datentyp *Object*. Im Fall des Local-Client-View kann der Rückgabewert durch type-casting in den Typ des Local-Home-Interfaces konvertiert werden. Beim Remote-Client-View ist dies nicht ohne weiteres möglich. Seit der Version 1.1 beschränkt sich EJB nicht mehr ausschließlich auf JRMP (Standardprotokoll von RMI) als Netzwerkprotokoll, sondern unterstützt auch IIOP, um mit CORBA-Objekten kommunizieren zu können. Deshalb ist jetzt immer eine Anpassung der über das Netz übertragenen Daten erforderlich.

Für die richtige Konvertierung der Datentypen gibt es die Methode *narrow* von der Klasse *javax.rmi.PortableRemoteObject*. Die Klasse gehört seit der Version 1.3 zum Standardsprachumfang von Java. Für ältere Java-Versionen kann auf die Veröffentlichung mit den RMI-IIOP-Mapping zurückgegriffen werden. Das Vorgehen bei der Datentypkonvertierung des Home-Interface ist immer das gleiche und wird in Listing 4.2 demonstriert.

Nachdem das Home-Interface bekannt ist, kann der Client eine neue Bean erzeugen (*create*) und erhält als Rückgabewert das Remote-Interface der Bean. Mit dem Remote-Interface kann der Client mit der Bean fast so arbeiten, als wäre sie ein lokales Objekt und würde nicht auf dem Server ausgeführt. Nach dem Gebrauch der Bean muss er die Bean wieder löschen (*remove*).

```
...
javax.naming.Context ctx;
String beanName = "BeanName";
HomeInterfaceClass home = null;
RemoteInterfaceClass bean = null;
try {
  ctx = getInitialContext();
  home = (HomeInterfaceClass)
    javax.rmi.PortableRemoteObject.narrow(
```

```
      ctx.lookup(beanName),
      HomeInterfaceClass.class);
    bean = home.create(...);
    ...
    bean.remove ();
  }
  catch(javax.ejb.CreateException e){
    System.out.println(
      "Fehler beim erzeugen der Bean");
  }
  catch(javax.ejb.RemoveException e){
    System.out.println(
      "Fehler beim löschen der Bean");
  }
  catch(javax.naming.NamingException e) {
    System.out.println(
      "Namensdienst meldet Fehler");
  }
  catch(RemoteException e) {
    System.out.println(
      "Probleme mit der Netzwerkverbindug");
  }
  ...
```

Listing 4.2: Zugriff des Clients auf eine Session-Bean (Remote-Client-View)

Listing 4.3 demonstriert die Verwendung einer Session-Bean über den Local-Client-View.

```
...
javax.naming.Context ctx;
String beanName = "LocalBeanName";
LocalHomeInterfaceClass home = null;
LocalBeanInterfaceClass bean = null;
try {
  ctx = getInitialContext();
  home = (LocalHomeInterfaceClass)
            ctx.lookup(beanName);
  bean = home.create(...);
  ...
  bean.remove ();
}
catch(javax.ejb.CreateException e){
  System.out.println(
    "Fehler beim erzeugen der Bean");
}
catch(javax.ejb.RemoveException e){
  System.out.println(
    "Fehler beim löschen der Bean");
}
catch(javax.naming.NamingException e) {
```

```
    System.out.println(
      "Namensdienst meldet Fehler");
  }
  ...
```

Listing 4.3: Zugriff des Clients auf eine Session-Bean (Local-Client-View)

Die Identität einer Session-Bean

Der Client verwendet die Identität von Session-Beans nur, um zu unterscheiden, ob zwei Referenzen auf die gleiche oder auf unterschiedliche Bean-Instanzen zeigen. Der Umgang mit zustandslosen und zustandsbehafteten Session-Beans ist hier unterschiedlich und soll im Folgenden beschrieben werden.

Jede zustandsbehaftete Session-Bean hat eine eigene Identität. Es handelt sich dabei um eine ID, die der EJB-Container für die interne Verwaltung der Instanzen nutzt. Die ID selbst ist für den Client nicht zugänglich. Für den Vergleich von Session-Beans stellt der EJB-Container dem Client eine Implementierung der Methode *isIdentical* zur Verfügung, die im Interface *javax.ejb.EJBObject* bzw. *javax.ejb.EJBLocalObject* wiefolgt definiert ist:

```
public abstract boolean isIdentical (EJBObject obj);
```

bzw.

```
public abstract boolean isIdentical (EJBLocalObject obj);
```

Listing 4.4 demonstriert den Gebrauch der Methode für zustandsbehaftete Session-Beans für den Fall, dass die Session-Bean ein Remote-Interface hat. Die Anwendung der Methode *isIdentical* für Local-Interfaces ist entsprechend.

```
// Holen des HomeObject mittels JNDI
StatefulBeanHome homeObject = ...;

// Zwei zustandsbehaftete
// Session-Beans erzeugen
StatefulBean sfb1 = homeObjekt.create(...);
StatefulBean sfb2 = homeObjekt.create(...);

// Vergleich der Identitäten
if (sfb1.isIdentical(sfb1)) {
  // Dieser Vergleich ergibt "true".
}
if (sfb1.isIdentical(sfb2)) {
  // Dieser Vergleich ergibt "false".
}
```

Listing 4.4: Vergleich der Identitäten von zustandsbehafteten Session-Beans

Die einzelnen Instanzen einer zustandslosen Session-Bean unterscheiden sich im Gegensatz zu zustandsbehafteten Session-Bean nicht durch einen Conversational-State. Der EJB-Container betrachtet alle Instanzen einer zustandslosen Session-Bean, die das gleiche Home- bzw. Local-Home-Interface haben, als gleichwertig, da sie untereinander austauschbar sind. Deshalb haben alle Instanzen einer zustandslosen Session-Bean-Klasse eine gemeinsame Identität. Listing 4.5 verdeutlicht diesen Zusammenhang für den Fall, dass die Session-Bean ein Remote-Interface hat. Auch hier ist die Anwendung im Fall eines Local-Interfaces entsprechend.

```
// Holen des HomeObject mittels JNDI
StatelessBeanHome homeObject = ...;

// Zwei zustandslose Session-Beans erzeugen
StatelessBean slb1 = homeObjekt.create(...);
StatelessBean slb2 = homeObjekt.create(...);

// Vergleich der Identitäten
if (slb1.isIdentical(slb1)) {
  // Dieser Vergleich ergibt "true".
}
if (slb1.isIdentical(slb2)) {
  // Dieser Vergleich ergibt "true".
}
```

Listing 4.5: Vergleich der Identitäten von zustandslosen Session-Beans

Handles

Handles sind nur beim Remote-Client-View relevant. Sie werden eingesetzt, wenn eine Referenz auf das Home- oder Remote-Interface benötigt wird, die auch außerhalb des Client-Prozesses verwendet werden kann. Handles können versendet oder über die Lebensdauer des Clients hinaus gespeichert werden. Ein Handle kann somit als serialisierbarer Zeiger auf ein Remote- oder Home-Objekt verstanden werden. Die Existenz eines Handle-Objekts beeinflusst den Lebenszyklus der Bean-Instanz jedoch nicht. Wenn die Bean-Instanz gelöscht wird (vom Client gesteuert oder nach Ablauf eines Timeout) zeigt das Handle-Objekt ins Leere.

Für das Erzeugen von Handles besitzt jedes Remote-Interface eine Methode *getHandle*, die ein Handle-Objekt zu dem Remote-Interface dieser Bean-Instanz zurückgibt (siehe Listing 4.6). Die entsprechende Methode beim Home-Objekt heißt *getHomeHandle*. Die Handle-Objekte haben ihrerseits eine Methode *getEJBObject*, deren Rückgabewert das verbundene Remote-Objekt der Bean ist.

```
...
try{
  javax.ejb.Handle handle;
```

```
    // get Handle
    Handle handle = remote.getHandle();

    // get Remote-Object with handle
    RemoteClass remote = (RemoteClass)
      javax.rmi.PortableRemoteObject.narrow(
      handle.getEJBObject(),
      RemoteClass.class);
  }catch(RemoteException e){
    // cannot get handle
  }
  ...
```

Listing 4.6: Verwendung eines Remote-Handles

Die Existenz eines Handle kann vom EJB-Container beim Verwalten der Bean-Instanzen jedoch nicht berücksichtigt werden. Nach dem Ablauf einer konfigurierten Zeitspanne (Timeout) wird jede Session-Bean vom EJB-Container gelöscht. Der Versuch, auf eine gelöschte Instanz zuzugreifen, führt zu einer RemoteException.

4.3.7 Umgebung der Bean

Eine Bean wird immer in einer definierten Umgebung ausgeführt. Alle Bean-Instanzen einer Klasse, die das gleiche Home- bzw. Local-Home-Interface haben, haben die gleiche Umgebung. Die Umgebung einer Bean besteht aus folgenden Teilen:

- Umgebungsvariablen für Konfigurationswerte
- Referenzen auf andere Bean-Klassen
- Ressourcen des EJB-Containers
- Administrierte Objekte (siehe unten)

Der Bean-Entwickler bestimmt, welche Eigenschaften die Umgebung der Bean hat. Er legt fest, welche Konfigurationsparameter die Bean hat, mit welchen anderen Beans sie arbeitet, und welche Ressourcen des EJB-Containers sie verwendet. Darauf basierend definiert der Application-Assembler und der Deployer die konkrete Laufzeitumgebung. Sie legen die Werte für die Konfiguration fest, wählen die Bean-Klassen für die Referenzen aus und konfigurieren die verfügbaren Dienste.

Zur Laufzeit stellt der EJB-Container der Bean-Instanz ihre Umgebung in einem Namensdienst zur Verfügung. Unter definierten Namen findet die Bean-Instanz die einzelnen Teile ihrer Umgebung. Das Vorgehen ist vergleichbar mit dem Zugriff des Clients auf JNDI. Der EJB-Container stellt der Bean jedoch den passenden *InitialContext* zur Verfügung, damit sie auf ihre eigene Umgebung zugreifen kann.

Die Enterprise-Edition der Java Version 2 (*J2EE*) schlägt ein Schema für die Namen der unterschiedlichen Services im Namensdienst vor. Die folgenden Namen sollten seit der Version 1.1 von EJB-konformen Servern verwendet werden:

- *java:comp/env/* – Umgebungsvariablen
- *java:comp/env/ejb* – Referenzen auf andere Bean-Klassen
- *java:comp/env/jdbc* – Zugriff auf JDBC-Datenbanken
- *java:comp/env/jms* – Zugriff auf den Java Messaging Service
- *java:comp/env/mail* – Zugriff auf den Mail
- *java:comp/env/url* – Zugriff auf Webserver und vergleichbare Dienste

Umgebungsvariablen

Der Bean-Entwickler kann eine nachträgliche Konfiguration einer Bean vorsehen. Dafür definiert er im Deployment-Deskriptor Umgebungsvariablen (*Beschreibung*, *Name der Variablen*, *Datentyp*), denen der Application-Assembler oder der Deployer *Werte* zuweisen muss (*Wert*). Nur bestimmte einfache Datentypen sind erlaubt: *String*, *Integer*, *Boolean*, *Double*, *Byte*, *Short*, *Long*, *Float*. Der Datentyp muss vollständig qualifiziert im Deployment-Deskriptor angegeben werden (Siehe Listing 4.7).

```
...
<enterprise-beans>
  <session>
    ...
    <env-entry>
      <description>
        Beschreibung
      </description>
      <env-entry-name>
        Name
      </env-entry-name>
      <env-entry-type>
        Datentyp
      </env-entry-type>
      <env-entry-value>
        Wert
      </env-entry-value>
      ...
    </env-entry>
  </session>
</enterprise-beans>
...
```

Listing 4.7: Schematischer Deployment-Deskriptor mit Definition einer Umgebungsvariablen

Zur Laufzeit kann eine Bean-Instanz mittels JNDI auf ihre Umgebungsvariablen zugreifen (siehe Listing 4.8). Die Variable ist unter ihrem Namen in dem Pfad *java:comp/env/* zu finden.

```
javax.naming.Context beanCtx =
  new javax.naming.InitialContext();
Datentyp something = (Datentyp)
  beanCtx.lookup("java:comp/env/Name");
```

Listing 4.8: Schematisches Auslesen einer Umgebungsvariablen

Referenzen

Eine Bean hat die Möglichkeit andere Beans zu verwenden. Mit der EJB-1.1-Version wurde die Verwaltung für diese Abhängigkeiten verbessert und die EJB-2.0-Version verstärkt den Einsatz des neuen Konzepts weiter. Der Bean-Entwickler spricht andere Beans nur mit einem logischen Namen an. Er definiert alle Referenzen auf andere Beans im Deployment-Deskriptor (*Beschreibung*, *ReferenzName*, *Beantyp* – *Session* oder *Entity*).

Erst später definiert der Application-Assembler welche Bean wirklich verwendet wird. Dafür erweitert er die Definitionen im Deployment-Deskriptor. Er benennt die referierte Bean (*ejb-link*). Hier wird der Name der Bean im Deployment-Deskriptor (wie in *ejb-name*) verwendet. Falls auf eine Bean in einem anderen Deployment-Deskriptor verwiesen wird, steht hier Pfad und Name dieses Deployment-Deskriptors angegeben, gefolgt von einem Doppelkreuz (#) und dem Namen der Bean (wie in *ejb-name*). Leider muss hier gesagt werden, dass das beschriebene Verfahren nicht von allen Herstellern von EJB-Containern gleichermaßen berücksichtigt wurde und teilweise proprietäre Schnittstellen verwendet werden müssen.

Listing 4.9 zeigt die Definitionen im Deployment-Deskriptor von Bean-Provider und Application-Assembler für eine Referenz auf eine Bean mit Remote-Interface. Listing 4.10 zeigt analog dazu die Definitionen für eine Referenz auf eine Bean mit Local-Interface.

```
...
<enterprise-beans>
  <session>
    ...
    <ejb-ref>
      <description>
        Beschreibung
      </description>
      <ejb-ref-name>
        ejb/ReferenzName
      </ejb-ref-name>
      <ejb-ref-type>
```

```
        Session oder Entity
      </ejb-ref-type>
      <home>
        HomeInterface
      </home>
      <remote>
        RemoteInterface
      </remote>
      <ejb-link>
        Name der referenzierten Bean
      </ejb-link>
    </ejb-ref>
    ...
  </session>
</enterprise-beans>
...
```

Listing 4.9: Schematischer Deployment-Deskriptor mit Referenz auf eine andere Bean (Remote-Interface)

```
...
<enterprise-beans>
  <session>
    ...
    <ejb-local-ref>
      <description>
        Beschreibung
      </description>
      <ejb-ref-name>
        ejb/ReferenzName
      </ejb-ref-name>
      <ejb-ref-type>
        Session oder Entity
      </ejb-ref-type>
      <local-home>
        LocalHomeInterface
      </local-home>
      <local>
        LocalInterface
      </local>
      <ejb-link>
        Name der referenzierten Bean
      </ejb-link>
    </ejb-local-ref>
    ...
  </session>
</enterprise-beans>
...
```

Listing 4.10: Schematischer Deployment-Deskriptor mit Referenz auf eine andere Bean (Local-Interface)

Zur Laufzeit kann die Bean-Instanz mittels JNDI auf die referierte Bean zugreifen. Das Home- bzw. Local-Home-Interface der referierten Bean ist unter ihrem logischen Namen in dem Pfad *java:comp/env/ejb* zu finden. Listing 4.11 verdeutlicht diesen Zusammenhang unter Verwendung einer Referenz auf eine Bean mit Remote-Interface. Die Vorgehensweise bei einer Referenz auf eine Bean mit Local-Interface ist analog; auf die Verwendung der Methode *narrow* kann in so einem Fall verzichtet werden.

```
...
javax.naming.Context beanCtx =
  new javax.naming.InitialContext();
Object temp = beanCtx.lookup(
  "java:comp/env/ejb/ReferenzName");
HomeInterface home = (HomeInterface)
  javax.rmi.PortableRemoteObject.narrow(
  temp, HomeInterface.class);
...
```

Listing 4.11: Schematische Verwendung einer Referenz auf eine andere Bean

Ressourcen

Der EJB-Container bietet den Zugriff auf unterschiedliche Ressourcen als Dienste an. Für jede Ressource stellt er eine sogenannte Ressource-Factory zur Verfügung, mit deren Hilfe die Bean Zugriff auf den Dienst erhält. Zur Verdeutlichung der Mechanismen werden wir im Folgenden den Umgang mit einer JDBC-Datenbank erläutern.

Eine Bean hat die Möglichkeit, eine Datenbank direkt anzusprechen. Damit die Bean in unterschiedlichen Anwendungen eingesetzt werden kann, wird seit der Version 1.1 hier eine flexible Konfiguration vorgesehen. Der Bean-Entwickler definiert nur einen logischen Namen für die Datenbank im Deployment-Deskriptor (*Name*, *Beschreibung*, *Sicherheitsstrategie* – *Application* oder *Container*). Außerdem kann der Entwickler angeben (über das optionale Element *res-sharing-scope*), ob die Verbindungen zur Datenbank aus dieser Resource mit anderen Enterprise-Beans geteilt werden können, oder nicht. Der Deployer verwendet Werkzeuge des EJB-Containers, um später die konkrete Datenbank zu bestimmen.

```
...
<enterprise-beans>
  <session>
    ...
    <resource-ref>
      <description>
        Beschreibung
      </description>
      <res-ref-name>
        jdbc/ReferenzName
```

```
        </res-ref-name>
        <res-type>
          javax.sql.DataSource
        </res-type>
        <res-auth>
          Application oder Container
        </res-auth>
        <res-sharing-scope>
          Shareable oder Unshareable
        </res-sharing-scope>
      </resource-ref>
        ...
    </session>
  </enterprise-beans>
  ...
```

Listing 4.12: Schematischer Deployment-Deskriptor mit Referenz auf eine JDBC-Datenbank

Der Deployer verwendet Werkzeuge des EJB-Containers, um später die konkrete Datenbank zu bestimmen, die der Referenz in Listing 4.12 zugewiesen wird. Wenn der Applikations-Server auch die zusätzlichen Ressourcen mit JNDI verwaltet, muss nur noch der logische Name auf den richtigen JNDI-Namen abgebildet werden. Listing 4.13 zeigt, wie dies in der Konfigurations-Sprache eines Applikations-Servers aussehen könnte.

```
<reference-descriptor>
  <resource-description>
    <res-ref-name>
      jdbc/ReferenzName
    </res-ref-name>
    <jndi-name>
      JNDI-Name-Datenbank
    </jndi-name>
  </resource-description>
</reference-descriptor>
```

Listing 4.13: Abbildung des logischen Namens auf den JNDI-Namen der Ressource

Zur Laufzeit kann die Bean mittels JNDI-Zugriff auf die referierte JDBC-Datenbank erhalten. Die Ressource-Factory ist unter dem definierten logischen Namen in dem Pfad *java:comp/env/jdbc* zu finden. Die Namensdefinition im Deployment-Deskriptor ist relativ zu *java:com/env,* der Umgebung der Bean. Die Ressource-Factory ist ein Objekt der Klasse *javax.sql.DataSource*. Diese Klasse muss bei jedem EJB-konformen Server vorhanden sein, der die Version 1.1 oder 2.0 unterstützt. Sie ist auch Bestandteil von J2EE oder mit der optionalen Erweiterung zu JDBC 2.0 verfügbar.

```
javax.naming.Context beanCtx =
  new javax.naming.InitialContext();
javax.sql.DataSource ds =
  (javax.sql.DataSource)
  beanCtx.lookup(
  "java:comp/env/jdbc/ReferenzName");
java.sql.Connection con =
  ds.getConnection();
```

Listing 4.14: Schematische Verwendung einer Referenz auf eine JDBC-Datenbank

Administrierte Objekte

Seit der Version 2.0 gibt es die Möglichkeit, sogenannte administrierte Objekte (*administered objects*) in den Deployment Deskriptor einzutragen. Administrierte Objekte sind z.B. JMS-Queues oder JMS-Topics. Verwendet eine Enterprise-Bean JMS (Java Message Service), um z.B. Nachrichten über einen Topic *Ereignis-Topic* zu senden, so braucht sie Zugang zu einer Topic-Factory und zu angesprochenem Topic. Die Topic-Factory ist eine Ressource-Factory, der Topic ist ein sogenanntes administriertes Objekt. Listing 4.15 zeigt, wie ein solcher Eintrag im Deployment-Deskriptor vorzunehmen ist. Kapitel 6 wird sich ausführlich mit dem Themenkomplex rund um den Java Message Service auseinandersetzen und erklären, was eine Topic-Factory und ein Topic ist und wie sie zu verwenden sind.

```
...
<enterprise-beans>
  <session>
    ...
    <resource-ref>
      <description>
        Beschreibung
      </description>
      <res-ref-name>
        jdbc/TopicFactory
      </res-ref-name>
      <res-type>
        javax.jms.TopicConnectionFactory
      </res-type>
      <res-auth>
        Container
      </res-auth>
    </resource-ref>
    <resource-env-ref>
      <resource-env-ref-name>
        jms/EreignisTopic
      </resource-env-ref-name>
      <resource-env-ref-type>
        javax.jms.Topic
      </resource-env-ref-type>
```

```
      </resource-env-ref>
    </session>
    ...
  </enterprise-beans>
  ...
```

Listing 4.15: Verwendung einer Referenz auf administrierbare Objekte

4.4 Beispiele

Dieser Abschnitt soll den einfachen Einstieg in die Programmierung von Session-Beans ermöglichen. Es werden zwei einfache Beispiele vorgestellt. Das erste zeigt die Implementierung einer zustandslosen Session-Bean. Das zweite zeigt eine zustandsbehaftete Session-Bean.

Beide Beispiele verwenden JDBC (Java Database Connectivity) für den Zugriff auf eine Datenbank. JDBC ist kein Schwerpunkt dieses Buches. In den folgenden Beispielen wird der Programmcode für den Datenbankzugriff nur kurz vorgestellt. Für eine schnelle Einarbeitung in JDBC empfehlen wir die Tutorials von Sun Microsystems [JDBC Sun Microsystems, 1997].

Die folgenden Beispiele sollten unverändert auch mit EJB 1.1 laufen. Falls sie für EJB 1.0 angepasst werden sollen, müssen einige Änderungen vorgenommen werden:

- In der Bean-Klasse muss anstelle der *java.ejb.EJBException* eine *java.rmi.RemoteException* verwendet werden, um einen Systemfehler zu melden.
- EJB 1.0 kennt das Konzept der Resource-Factory-Reference noch nicht. Die Datenbankverbindung muss deshalb unter der Verwendung des richtigen Datenbanktreibers direkt geöffnet werden.
- Der Deployment-Deskriptor von EJB 1.0 bestand aus serialisierten Objekten und nicht aus XML. Eine entsprechende Anpassung ist hier erforderlich.

4.4.1 Zustandslose Session-Beans

Überblick

Schritt für Schritt wollen wir eine einfache Session-Bean entwickeln. Als Beispiel wurde die Umrechnung zwischen Euro und anderen Währungen gewählt. Der symbolische Name der Bean ist *EuroExchangeSL* (SL steht für stateless, also zustandslos).

Die Bean bietet einen Dienst auf dem Server an, der die Umrechnung mit Hilfe einer Kurstabelle übernimmt. Die Tabelle mit den Wechselkursen liegt in der Datenbank. Für Kauf und Verkauf der Währungen werden zwei unterschiedliche Kurse verwendet. Die Bean selbst wird keinen Conversational-State besitzen. Bei jeder Anfrage greift sie auf die Wechselkurse in der Datenbank zu.

Im Folgenden werden wir zuerst das Home- und das Remote-Interface definieren. Wir werden keine gesonderte Implementierung für Local-Interfaces darstellen, da die Unterschiede in der Programmierung sehr gering sind. Wir weisen an den einschlägigen Stellen auf die Unterschiede hin. Im Anschluss daran bereiten wir die Datenbank vor und implementieren die Bean-Klasse selbst. Danach sehen wir uns den Deployment-Deskriptor an. Zum Schluss wird noch ein einfacher Client zum Testen der Bean entwickelt.

Interfaces

Um die Funktionalität der Bean festzulegen, definieren wir zu Beginn die Interfaces, die dem Client zur Verfügung stehen. Oft ist der Programmcode des Home- und Remote-Interface mit ausführlichen Kommentaren für Programmierer eine hinreichende Spezifikation der Bean.

Das Home- und Remote-Interface wird auf dem Clientrechner verwendet, und fast alle Methodenaufrufe werden über das Netz zum Server übertragen. Deshalb kann jede Methode in diesen Interfaces eine *RemoteException* auslösen. Das Auftreten dieser Exception signalisiert dem Client ein Netzwerkproblem oder einen Systemfehler auf dem Server.

Das Home-Interface erbt vom Interface *javax.ejb.EJBHome*. Session-Beans definieren im Home-Interface zusätzlich zu den geerbten Methoden nur *create*-Methoden. Bei zustandslosen Session-Beans existiert immer nur eine *create*-Methode ohne Parameter. Jede *create*-Methode deklariert zusätzlich zur *java.rmi.RemoteException* die *javax.ejb.CreateException*.

```
package ejb.exchangeSL;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;


public interface EuroExchangeHome extends EJBHome {

    public EuroExchange create()
        throws RemoteException, CreateException;

}
```

Listing 4.16: Das Home-Interface der zustandslosen Session-Bean EuroExchangeSL

Würde die Session-Bean ein Local-Interface verwenden, würde *EuroExchangeHome* (siehe Listing 4.16) von *EJBLocalHome* statt von *EJBHome* erben. Da die Kommunikation mit der Session-Bean im gleichen Java-Prozess stattfände, würde auch an keiner Methode die Ausnahme *RemoteException* deklariert.

Das Remote-Interface (siehe Listing 4.17) definiert die Methoden der Anwendungslogik, auf die der Client zugreifen kann. Unsere Bean hat zwei Methoden zum Umrechnen von Währungen. Die Methode *changeFromEuro* konvertiert Euro in andere Währungen. Die Methode *changeToEuro* nimmt die umgekehrte Berechnung vor.

Zusätzlich gibt es eine Methode *setExchangeRate*, mit deren Hilfe neue Wechselkurse in die Tabelle aufgenommen werden können. Um das Beispiel einfach und übersichtlich zu halten, wird jedoch keine vollständige Verwaltung von Umrechnungskursen implementiert. Diese Methode soll lediglich einen einfachen Weg zum Testen der Bean ermöglichen.

```
package ejb.exchangeSL;

import java.rmi.RemoteException;
import javax.ejb.EJBObject;


public interface EuroExchange extends EJBObject {

    public float changeFromEuro(String currency, float amount)
        throws RemoteException;

    public float changeToEuro(String currency, float amount)
        throws RemoteException;

    public void setExchangeRate(String currency,
                                float euro,
                                float foreignCurr)
        throws RemoteException;

}
```

Listing 4.17: Remote-Interface der zustandslosen Session-Bean EuroExchangeSL

Würde die Session-Bean ein Local-Interface verwenden, würde *EuroExchange* von *EJBLocalObjet* statt von *EJBObject* erben. Da die Kommunikation mit der Session-Bean im gleichen Java-Prozess stattfände, würde auch an keiner Methode die Ausnahme *RemoteException* deklariert.

Datenbank

Das Beispiel greift mittels JDBC auf eine Datenbank zu. Die Datenbank enthält die Tabelle mit Umtauschkursen. Die Tabelle ist einfach gehalten und wird beispielsweise mit den folgenden SQL-Befehlen erzeugt:

```
CREATE TABLE EURO_EXCHANGE
(CURRENCY CHAR(10) NOT NULL,
 EURO REAL ,
 FOREIGNCURR REAL )
ALTER TABLE EURO_EXCHANGE ADD CONSTRAINT
 EURO_PRIM PRIMARY KEY (CURRENCY)
```

Listing 4.18: SQL-Befehle für die Tabelle mit Umtauschkursen

Enterprise-Bean

Für die Implementierung der Bean-Klasse benutzen wir Interfaces, Klassen und Exceptions von EJB. Deshalb werden immer Teile des *ejb*-Packages importiert. Die Verwendung des Namensdiensts erfordert das *naming*-Package. Beim Zugriff auf die Datenbank wird JDBC verwendet, wodurch das Importieren von Teilen des *sql*-Packages erforderlich wird. Zusätzlich dazu wird die Klasse *DataSource* aus dem Package *javax.sql* als Ressource-Factory für Datenbankverbindungen verwendet. Die folgende Auflistung fasst die nötigen *import*-Anweisungen für die Implementierung der Bean-Klasse zusammen:

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

import javax.ejb.CreateException;
import javax.ejb.EJBException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.sql.DataSource;
```

Da wir eine Session-Bean schreiben, implementiert unsere Bean-Klasse das Interface *javax.ejb.SessionBean*. Jede Methode aus dem Interface muss in der Bean-Klasse implementiert werden. Alle Bean-Klassen müssen *public* sein und dürfen keine *finalize-Methode* definieren. Der Name der Bean-Klasse muss nicht dem Namen entsprechen, unter dem sie im Namensdienst zu finden ist.

Von der Session-Bean wird ein Zustand in Instanzvariablen gehalten, der beim Erzeugen einer Instanz initialisiert wird. Es handelt sich dabei um einen rein technischen Zustand, der nicht den Zustand einer Client-Session wiedergibt, wie es bei zustandsbehafteten Session-Beans möglich ist (siehe Listing 4.19). In *beanCtx* wird der Context der Bean abgelegt und in *dataSource* ein Objekt der Klasse *javax.sql.DataSource*, bei dem es sich um eine Ressource-Factory für Datenbankverbindungen zu einer definierten Datenbank handelt. In der Konstanten *dbRef* wir der Name dieser Ressource-Factory im Namensdienst vermerkt.

```
public class EuroExchangeBean implements SessionBean {

    public static final String dbRef = "java:comp/env/jdbc/EuroDB";

    private SessionContext beanCtx    = null;
    private DataSource     dataSource = null;
    ...

}
```

Listing 4.19: Klassendefinition für EuroExchangeBean

Zuerst werden wir die Erstellung einer Bean-Instanz betrachten. Der EJB-Container ruft die Methode *ejbCreate* bei jeder neuen Bean-Instanz auf. Bei einer zustandslosen Session-Bean hat die Methode *ejbCreate* die Aufgabe, die Bean-Instanz zu initialisieren. Die Bean-Instanz reserviert sich Ressourcen, die sie für die Arbeit benötigt. Es sei nochmals erwähnt, dass eine zustandslose Bean-Instanz keinen Conversational-State hat und abwechselnd von unterschiedlichen Clients verwendet werden kann. Die Initialisierung einer zustandslosen Session-Bean beschränkt sich deshalb auf Ressourcen, die alle Clients teilen.

In dem gegebenen Beispiel (siehe Listing 4.20) wird der initiale Kontext des Namensdienstes erzeugt. Dann wird die Ressource-Factory für die benötigte Datenbank aus dem Namensdienst gelesen. Letztere bleibt während der gesamten Lebensdauer der Bean erhalten.

```
...
    public void ejbCreate()
        throws CreateException
    {
        try {
            Context c = new InitialContext();
            this.dataSource = (DataSource)c.lookup(dbRef);
        } catch(NamingException ex) {
            String msg = "Cannot get Resource-Factory:"
                        + ex.getMessage();
            throw new CreateException(msg);
        }
    }
...
```

Listing 4.20: EuroExchangeBean.ejbCreate()

Das Gegenstück zur Methode *ejbCreate* ist die Methode *ejbRemove*. Der EJB-Container ruft diese Methode auf, um sie von ihrer bevorstehenden Löschung zu unterrichten. In der Methode muss sichergestellt werden, dass alle Ressourcen, die die Bean-Instanz

verwendet, wieder freigegeben werden. In dem Beispiel wird die Referenz auf die Resource-Factory, die in der Methode *ejbCreate* erstellt wurde, wieder freigegeben.

```
...
    public void ejbRemove() {
        this.dataSource = null;
    }
...
```

Listing 4.21: EuroExchangeBean.ejbRemove()

Das hier diskutierte, einfache Beispiel verwendet nicht den Session-Kontext, den es vom EJB-Container übergeben bekommt. Um spätere Erweiterungen zu unterstützen wird jedoch der Session-Kontext in der Methode *setSessionContext* gespeichert. Die Methoden *ejbActivate* und *ejbPassivate* müssen implementiert werden, um dem Interface *Session-Bean* gerecht zu werden. Die Implementierung bleibt jedoch leer, da zustandslose Session-Beans weder aktiviert noch passiviert werden.

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

    public void ejbActivate() { }

    public void setSessionContext(SessionContext ctx) {
        this.beanCtx = ctx;
    }
...
```

Listing 4.22: Methoden für das Zustandsmanagement

Die Implementierung der Anwendungsmethoden ist jetzt vergleichsweise einfach. Durch das Zusammenspiel von *ejbCreate* und *ejbRemove* sind die Anwendungsmethoden von der Verantwortung für den Namensdienst und die *DataSource* entlastet.

Die Methode *changeFromEuro* hat die Aufgabe, einen Betrag in Euro in eine andere Währung umzurechnen. Zu Beginn muss eine Datenbankverbindung mit Hilfe der Ressource-Factory *DataSource* erzeugt werden. Dann wird ein *Statement* für die Ausführung von Datenbankzugriffen erzeugt. Das Ergebnis wird in einem *ResultSet* zurückgegeben. Am Ende der Methode werden *Connection*, *Statement* und *ResultSet* im *finally*-Block wieder mit *close* geschossen.

Der Betrag und die gewünschte Währung werden als Parameter übergeben. Die Methode liest den erforderlichen Wechselkurs aus der Datenbank, errechnet den Wert und gibt diesen zurück. Im Fehlerfall wird eine *EJBException* ausgelöst. Der EJB-Container gibt diese als *RemoteException* an den Client weiter. Würde die Bean ein Local-Interface verwenden, gäbe der EJB-Container die EJBException selbst weiter.

Listing 4.23 zeigt die Implementierung. Die Methode *changeToEuro* arbeitet entsprechend und wird deshalb hier nicht gezeigt.

```
...
    public float changeFromEuro(String currency, float amount) {
        if(currency == null) {
            throw new EJBException("illegal argument: currency");
        }
        final String query =
          "SELECT FOREIGNCURR FROM EURO_EXCHANGE WHERE CURRENCY=?";
        Connection        con = null;
        PreparedStatement st  = null;
        ResultSet         rs  = null;
        try {
            con = this.dataSource.getConnection();
            st  = con.prepareStatement(query);
            st.setString(1, currency);
            rs = st.executeQuery();
            if(!rs.next()) {
                throw new EJBException("no such currency:"
                                       + currency);
            }
            return amount * rs.getFloat(1);
        } catch(SQLException ex) {
            ex.printStackTrace();
            throw new EJBException("db-error:" + ex.getMessage());
        } finally {
            try { rs.close();  } catch(Exception ex) {}
            try { st.close();  } catch(Exception ex) {}
            try { con.close(); } catch(Exception ex) {}
        }
    }
...
```

Listing 4.23: EuroExchangeBean.changeFromEuro(...)

Mit *setExchangeRate* soll nur der einfache Test des Beispiels ermöglicht werden. Die Methode kann verwendet werden, um Wechselkurse in die Datenbank zu schreiben. Im Fehlerfall wird wieder eine *EJBException* ausgelöst. Listing 4.24 zeigt die Implementierung.

```
...

    public void setExchangeRate(String currency,
                                float  euro,
                                float  foreignCurr)
    {
```

```
        if(currency == null) {
            throw new EJBException("illegal argument: currency");
        }
        final String delQuery =
            "DELETE FROM EURO_EXCHANGE WHERE CURRENCY=?";
        final String insQuery =
            "INSERT INTO EURO_EXCHANGE" +
            "(CURRENCY, EURO, FOREIGNCURR) VALUES(?, ?, ?)";
        Connection        con = null;
        PreparedStatement del = null;
        PreparedStatement ins = null;
        boolean success = false;
        try {
            con = this.dataSource.getConnection();
            con.setAutoCommit(false);
            del = con.prepareStatement(delQuery);
            del.setString(1, currency);
            del.executeUpdate();
            ins = con.prepareStatement(insQuery);
            ins.setString(1, currency);
            ins.setFloat(2, euro);
            ins.setFloat(3, foreignCurr);
            ins.executeUpdate();
            success = true;
        } catch(SQLException ex) {
            ex.printStackTrace();
            throw new EJBException("db-error:" + ex.getMessage());
        } finally {
            if(success) {
                try { con.commit(); } catch(Exception ex) {}
            } else {
                try { con.rollback(); } catch(Exception ex) {}
            }
            try { del.close(); } catch(Exception ex) {}
            try { ins.close(); } catch(Exception ex) {}
            try { con.setAutoCommit(true); } catch(Exception ex) {}
            try { con.close(); } catch(Exception ex) {}
        }
    }
```

Listing 4.24: EuroExchangeBean.setExchangeRate(...)

Damit ist die Implementierung der Bean-Klasse vollständig. Alle Methoden für die Identitätenverwaltung, das Zustandsmanagement und die Anwendungslogik wurden implementiert.

Deployment

Damit die Bean von einem Client verwendet werden kann, muss sie im EJB-Container installiert werden (*Deployment*). Dazu werden im Deployment-Deskriptor in einer XML-Syntax Informationen über die Bean zusammengestellt (siehe Listing 4.25). Der Bean-Entwickler, der Application-Assembler und der Deployer sind dafür verantwortlich. Die meisten Hersteller von Applikationsservern bieten dazu entsprechende Tools an, die die Erstellung solcher Deployment-Deskriptoren durch graphische Oberflächen erleichtern. Es ist jedoch gängige Praxis, Deployment-Deskriptoren per Hand (mit einem Text-Editor) zu erstellen.

Der Bean-Entwickler muss im Deployment-Deskriptor nur Angaben machen, die der EJB-Container nicht mittels der Untersuchung der Klassen (*Introspection*) selbst ermitteln kann. Zuerst definiert er, dass es sich um eine Session-Bean handelt (*session*), und gibt der Bean einen eindeutigen Namen (*ejb-name*). Wenn der Deployer die Bean später im EJB-Container installiert, wird dieser Name häufig verwendet, um die Bean beim Namensdienst anzumelden.

Dann gibt er die Namen der benötigten Klassen und Interfaces an. Er definiert das Home-Interface (*home*) und das Remote-Interface (*remote*) sowie die Implementierungsklasse der Bean (*ejb-class*). Würde die Bean Local-Interfaces verwenden, so würde statt *home* das Element *local-home*, statt *remote* das Element *local* verwendet, um die Klasse für das Local-Home- und das Local-Interface zu deklarieren.

Der EJB-Container benötigt die Information, ob es sich um eine zustandslose oder zustandsbehaftete Session-Bean handelt (*session-type: Stateful / Stateless*). In unserem einfachen Beispiel übernimmt die Bean die Transaktionssteuerung (*transaction-type*) selbst.

Außerdem muss der Bean-Entwickler die Referenz auf die benötigte Datenbank definieren (*resource-ref*). Er definiert einen logischen Namen (*res-ref-name*) relativ zur Umgebung der Bean (*java:comp/env*) und legt den Datentyp der Ressource-Factory fest (*res-type*) und bestimmt, dass der EJB-Container die Sicherheitsstrategie festlegen soll (*res-auth*). Zusätzlich beschreibt er in einem Kommentar die Aufgabe dieser Referenz (*description*), um dem Deployer seine Arbeit zu ermöglichen.

Die Aufgabe des Bean-Providers ist jetzt abgeschlossen. Er hat alle erforderlichen Informationen über die Bean bereitgestellt. Der Application-Assembler könnte die Berechtigungen für den Zugriffsschutz definieren und das Zusammenspiel mit anderen Beans festlegen. Für unser einfaches Beispiel sind diese Definitionen jedoch nicht unbedingt erforderlich.

```
<?xml version="1.0" ?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0/
/EN" "http://java.sun.com/j2ee/dtds/ejb-jar_2_0.dtd">
```

```
<ejb-jar>
  <description>
    Dieser Deployment-Deskriptor enthält Informationen
    über die Session-Bean Exchange.
  </description>
  <enterprise-beans>
    <session>
      <ejb-name>Exchange</ejb-name>
      <home>ejb.exchangeSL.EuroExchangeHome</home>
      <remote>ejb.exchangeSL.EuroExchange</remote>
      <ejb-class>ejb.exchangeSL.EuroExchangeBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Bean</transaction-type>
      <resource-ref>
        <description> Euro-Datenbank </description>
        <res-ref-name>jdbc/EuroDB</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
      </resource-ref>
    </session>
  </enterprise-beans>

  <assembly-descriptor>
  </assembly-descriptor>

</ejb-jar>
```

Listing 4.25: Deployment-Deskriptor der Session-Bean EuroExchangeSL

Für das Deployment werden in der Regel weitere Informationen benötigt, die sich von Applikationsserver zu Applikationsserver unterscheiden. So muss z. B. festgelegt werden, unter welchem Namen die Bean im JNDI zu finden ist. Die Ressource-Referenz muss auf eine existierende Datenbank abgebildet werden, die die benötigten Tabellen enthält. In vielen Fällen ist es nötig, dem EJB-Container Parameter für die Laufzeitumgebung mitzuteilen, z. B. die Größe des Instanzenpools, Cache-Größen oder Timeout-Werte. Art und Umfang solcher zusätzlichen Angaben unterscheiden sich wie gesagt von Produkt zu Produkt, ebenso die Art, wie diese Angaben gemacht werden müssen. Listing 4.26 zeigt ein fiktives Beispiel, wie zusätzliche Angaben zum Deloyment der EuroExchange-Bean gemacht werden könnten.

```
<bean>
  <ejb-name>Exchange</ejb-name>
  <stateless-session-descriptor>
    <max-pool-size>100</max-pool-size>
  </stateless-session-descriptor>
  <resource-reference-description>
    <res-ref-name>jdbc/EuroDB</res-ref-name>
    <jndi-name>jdbc/Oracle_Test_DB</jndi-name>
```

```
    </resource-reference-description>
    <jndi-name>
      ExchangeSL
    </jndi-name>
  <bean>
```

Listing 4.26: Definitionen für das Deployment eines EJB-Containers

Client

Zu der Session-Bean *EuroExchangeSL* wird ein einfacher Client entwickelt (siehe Listing 4.27). Dieser definiert den Wechselkurs für US-Dollar. Danach wechselt er einen Betrag US-Dollar in Euro und wieder zurück in US-Dollar. Unser Client benötigt Zugriff auf den Namensdienst *javax.naming* und muss mit Netzwerkfehlern umgehen können (*javax.rmi.RemoteException)*. Außerdem benötigt der Client die Interfaces der Bean, mit der er arbeiten möchte. Diese Interfaces haben wir im Package *ejb.exchangeSL* zusammengefasst. Der JNDI-Name der Bean ist *ExchangeSL*. Dieser Name wurde der Bean durch die zusätzlich gemachten Deployment-Angaben in Listing 4.26 zugewiesen.

```
package ejb.exchangeSL.client;

import ejb.exchangeSL.EuroExchange;
import ejb.exchangeSL.EuroExchangeHome;

import java.rmi.RemoteException;

import javax.ejb.CreateException;
import javax.ejb.RemoveException;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.rmi.PortableRemoteObject;


public class Client {

    private EuroExchange exchange;

    public Client() { }

    public void init()
        throws NamingException, RemoteException, CreateException
    {
        java.util.Properties p = new java.util.Properties();
        p.put(Context.INITIAL_CONTEXT_FACTORY,
              "JNDI-PROVIDER-CLASS");
        p.put(Context.PROVIDER_URL, "JNDI-URL");
        Context ctx = new InitialContext(p);
```

```
        Object o = ctx.lookup("ExchangeSL");
        EuroExchangeHome home =
            (EuroExchangeHome)PortableRemoteObject.narrow(o,
                              EuroExchangeHome.class);
        this.exchange = home.create();
    }

    public void run() {
        try {
            this.exchange.setExchangeRate("US-Dollar", 2f, 0.5f);
        } catch(RemoteException ex) {
            ex.printStackTrace();
            return;
        }
        System.out.println("Changing 100 US-Dollars to Euro");
        float amount = 0f;
        try {
            amount = this.exchange.changeToEuro("US-Dollar", 100);
        } catch(RemoteException ex) {
            ex.printStackTrace();
            return;
        }
        System.out.println("Result: " + amount);

        System.out.println("Changing " + amount +
                           " Euro to US-Dollars");
        float n_amount = 0f;
        try {
            n_amount =
                this.exchange.changeFromEuro("US-Dollar", amount);
        } catch(RemoteException ex) {
            ex.printStackTrace();
            return;
        }
        System.out.println("Result: " + n_amount);
    }

    public void cleanUp() {
        try {
            this.exchange.remove();
        } catch(RemoteException ex) {
            ex.printStackTrace();
        } catch(RemoveException ex) {
            ex.printStackTrace();
        }
    }

    public static void main(String[] args) {
        Client c = new Client();
        try {
            c.init();
```

```
            } catch(Exception ex) {
                ex.printStackTrace();
                return;
            }
            c.run();
            c.cleanUp();
        }

    }
```

Listing 4.27: Clientprogramm für Session-Bean EuroExchangeSL

Würde die Bean Local- statt Remote-Interfaces verwenden, würde sich von der Client-Programmierung her nicht viel ändern. Der Client müsste *PortableRemoteObject.narrow* nur dann verwenden, wenn der Namensdienst nicht im Prozess des Applikationsservers laufen würde. In der Regel wird der Namensdienst des Applikationsservers verwendet. Wenn der Client Local-Interfaces verwendet, muss er sich im selben Prozess befinden wie die Enterprise-Bean, also im Prozess des Applikationsservers. Er könnte in diesem Fall den Rückgabewert der Methode lookup durch ein Type-Casting direkt in den gewünschten Typ konvertieren. Außerdem entfällt im Fall von Local-Interfaces die Deklaration der *java.rmi.RemoteException*. Wie bereits erwähnt, liegt der wesentliche Unterschied in der Verwendung von Local-Interfaces weniger in der Programmierung, sondern eher im Laufzeitverhalten. Zum einen wird der RMI-Overhead umgangen, weil sich der Client im gleichen Prozess befindet, zum anderen werden Aufrufparameter und Rückgabewerte nicht call-by-value sondern call-by-reference übergeben (mit Ausnahme der primitiven Datentypen von Java, die immer call-by-value übergeben werden).

Zusammenfassung

In diesem Beispiel wurde ein einfacher Dienst auf dem Server realisiert. Die zustandslose Session-Bean geht sparsam mit den Systemressourcen um. Eine zustandslose Session-Bean kann nacheinander mehrere Clients bedienen. Beim Deployment kann festgelegt werden, wie viele Bean-Instanzen einer Klasse maximal vom EJB-Container verwendet werden. Bei der Implementierung einer zustandslosen Session-Bean müssen diese Tatsachen immer berücksichtigt werden. Der Zustand der Bean-Instanz muss allgemein für alle Zugriffe der Clients gelten.

Obwohl die Bean viele parallele Clients bedienen kann, muss sie nicht mit parallelen Zugriffen der Clients umgehen. Der EJB-Container serialisiert alle Methodenaufrufe. Dadurch entfallen die Probleme der Synchronisation paralleler Threads vollständig. Dies ermöglicht dem Anwendungsentwickler, sich bei der Programmierung auf die eigentliche Aufgabenstellung zu konzentrieren, da er von den Problemen der Serverprogrammierung weitgehend entlastet ist.

Es wurde eine Ressource-Factory verwendet, um Zugriff auf die Datenbank mit den Wechselkursen zu erhalten. Mit dieser wurde vor jede Berechnung eine neue Datenbankverbindung erzeugt und danach wieder geschlossen. Diese Implementierung geht davon aus, dass der EJB-Container dabei nicht immer eine neue Datenbanksitzung und eine neue Netzwerkverbindung öffnet und wieder schließt, sonder die Datenbankverbindungen in einem Pool verwaltet und wiederverwendet. Die meisten Applikations-Server verfolgen bei Datenbankverbindungen eine entsprechende Strategie. Allgemein müssen Sie für den effizienten Umgang mit Ressourcen, die von Ihrem Applikations-Server verwendeten Verfahren genau kennen.

4.4.2 Zustandsbehaftete Session-Beans

Überblick

In diesem Abschnitt soll eine zustandsbehaftete Session-Bean entwickelt werden. Dazu wird die Funktionalität wieder aufgegriffen, die aus dem vorhergehenden Beispiel bekannt ist. Der symbolische Name der Bean ist *EuroExchangeSF* (SF steht für stateful, also zustandsbehaftet).

Die Lösung mit einer zustandsbehafteten Session-Bean erlaubt ein anderes Vorgehen. Dieses Beispiel nutzt die Identität von zustandsbehafteten Session-Beans. Jeweils eine Session-Bean wird verwendet, um die Umrechnungen von Euro in eine bestimmte andere Währung zu realisieren. Der Conversational-State speichert die Wechselkurse der Währung. Dadurch wird vermieden, dass für jede Umrechnung ein Datenbankzugriff erfolgt.

Bei der Erzeugung der Bean wird ihr mitgeteilt, mit welcher Währung sie arbeiten soll. Den benötigten Wechselkurs holt sie sich aus der Datenbanktabelle, die auch im vorhergehenden Beispiel verwendet wurde. Zu einem späteren Zeitpunkt kann die Zuordnung der Währung vom Client auch wieder geändert werden.

Dieses Beispiel für zustandsbehaftete Session-Beans baut auf dem vorhergehenden Beispiel für zustandslose Session-Beans auf. Auf zusätzliche Erläuterungen bezüglich des Local-Client-View wird verzichtet, dass sie identisch mit denen im vorangegangenen Beispiel sind.

Interfaces

Das Home-Interface der zustandsbehafteten Session-Bean definiert eine *create*-Methode mit Parametern, die den Conversational-State initialisieren. In unserem Beispiel (siehe Listing 4.28) wird die Währung übergeben, für deren Umrechnung die Session-Bean-Instanz zuständig sein soll.

```
package ejb.exchangeSF;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;

public interface EuroExchangeHome extends EJBHome {

    public EuroExchange create(String foreignCurr)
        throws RemoteException, CreateException;

}
```

Listing 4.28: Home-Interface der zustandsbehafteten Session-Bean EuroExchangeSF

Das Remote-Interface (siehe Listing 4.29) definiert drei Methoden. Mit *setForeignCurr* kann eine neue Währung ausgewählt werden. Der in der *create*-Methode übergebene Wert wird damit überschrieben. Die Funktionalität der beiden anderen Methoden *changeToEuro* und *changeFromEuro* ist aus dem vorhergehenden Beispiel bekannt.

```
package ejb.exchangeSF;

import java.rmi.RemoteException;
import javax.ejb.EJBObject;

public interface EuroExchange extends EJBObject {

    public void  setForeignCurr(String foreignCurr)
        throws RemoteException;

    public float changeToEuro(float amount)
        throws RemoteException;

    public float changeFromEuro(float amount)
        throws RemoteException;

}
```

Listing 4.29: Remote-Interface der zustandsbehafteten Session-Bean EuroExchangeSF

Bean

Die Klassendefinition (siehe Listing 4.30) der zustandsbehafteten Session-Bean ist der Klassendefinition der zustandslosen Session-Bean sehr ähnlich. Der entscheidende Unterschied liegt in den Instanzvariablen, die den Conversational-State bilden. Die Variable *foreignCurr* enthält die Währung, *euroRate* und *currRate* enthalten die zugehörigen Umtauschkurse. Der Inhalt dieser Variablen steht exklusiv nur dem einen Client der Bean zur Verfügung.

```
package ejb.exchangeSF;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

import javax.ejb.CreateException;
import javax.ejb.EJBException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.sql.DataSource;

public class EuroExchangeBean implements SessionBean {

    public static final String dbRef = "java:comp/env/jdbc/EuroDB";

    private String foreignCurr;
    private float  euroRate;
    private float  currRate;

    private SessionContext beanCtx = null;

    private transient DataSource dataSource = null;

    ...

}
```

Listing 4.30: EuroExchangeBean

Nach dem Erzeugen einer neuen Bean-Instanz ruft der EJB-Container zuerst die Methode *setSessionContext* und danach die Methode *ejbCreate* auf. Die Aufrufe finden als direkte Reaktion auf den Aufruf der Methode *create* des Clients statt.

Mit dem Aufruf der Methode *setSessionContext* (siehe Listing 4.31) gibt der EJB-Container der Bean-Instanz ihren Session-Kontext bekannt. Üblicherweise speichert die Bean den Context in einer Instanzvariablen.

```
...
    public void setSessionContext(SessionContext ctx) {
        this.beanCtx = ctx;
    }
...
```

Listing 4.31: EuroExchangeBean.setSessionContext

Die Methode *ejbCreate* (siehe Listing 4.32) kann neben dem Initialisieren des Conversational-State zusätzlich Ressourcen reservieren. Sie benutzt die Methode *readCurrencyValues*, auf die wir später noch im Detail zurückkommen. *ReadCurrencyValues* liest mit Hilfe der als Parameter übergebenen Währung die Wechselkurse aus der Datenbank und speichert sie in den entsprechenden Variablen.

```
...
    public void ejbCreate(String foreignCurr)
        throws CreateException
    {
        if(foreignCurr == null) {
            throw new CreateException("foreignCurr is null");
        }
        try {
            Context c = new InitialContext();
            this.dataSource = (DataSource)c.lookup(dbRef);
        } catch(NamingException ex) {
            String msg = "Cannot get Resource-Reference:"
                        + ex.getMessage();
            throw new CreateException(msg);
        }
        this.foreignCurr = foreignCurr;
        this.euroRate    = 0f;
        this.currRate    = 0f;
        try {
            this.readCurrencyValues();
        } catch(EJBException ex) {
            throw new CreateException(ex.getMessage());
        }
    }
...
```

Listing 4.32: EuroExchangeBeanBean.create(...)

Wenn der Client die Methode *remove* am Remote- bzw. Local-Interface aufruft, wird vor dem Löschen der Bean-Instanz vom EJB-Container die Methode *ejbRemove* aufgerufen (vgl. Listing 4.33). Eventuell reservierte Ressourcen müssen hier wieder freigegeben werden.

```
...
    public void ejbRemove() {
        this.dataSource  = null;
        this.foreignCurr = null;
        this.euroRate    = 0f;
        this.currRate    = 0f;
    }
...
```

Listing 4.33: EuroExchangeBeanBean.remove()

Zustandsbehaftete Session-Beans werden aktiviert und passiviert. Der EJB-Container ruft die zugehörigen Methoden auf, um die Bean über diesen Vorgang zu benachrichtigen. Die Implementierung der Methoden muss sicherstellen, dass der Zustand der Bean-Instanz für den jeweils folgenden Zustand geeignet ist. In unserem Beispiel wird in der Methode *ejbPassivate* (vgl. Listing 4.34) die Referenz auf die Data-Source gelöscht und in der Methode *ejbActivate* (vgl. Listing 4.35) wieder aus dem JNDI gelesen. Die Werte in den Variablen *foreignCurr*, *euroRate* und *currRate* bleiben erhalten, da sie nicht als *transient* deklariert sind und es sich um serialisierbare Datentypen handelt.

```
...
    public void ejbPassivate() {
        this.dataSource = null;
    }
...
```

Listing 4.34: EuroExchangeBean.ejbPassivate()

```
...
    public void ejbActivate() {
        try {
            Context c = new InitialContext();
            this.dataSource = (DataSource)c.lookup(dbRef);
        } catch(NamingException ex) {
            String msg = "Cannot get Resource-Reference:"
                        + ex.getMessage();
            throw new EJBException(msg);
        }
    }
...
```

Listing 4.35: EuroExchangeBean.ejbActivate()

Die Implementierung der Anwendungsmethoden (siehe Listing 4.36) ist in diesem Beispiel besonders einfach. Für das Umrechnen eines übergebenen Betrags ist nicht einmal ein Datenbankzugriff erforderlich.

```
...
    public float changeFromEuro(float amount) {
        return amount * this.currRate;
    }

    public float changeToEuro(float amount) {
        return amount * this.euroRate;
    }
...
```

Listing 4.36: EuroExchangeBean.changeToEuro und EuroExchangeBean.changeFromEuro

Die Methode *setForeignCurr* (Siehe Listing 4.37) ändert die der Bean-Instanz zugewiesene Währung. Die neue Währung wird als Parameter übergeben. Die Methode liest die zugehörigen Wechselkurse unter Zuhilfenahme der Methode *readCurrencyValues* aus der Datenbank und speichert sie in den Instanzvariablen.

```
...
    public void  setForeignCurr(String foreignCurr) {
        if(foreignCurr == null) {
            throw new EJBException("foreignCurr is null!");
        }
        this.foreignCurr = foreignCurr;
        this.euroRate    = 0f;
        this.currRate    = 0f;
        this.readCurrencyValues();
    }
...
```

Listing 4.37: EuroExchangeBean.setForeignCurr

Abschließend stellen wir in Listing 4.38 die Implementierung der Methode *readCurrencyValues* vor.

```
...
    private void readCurrencyValues()
        throws EJBException
    {
        if(this.foreignCurr == null) {
            throw new EJBException("foreignCurr not set");
        }
        final String query =
          "SELECT EURO, FOREIGNCURR FROM EURO_EXCHANGE "
        + "WHERE CURRENCY=?";
        Connection        con = null;
        PreparedStatement st  = null;
        ResultSet         rs  = null;
        try {
            con = this.dataSource.getConnection();
            st  = con.prepareStatement(query);
            st.setString(1, this.foreignCurr);
            rs = st.executeQuery();
            if(!rs.next()) {
                throw new EJBException("no such currency:"
                                       + this.foreignCurr);
            }
            this.euroRate = rs.getFloat(1);
            this.currRate = rs.getFloat(2);
        } catch(SQLException ex) {
            ex.printStackTrace();
            throw new EJBException("db-error:" + ex.getMessage());
        } finally {
```

```
            try { rs.close();  } catch(Exception ex) {}
            try { st.close();  } catch(Exception ex) {}
            try { con.close(); } catch(Exception ex) {}
        }
    }
...
```

Listing 4.38: EuroExchangeBean.readCurrencyValues

Deployment

Der Deployment-Deskriptor für das Beispiel (siehe Listing 4.39) unterscheidet sich von dem des vorhergehenden Beispiels nur durch den Namen der Bean, das Home- und Remote-Interface, die Bean-Klasse und ein weiteres wichtiges Detail: Da es sich um eine zustandsbehaftete Session-Bean handelt, muss der Wert *Stateful* in Abschnitt *session-type* gesetzt werden.

```
<?xml version="1.0" ?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0/
/EN" "http://java.sun.com/j2ee/dtds/ejb-jar_2_0.dtd">

<ejb-jar>
  <description>
    Dieser Deployment-Deskriptor enthält Informationen
    über die Session-Bean Exchange.
  </description>
  <enterprise-beans>
    <session>
      <ejb-name>Exchange</ejb-name>
      <home>ejb.exchangeSF.EuroExchangeHome</home>
      <remote>ejb.exchangeSF.EuroExchange</remote>
      <ejb-class>ejb.exchangeSF.EuroExchangeBean</ejb-class>
      <session-type>Stateful</session-type>
      <transaction-type>Bean</transaction-type>
      <resource-ref>
        <description> Euro-Datenbank </description>
        <res-ref-name>jdbc/EuroDB</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
        <res-sharing-scope>Shareable</res-sharing-scope>
      </resource-ref>
    </session>
  </enterprise-beans>

  <assembly-descriptor>
  </assembly-descriptor>

</ejb-jar>
```

Listing 4.39: Deployment-Deskriptor der Session-Bean EuroExchangeSF

Der Deployer muss weitere Informationen ergänzen. Wie beim vorhergehenden Beispiel wird der JNDI-Name definiert und der Ressource-Referenz eine Datenbank zugewiesen. Weitere Konfigurationswerte können je nach EJB-Container erforderlich sein. Bei zustandbehafteten Session-Beans wird normalerweise die maximale Anzahl der Bean-Instanzen im Arbeitsspeicher definiert. Werden mehr Bean-Instanzen zu einem Zeitpunkt benötigt, lagert der EJB-Container durch Passivierung einen Teil der Instanzen auf Platte aus. Wie im vorhergehenden Beispiel bereits erwähnt, sind solche zusätzlichen Deployment-Angaben in Art und Umfang stark unteschiedlich je nach eingesetztem Applikationsserver. Listing 4.40 zeigt ein fiktives Beispiel, wie diese Definitionen in der Konfigurationssprache eines EJB-Containers aussehen könnten.

```
<bean>
  <ejb-name>Exchange</ejb-name>
  <stateful-session-descriptor>
    <max-cache-size>200</max-cache-size>
  </stateful-session-descriptor>
  <reference-descriptor>
  <resource-reference-description>
    <res-ref-name>jdbc/EuroDB</res-ref-name>
    <jndi-name>jdbc/Oracle_Test_DB</jndi-name>
  </resource-reference-description>
  <jndi-name>
    ExchangeSF
  </jndi-name>
</bean>
```

Listing 4.40: Definitionen für das Deployment eines EJB-Containers

Client

Für einen einfachen Test der Funktionalität wird wieder ein Client entwickelt (vgl. Listing 4.41). Die getestete Funktionalität soll die gleiche wie im vorhergehenden Beispiel sein. Die Implementierung weicht jedoch ab, da sie ein anderes Interface bedienen muss.

```
package ejb.exchangeSF.client;

import ejb.exchangeSF.EuroExchange;
import ejb.exchangeSF.EuroExchangeHome;

import java.rmi.RemoteException;

import javax.ejb.CreateException;
import javax.ejb.RemoveException;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.rmi.PortableRemoteObject;
```

```
public class Client {

    private EuroExchange exchange;

    public Client() { }


    public void init(String curr)
        throws NamingException, RemoteException, CreateException
    {
        java.util.Properties p = new java.util.Properties();
        p.put(Context.INITIAL_CONTEXT_FACTORY, "JNDI-PROVIDER");
        p.put(Context.PROVIDER_URL, "PROVIDER-URL");
        Context ctx = new InitialContext(p);

        Object o = ctx.lookup("ExchangeSF");
        EuroExchangeHome home =
            (EuroExchangeHome)PortableRemoteObject.narrow(o,
             EuroExchangeHome.class);
        this.exchange = home.create(curr);
        System.out.println("init successful");
    }

    public void run() {
        System.out.println("Changing 100 US-Dollars to Euro");
        float amount = 0f;
        try {
            amount = this.exchange.changeToEuro(100);
        } catch(RemoteException ex) {
            ex.printStackTrace();
            return;
        }
        System.out.println("Result: " + amount);

        System.out.println("Changing " + amount +
                           " Euro to US-Dollars");
        float n_amount = 0f;
        try {
            n_amount = this.exchange.changeFromEuro(amount);
        } catch(RemoteException ex) {
            ex.printStackTrace();
            return;
        }
        System.out.println("Result: " + n_amount);
    }

    public void cleanUp() {
        try {
            this.exchange.remove();
        } catch(RemoteException ex) {
            ex.printStackTrace();
```

```
            } catch(RemoveException ex) {
                ex.printStackTrace();
            }
        }

        public static void main(String[] args) {
            Client c = new Client();
            try {
                c.init("US-Dollar");
            } catch(Exception ex) {
                ex.printStackTrace();
                return;
            }
            c.run();
            c.cleanUp();
        }

    }
```

Listing 4.41: Clientprogramm für Session-Bean EuroExchangeSF

Zusammenfassung

Die Verwendung einer zustandsbehafteten Session-Bean in diesem Beispiel hat Vor- und Nachteile. Der Conversational-State macht die Implementierung der Anwendungslogik in der Bean deutlich einfacher. Auch der Client kann vom Conversational-State profitieren und einen Teil seines Zustands auf den Server verlagern. Dadurch wird der Client weniger komplex und die Entwicklung einfacher. Dieser Effekt ist im Beispiel jedoch nicht zu beobachten. Bei der Auswahl des Beispiels stand eine vergleichbare Funktionalität mit dem Beispiel für zustandslose Session-Beans im Vordergrund.

Das Speichern der Wechselkurse in der Bean hat den Nachteil, dass eventuell mit veralteten Daten gearbeitet wird. Von Änderungen in der Datenbank erfährt die Bean-Instanz nicht automatisch. Wenn viele parallele Client-Zugriffe erfolgen, ist die zustandsbehaftete Bean eine weitaus größere Belastung für den Applikationsserver als eine zustandslose Session-Bean. Aufgrund der Tatsache, dass im Fall von zustandsbehafteten Session-Beans jeder Client eine exklusiv für ihn reservierte Bean-Instanz verwendet, hat der EJB-Container wenig Möglichkeiten den Ressourcenverbrauch einzuschränken. Zwar kann der EJB-Container Bean-Instanzen passivieren, um den Ressourcenverbrauch einzudämmen; die Passivierung auf ein externes Speichermedium und die Aktivierung sind jedoch relativ zeitaufwändige Operationen, was den Vorteil des eingeschränkten Ressourcenverbrauchs durch eine schlechte Performanz neutralisieren kann.

Zustandsbehaftete Session-Beans bieten sich an, wenn die Verlagerung von Anwendungszustand vom Client auf den Server sinnvoll ist und wenn sich die Anzahl der parallelen Client-Zugriffe in Grenzen hält. Viele parallele Clients sind nur dann nicht unbedingt ein Problem, wenn die Beanspruchungszeit einer stateful Session-Bean durch den Client von relativ kurzer Dauer ist und die zustandsbehaftete Session-Bean nach Beendigung der Benutzung mit *EJBObject.remove()* wieder freigegeben wird. Zustandslose Session-Beans bieten sich dann an, wenn kein Conversational-State benötigt wird bzw. wenn viele parallele Client-Zugriffe bedient werden müssen.

5 Entity-Beans

5.1 Einleitung

Eine Entity-Bean ist ein Objekt, das persistente Daten repräsentiert. In den meisten Fällen werden die Daten in einer Datenbank gespeichert. Eine Entity-Bean ist aber mehr als ein einfacher Datenspeicher. Bei einer Entity-Bean handelt es sich um ein Geschäftsobjekt mit eigener Funktionalität. Sie ermöglicht mehreren Clients den parallelen Zugriff auf transaktionsgesicherte Daten.

In der Weiterentwicklung der Enterprise JavaBeans Spezifikation wurde den Entity-Beans eine immer größere Bedeutung zuerkannt. Während die Version 1.0 Entity-Beans noch als optionalen Bestandteil definiert hat, wurden sie in der Version 1.1 zum Muss für jeden EJB-Container. Die Version 2.0 ändert die Definition der Entity-Beans in einigen Punkten und erweitert die Funktionalität maßgeblich. Für die Kompatibilität der Version 2.0 mit älteren Versionen sind beide Arten von Entity-Beans (1.1 und 2.0) Bestandteil der Spezifikation 2.0 und müssen von den EJB-Containern unterstützt werden.

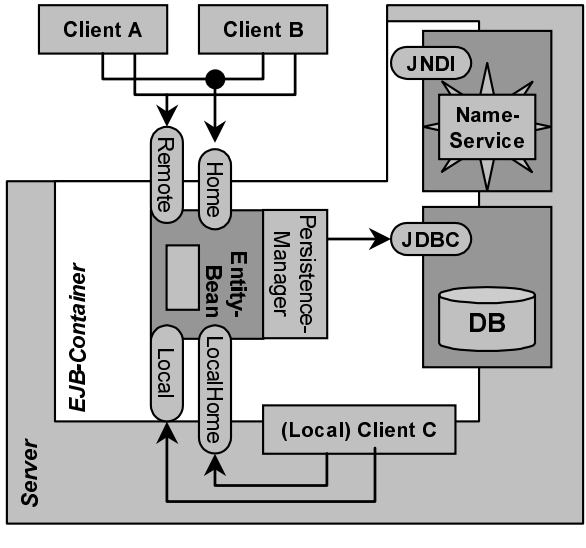

Abbildung 5.1: Überblick Entity-Beans

Eine Anwendung arbeitet mit Entity-Beans, wenn sie Informationen über mehrere Programmläufe hinweg speichert. Der Zustand des Objekts geht auch nicht verloren, wenn der Server gestoppt und wieder gestartet wird. Die Entity-Bean existiert so lange, bis ein Client sie explizit löscht. Im Gegensatz dazu sind die zuvor besprochenen Session-Beans nicht persistent. Deshalb geht der Zustand einer Session-Bean immer verloren, wenn der Client oder der Server heruntergefahren wird.

Weil es Entity-Beans mehreren Clients ermöglichen, parallel auf die gleichen Daten zuzugreifen, werden sie als zentrale Ressource betrachtet. Jeder Client einer Entity-Bean arbeitet fast so, als würde er exklusiv auf sie zugreifen. Die bekannten Probleme, die durch den parallelen Zugriff entstehen, vermeidet der EJB-Container durch die teilweise Serialisierung der Zugriffe und durch den Einsatz von Transaktionen (siehe auch Kapitel 7).

Beim parallelen Zugriff durch mehrere Clients unterscheiden sich Entity-Beans von Session-Beans. Bei zustandsbehafteten Session-Beans verwendet jeder Client eine eigene Bean-Instanz. Der Zustand einer Session-Bean steht somit immer nur exklusiv einem Client zu Verfügung. Greifen die Clients parallel auf den selben Datensatz zu, so benutzen alle die gleiche Entity-Bean. Der Zustand der Entity-Bean, die einen bestimmten Datensatz repräsentiert, steht allen Clients zur Verfügung.

Eine **Entity-Bean** definiert Methoden, Attribute, Beziehungen und einen Primärschlüssel. Wie bei den Session-Beans, realisieren die Methoden die Funktionalität der Bean und bestimmen damit das Verhalten. Die Attribute halten die Daten und bestimmen damit den Zustand einer Entity-Bean. Beziehungen definieren Abhängigkeiten zwischen Entity-Beans. Ein oder mehrere Attribute der Entity-Bean werden als Primärschlüssel zusammengefasst. Der Primärschlüssel ermöglicht es, alle Instanzen der Klasse eindeutig zu identifizieren. Das Home-Objekt (das wir in Kapitel 3 auch als *EJB-Home*-Objekt kennen gelernt haben) und der Wert des Primärschlüssels bestimmen zusammen die **Entity-Bean-Identität**. Diese identifiziert eine bestimmte Bean, die ein Objekt wie beispielsweise einen Kunden oder einen Lagerartikel modelliert.

Die Modellierung komplex strukturierter Daten wird von der Version 2.0 der Spezifikation durch Beziehungen zwischen Entity-Beans in Verbindung mit dem Local-Client-View erstmals zufriedenstellend durch den EJB-Container unterstützt. Bislang musste die Abbildung komplexer Datenstrukturen von Hand programmiert werden. Eine andere Alternative war die Verwendung sogenannter OR-Mapping-Tools von Drittherstellern. Durch den Einsatz von OR-Mapping-Tools war eine Enterprise-Bean aber nicht mehr ohne weiteres portierbar.

Der **Persistenzmechanismus** des EJB-Containers hat die Aufgabe, den Zustand einer Entity-Bean in eine Datenbank oder ein anderes Speichersystem zu schreiben. Wenn die Bean wieder benötigt wird, liest der Persistenzmechanismus die persistenten Daten aus und erstellt eine Entity-Bean-Instanz im Arbeitsspeicher. Dabei kann jede Entity-Bean immer mit ihrem Home-Objekt und dem Primärschlüssel identifiziert werden.

Bei der Verwendung eines Persistenzmanagers (Container-Managed-Persistence) wird der Aufwand für die Implementierung minimiert. Der Bean Provider kann aber auch den Persistenzmechanismus selbst in der Entity-Bean implementieren (Bean-Managed-Persistence), um Flexibilität zu gewinnen.

Ein EJB-Container bietet einen oder mehrere Persistenzmanager an. Jeder Persistenzmanager implementiert einen speziellen Persistenzmechanismus, der die persistenten Daten mit einer bestimmten Methodik abspeichert. Bei Container-Managed-Persistence ermöglicht die Abfragesprache *EJB-QL* den Zugriff auf die persistenten Daten, unabhängig von der verwendeten Methodik.

5.2 Konzepte

5.2.1 Entity-Bean-Typen

Dieses Buch behandelt drei Typen von Entity-Beans. Bei der Vorstellung der Konzepte von Entity-Beans werden alle drei miteinander verglichen. Die folgenden Typen von Entity-Beans sollten stets unterschieden werden:

- *Container-Managed-Persistence 2.0* – Entity-Beans mit Container-Managed-Persistence, wie sie in der Enterprise JavaBean-2.0-Spezifikation definiert wurden.
- *Container-Managed-Persistence 1.1* – Entity-Beans mit Container-Managed-Persistence, wie sie in der Enterprise JavaBean-1.1-Spezifikation definiert wurden. Aus Gründen der Abwärtskompatibilität sind diese auch Bestandteil der Version 2.0 der Enterprise JavaBeans Spezifikation.
- *Bean-Managed-Persistence* – Entity-Beans, die nicht den Persistenzmanager verwenden, sondern den Persistenzmechanismus selbst implementieren. Diese wurden in der Version 1.1 der Enterprise JavaBeans Spezifikation definiert und weitgehend unverändert in die Version 2.0 übernommen.

Die folgende Betrachtung der Konzepte behandelt diese drei Typen vergleichend. Bei der parallele Darstellung von Container-Managed-Persistence 1.1 und 2.0 werden die Vorteile der neuen Architektur verdeutlicht. Durch Beispiele werden die Entwicklungsaspekte der unterschiedlichen Konzepte veranschaulicht.

5.2.2 Attribute und abhängige Objekte

Der persistente Zustand einer Entity-Bean wird in ihren Attributen gespeichert. Bei der Entwicklung muss der Vorgang des Speichers besonders berücksichtigt werden. Ein wesentlicher Unterschied zwischen den drei Entity-Bean-Typen liegt darin, wie mit den persistenten Attributen verfahren wird.

Bean-Managed-Persistence

Übernimmt der Bean-Entwickler selbst die Verantwortung für die persistente Ablage der Attribute, so liegt es auch in seiner Verantwortung, wie mit den Attributen verfahren wird. Beispiele für die Programmierung werden in einem späterem Abschnitt vorgestellt.

Container-Managed-Persistence 1.1

Die Version 1.1 der Enterprise JavaBeans Spezifikation sieht vor, dass die persistenten Attribute als Attribute der Bean-Klasse definiert werden und im Deployment-Deskriptor angegeben werden. Die Attribute dürfen die folgenden Datentypen haben:

- primitive Datentypen der Programmiersprache Java (*int*, *float*, *long*, etc.),
- serialisierbare Datentypen der Programmiersprache Java.

Die Attribute werden *public* definiert, damit der EJB-Container darauf Zugriff hat. Es liegt in der Verantwortung des EJB-Containers, die Attribute zum richtigen Zeitpunkt mit der Datenbank abzugleichen.

Container-Managed-Persistence 2.0

Mit der Version 2.0 der Enterprise JavaBeans Spezifikation wurde die Handhabung der persistenten Attribute grundlegend geändert. Der Bean-Entwickler arbeitet mit einer abstrakten Bean-Klasse. Anstelle von Attributen in Form von Membervariablen der Bean-Klasse definiert er abstrakte Zugriffsmethoden zum Lesen und Schreiben der Attribute (sogenannte *getter*- bzw. *setter*-Methoden). Beim Deployment generiert der EJB-Container eine abgeleitete Klasse, die alle abstrakten Methoden implementiert. Die konkrete Implementierung der Zugriffsmethoden ist Bestandteil des Persistenzmechanismus.

Die Attribute dürfen die folgenden Datentypen haben:

- Primitiven Datentypen der Programmiersprache Java (*int*, *float*, *long*, etc.)
- Serialisierbare Datentypen der Programmiersprache Java

Serialisierbare Datentypen sind alle Klassen, die das Interface *java.io.Serializable* implementieren. Dazu gehören vor allem auch die Klassen *java.lang.String* und *java.util.Date*. Häufig werden statt der primitiven Datentypen von Java die einschlägigen Wrapper-Klassen primitiver Datentypen (wie z.B. *java.lang.Integer*, *java.lang.Float*, *java.lang.Long* etc.) verwendet. Die bislang genannten Datentypen können in der Regel problemlos auf die Datentypen von Datenbanken abgebildet werden. Bei individuellen Datentypen (eigenentwickelte Klassen, die das *Serializable*-Interface implementieren), die als persistentes Attribut einer Entity-Bean deklariert sind, ist das nicht ohne weiteres mög-

lich. Diese können von den Datenbanken, in denen sie persistent gespeichert werden, oft nur als Binärdaten (sog. BLOBs, Binary Large Objects) verarbeitet werden. Attribute dieser Art werden *abhängige Werte-Klassen* genannt (*dependent value classes*). Die Verwendung solch individueller Datentypen als persistente Attribute einer Entity-Bean ist eher die Ausnahme. Die Abbildung komplexer und recherchierbarer Datenstrukturen erfolgt über die Verwendung mehrerer Entity-Bean-Klassen, deren Abhängigkeiten durch Beziehungen definiert werden.

5.2.3 Persistente Beziehungen

Eine persistente Beziehung (Container-Managed-Relationship) ermöglicht es, von einer Instanz zu einer in Beziehung stehenden Instanz zu navigieren und ist nur für Entity-Beans mit Container-Managed-Persistence 2.0 relevant. Als Beispiel betrachten wir zwei Entity-Bean-Klassen einer Projektverwaltung, Projekt-Bean und Mitarbeiter-Bean. Um zu modellieren, welche Mitarbeiter in welchen Projekten arbeiten, ist eine Beziehung das richtige Mittel. Wenn das Projekt (Instanz einer Projekt-Bean) bekannt ist, können alle zugehörigen Mitarbeiter (Instanzen der Mitarbeiter-Bean) über die Beziehung gefunden werden. Das Verfolgen der Beziehung von einem Objekt zu einem in Beziehung stehenden Objekt wird Navigation genannt.

Für die Speicherung der Beziehungen werden die üblichen Mittel einer Datenbank verwendet (z.B. eine Fremdschlüsselbeziehung in einer relationalen Datenbank). Dadurch sind auch Beziehungen mit Datenbankmitteln recherchierbar.

Beziehungen kann es nur zwischen Entity-Beans geben, die im gleichen Deployment-Deskriptor definiert sind. Dort werden die Beziehung auch formal beschrieben. Außerdem sind Beziehungen zwischen Entity-Beans nur über den Local-Client-View möglich (Details zum Local-Client-View siehe Kapitel 4, *Local- vs. Remote-ClientView*). Der EJB-Container sorgt für das Erzeugen der Instanzen beim Navigieren über Beziehungen. Außerdem sorgt der EJB-Container dafür, dass die Beziehung zwischen den Instanzen von Entity-Beans persistent gemacht werden.

Aus der Sicht des Entwicklers einer Entity-Bean stellt sich eine Beziehung als ein Paar abstrakter Methoden dar, eine *get<Beziehungsname>*- und eine *set<Beziehungsname>*-Methode. Die abstrakte *get*-Methode dient zur Navigation über eine Beziehung. Die *set*-Methode dient dazu, eine Beziehung zu einer bestimmten Instanz herzustellen. Wie bei den abstrakten Methoden zur Speicherung von Attributen, werden beim Deployment konkrete Implementierungen der abstrakten Methoden für die Handhabung von Beziehungen durch die Tools des EJB-Containers generiert.

Für jede Beziehung wird eine *Kardinalität* definiert. Diese definiert das Zahlenverhältnis der in Beziehung stehenden Instanzen. Die folgenden Kardinalitäten sind möglich:

- *1:1* – Eine Instanz der ersten Klasse ist mit keiner oder einer Instanz der zweiten Klasse verbunden.
- *1:N* – Eine Instanz der ersten Klasse ist mit beliebig vielen Instanzen der zweiten Klasse verbunden.
- *N:M* – Eine Instanz der ersten Klasse ist mit beliebig vielen Instanzen der zweiten Klasse verbunden. Die Elemente der zweiten Klasse sind ihrerseits mit beliebig vielen Instanzen der ersten Klasse verbunden.

Beziehungen können in eine oder in beide Richtungen navigierbar sein. Ist die Beziehung in beiden Richtungen navigierbar, kennt jeder Beziehungspartner die verbundenen Instanzen. Man spricht hier von einer bidirektionalen Beziehung. Ist die Beziehung nur in einer Richtung navigierbar, hat nur ein Beziehungspartner Zugang zu den verbundenen Instanzen. Der andere Beziehungspartner hat keine Informationen über die Beziehung. In diesem Fall spricht man von einer unidirektionalen Verbindung.

Eine Beziehung bietet auch die Möglichkeit, eine Existenzabhängigkeit zu definieren. Wenn eine Instanz gelöscht wird, ist es möglich die über eine Beziehung verbundenen Instanzen mit zu löschen. Dieses Verfahren wird als *kaskadierendes Löschen* bezeichnet (*cascading delete*).

5.2.4 Primärschlüssel

Jede Entity-Bean-Klasse benötigt einen Primärschlüssel (Primary Key), der es ermöglicht, alle Entity-Bean-Identitäten eindeutig zu identifizieren. Ein Client verwendet den Primärschlüssel, um nach Entity-Beans zu suchen. Der EJB-Container nutzt den Primärschlüssel als eindeutiges Kennzeichen, um jede Entity-Bean-Identität in der Datenbank wiederzufinden.

Der Primärschlüssel kann aus einem oder mehreren persistenten Attributen der Entity-Bean bestehen. Es wird sichergestellt, dass jede Instanz einer Entity-Bean-Klasse eine andere Wertkombination in den Attributen des Primärschlüssels besitzt.

Die Kundennummer in einem Versandhaus ist ein Beispiel für einen Primärschlüssel mit einem einzelnen Attribut. Sie ist für jeden Kunden eindeutig. Im Gegensatz dazu ist das Kennzeichen eines Kraftfahrzeugs kein ausreichender Primärschlüssel für die Menge aller zugelassenen Fahrzeuge. Hinzu kommt eine laufende Nummer, die bei jeder Neuvergabe des gleichen Kennzeichens erhöht wird. Zusammen sind das Kennzeichen und die laufende Nummer eindeutig und ein Beispiel für einen Primärschlüssel, der aus mehreren Attributen besteht.

Die Attribute, die Bestandteile des Primärschlüssels sind, werden zusätzlich vom Bean-Entwickler in einer eigenen Klasse zusammengefasst. In dieser Klasse haben die

Attribute den gleichen Namen und den gleichen Datentyp. Die Primärschlüsselklasse wird dem EJB-Container im Deployment-Deskriptor bekannt gegeben. Auch der Client verwendet Primärschlüssel, um nach bestimmten Entity-Beans zu suchen.

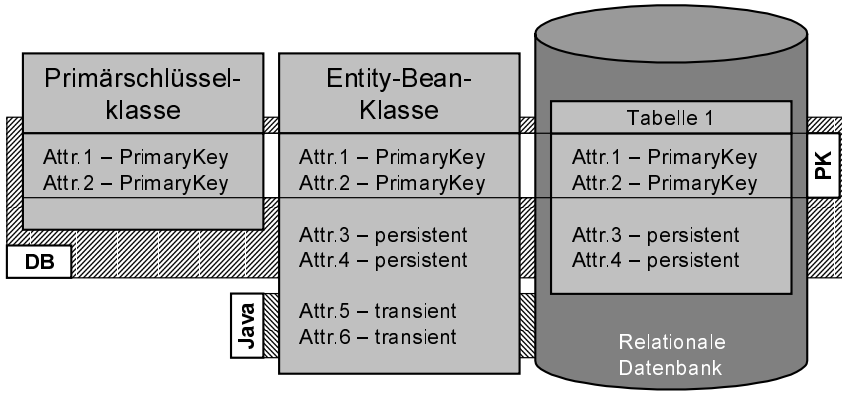

Abbildung 5.2: Überblick über die Attribute der Entity-Bean

Eine Ausnahme bilden Primärschlüssel, die aus einem einzigen Attribut bestehen. Zugunsten einer einfachen Programmierung kann hier auf eine gesonderte Primärschlüsselklasse verzichtet werden. Statt dessen kann eine Standard-Java-Klasse verwendet werden, die dem Typ des Primärschlüssels entspricht. Beispiele hierfür sind *java.lang.String*, *java.lang.Integer*, *java.lang.Long* etc. Das Primärschlüsselattribut wird im Deployment-Deskriptor bekannt gegeben.

5.2.5 Bean-Instanz und Bean-Identität

Für die Arbeit mit Entity-Beans ist es sehr wichtig, die Sicht des Entwicklers des Client-Programms von der Sicht des Entwicklers der Bean zu trennen. Dafür ist es besonders wichtig, dass die Begriffe Bean-Instanz und Bean-Identität verstanden werden.

Das Home-Objekt und der Wert des Primärschlüssels bilden zusammen die **Entity-Bean-Identität**. Eine solche Identität könnte z. B. den Kunden mit der Kundennummer 123 repräsentieren. Ein Objekt einer Entity-Bean-Klasse im Arbeitsspeicher wird als **Entity-Bean-Instanz** bezeichnet. Bei einer Entity-Bean-Identität in der Datenbank ohne eine Entity-Bean-Instanz im Arbeitsspeicher spricht man von einer passivierten Bean.

Vom Client aus gesehen ergibt sich ein einfaches Bild: Es existieren Entity-Bean-Objekte mit einer Entity-Bean-Identität. Das Bean-Objekt bietet dem Client das Remote- bzw. Local-Interface für den Zugriff an. Zur Verwaltung der Bean-Objekte steht dem Client das (Local-) Home-Interface der Bean-Klasse zur Verfügung. Wenn

der Client mit einer Entity-Bean-Identität arbeiten möchte, erhält er über das (Local-) Home-Interface eine Referenz auf das zugehörige Bean-Objekt. Allgemein ermöglicht das (Local-) Home-Interface dem Client das Erzeugen, Löschen und Finden von Bean-Objekten. Die Sicht des Clients beschränkt sich damit genau auf die Aspekte der Bean, die für ihn relevant sind.

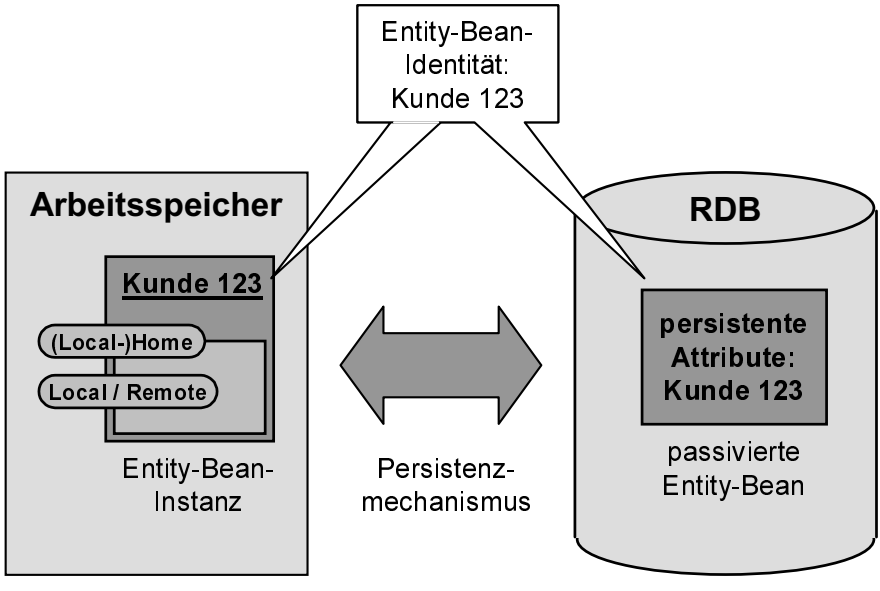

Abbildung 5.3: Die Entity-Bean-Identität am Beispiel eines Kunden

Der Entwickler der Bean benötigt eine andere Sicht. Er implementiert die Anwendungslogik in der Bean-Klasse. Zusätzlich wird von ihm ein (Local-) Home- und ein Remote- bzw. Local-Interface für diese Bean-Klasse geschrieben, die die Funktionalität der Bean veröffentlichen. Die Methoden dieser Interfaces werden teilweise vom Entwickler der Bean in der Bean-Klasse implementiert und teilweise vom EJB-Container bereitgestellt.

Wenn der Client mit einem Bean-Objekt arbeitet, wird die zugehörige Funktionalität von einer Bean-Instanz bereitgestellt. Es besteht jedoch ein wichtiger Unterschied zwischen der Sicht des Entwicklers des Client-Programms auf das Bean-Objekt und der Sicht des Entwicklers der Bean auf die Bean-Instanz. Das Bean-Objekt hat immer die gleiche Bean-Identität, während die Bean-Instanz während ihres Lebenszyklus die Bean-Identität wechseln kann. Diese Tatsache muss natürlich vom Entwickler der Entity-Bean berücksichtigt werden. Der folgende Abschnitt wird das Thema detailliert behandeln.

5.2.6 Persistenz

Eine typische Geschäftsanwendung verwaltet sehr große Datenmengen. Bei dem Einsatz von Enterprise JavaBeans werden diese Daten normalerweise in Entity-Beans gespeichert. Die große Menge entstehender Entity-Beans kann nicht durchgehend im Hauptspeicher verweilen, sondern muss auf ein Medium mit größerer Speicherkapazität ausgelagert werden. Im Regelfall wird dafür eine Datenbank eingesetzt, die die Daten auf der Festplatte speichert und Mechanismen für die Datensicherheit anbietet (z. B. Sicherung der Daten von der Festplatte auf Magnetband).

Damit eine Entity-Bean-Instanz persistent wird, müssen alle persistenten Attribute gespeichert werden. Wird die Instanz wieder benötigt, so kann mit den gespeicherten Attributen die Instanz in ihrem ursprünglichen Zustand wiederhergestellt werden. Unter dem Begriff *Persistenzmechanismus* wird das Verfahren zum Speichern und Wiederherstellen des Zustands verstanden.

Bezüglich der Persistenz von Entity-Beans werden zwei Vorgehensweisen unterschieden:

- *Bean-Managed-Persistence*: Eine Entity-Bean kann selbst die persistente Ablage ihrer Attribute realisieren.
- *Container-Managed-Persistence*: Die Verantwortung für die persistente Ablage der Attribute einer Entity-Bean wird dem Persistenz-Manager des EJB-Container übertragen.

Im ersten Fall ist es die Aufgabe des Bean-Entwicklers, die Methoden zu implementieren, die dem EJB-Container die Verwaltung der Entity-Beans ermöglichen. Innerhalb dieser Methoden kann der Entwickler die Strategie, mit der die Attribute geschrieben, geladen und gesucht werden, frei wählen. Die Möglichkeit, den Persistenzmechanismus selbst in der Bean zu realisieren, gibt dem Bean-Entwickler große Flexibilität. Er muss sich nicht auf die Persistenzmechanismen des EJB-Containers beschränken und kann auch anwendungsspezifische Methoden benutzen.

Im zweiten Fall wird der Entwickler der Entity-Bean vollständig von der Programmierung der Zugriffsmethoden entlastet. Der Persistenz-Manager sorgt dafür, dass die persistenten Attribute der Entity-Bean gespeichert werden.

Die EJB-Spezifikation definiert, dass der EJB-Container die persistenten Daten mit einem Verfahren speichert, das mit der Serialisierung von Java vergleichbar ist. Im Rahmen dieser Definition sind jedoch unterschiedlichste Verfahren denkbar. Um große Datenmengen zu verwalten, wird im Regelfall eine Datenbank eingesetzt. Neben den verbreiteten relationalen Datenbanken (RDB) ist auch der Einsatz von objektorientierten Datenbanken (OODB) und anderen Persistenzsystemen denkbar.

In einer bestehenden Infrastruktur kann es erforderlich sein, die Speichermechanismen von existierenden Anwendungen zu nutzen. Dies ist besonders interessant, um den existierenden Datenbestand in Mainframe-Anwendungen zu nutzen. Der Vorteil bei diesem Vorgehen ist, dass alle Daten der existierenden Anwendung und auch der neuen EJB-Anwendungen zur Verfügung stehen. Dies ermöglicht eine sanfte Migration, die im Gegensatz zur harten Migration nicht den vollständigen Austausch einer Technologie zu einem bestimmten Zeitpunkt anstrebt.

Der EJB-Container hat einen oder mehrere Persistenz-Manager, deren Verantwortung es ist, die benötigten Zugriffsmethoden zu generieren. Ein Persistenzmanager bestimmt die Art des eingesetzten Persistenzsystems und die Abbildung der persistenten Daten auf dieses System. In der Regel werden die EJB-Container genau einen Persistenz-Manager zur Verfügung stellen, der die Speicherung in relationalen Datenbanken unterstützt. Gibt es mehrere Persistenz-Manager, wird beim Deployment der Persistenzmechanismus durch die Wahl des Persistenz-Managers festgelegt.

Für die Generierung der Zugriffsmethoden benötigt der Persistenz-Manager genaue Informationen über die Entity-Bean. Beim Deployment wird die Implementierung der Entity-Bean-Klasse mit ihren abhängigen Klassen und der Deployment-Deskriptor untersucht. Abhängig vom gewählten Persistenzmechanismus werden zusätzliche Daten benötigt, wie z.B. Tabellen und Spaltennamen für die Attribute einer Entity-Bean in einer Datenbank. Der Vorgang des Deployments muss für die Installation der Bean in einem anderen EJB-Container wiederholt werden.

Der Aufwand für die Implementierung kann durch die Verwendung von Container-Managed-Persistence minimiert werden. In diesem Fall verlässt sich die Bean ganz auf die Infrastruktur des EJB-Containers und enthält keine eigenen Datenbankzugriffe. Deshalb bleibt die Bean auch portabel und kann ohne zusätzlichen Aufwand mit anderen Datenbanken oder anderen EJB-Containern verwendet werden.

5.2.7 Speicherung in relationalen Datenbanken

Die EJB-Architektur verwendet mit den Entity-Beans eine objektorientierte Struktur zum Speichern aller Daten. Deshalb würde eine Speicherung in objektorientierten Datenbanken nahe liegen. Aber relationale Datenbanken sind im Markt besser etabliert und haben sowohl bei neuen als auch bei existierenden Anwendungen die größere Verbreitung. Aus diesen Gründen werden auch Enterprise JavaBeans sehr häufig mit relationalen Datenbanken eingesetzt. Der Persistenzmechanismus muss hier jedoch die objektorientierte Datenstruktur auf eine relationale Struktur abbilden (*Object Relational Mapping*).

Die Abbildung von objektorientierten Datenstrukturen auf die Tabellen einer relationalen Datenbank ist ein komplexes Thema. Jedoch arbeiten Entity-Beans nicht mit Vererbung, da Komponentenvererbung von der EJB-Spezifikation nicht behandelt wird. Dadurch ist das benötigte Verfahren vergleichsweise einfach.

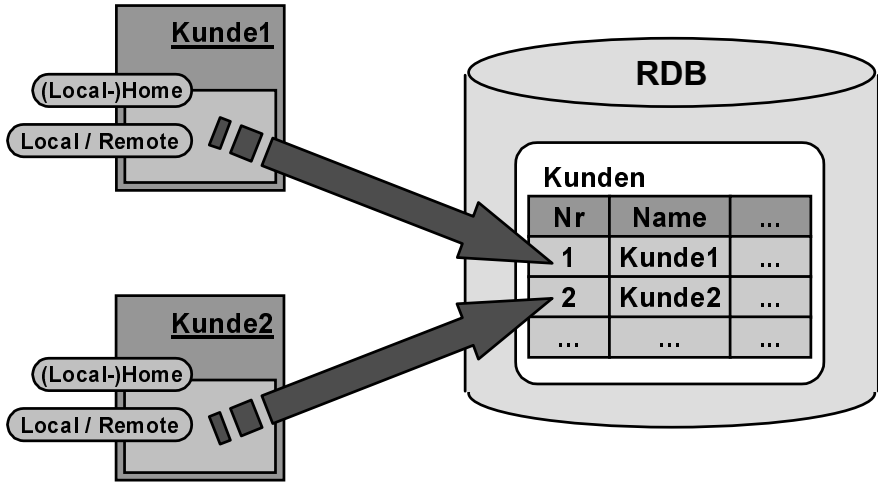

Abbildung 5.4: Object Relational Mapping von Entity-Beans

Eine Bean-Klasse, die nur einfache Attribute besitzt, kann auf eine einzelne Datenbanktabelle (auch *Relation* genannt) abgebildet werden. Jedes einfache persistente Attribut der Klasse erhält dabei eine eigene Spalte (auch *Attribut* genannt). Die Instanzen der Bean werden in je eine Zeile (auch *Tupel* genannt) geschrieben. Damit sind auch die Attribute des Primärschlüssels in bestimmten Spalten der Datenbanktabelle enthalten. Der Primärschlüssel der Entity-Bean wird auch als Primärschlüssel der zugehörigen Datenbanktabelle verwendet. Damit ist neben der notwendigen Auffindbarkeit ein effizienter Zugriff auf bestimmte Entity-Bean-Identitäten sichergestellt.

Für die Modellierung komplexer Strukturen reichen die einfachen Datentypen, die in einer Spalte der Datenbank abgebildet werden können, nicht aus. Hier können abhängige Werte-Klassen oder mehrere Entity-Bean-Klassen in Verbindungen mit container-managed-Relationships verwendet werden. Bei der Abbildung der Objekte auf eine relationale Datenbank wird der Unterschied zwischen den beiden Ansätzen deutlich:

- *Objekte Abhängiger Werte-Klassen (dependent value classes)* werden serialisiert und in einer Datenbankspalte für Binärdaten (BLOB) gespeichert. Die Datenbank kann das gespeicherte Binärformat nicht interpretieren. Deshalb kann auf diesen Spalten nicht gesucht werden. Auch können diese Spalten nicht für Fremdschlüsselbeziehungen zu anderen Tabellen verwendet werden.
- *Bei mehreren Entity-Beans, die über Container-Managed-Relationships in Beziehung zueinander gesetzt werden,* wird jede Entity-Bean in einer eigenen Tabelle gespeichert. Die Tabellen werden über die Attribute der Entity-Beans zueinander in Beziehung gesetzt. Die Attribute der Entity-Beans sind recherchierbar und es besteht die Möglichkeit über die Beziehungen zu navigieren.

Wird für die Abbildung komplexer Datenstrukturen die Modellierung über mehrere Entity-Beans und Container-Managed-Relationships gewählt (was nur für Container-Managed-Persistence der Version 2.0 möglich ist), muss für die Abbildung der Beziehungen auf die Datenbank abhängig von der Kardinalität der Entity-Bean-Beziehung zwischen zwei unterschiedliche Ansätzen gewählt werden:

- *Einfache Fremdschlüsselbeziehung*: Diese Methode eignet sich nur für die Kardinalitäten 1:1 oder 1:N. Die Tabelle mit der Kardinalität N (bei 1:1 eine beliebige) erhält zusätzlich die Attribute des Primärschlüssels der anderen Tabelle.
- *Relationstabelle*: Für die Kardinalität N:M wird eine zusätzliche Tabelle eingeführt. Die Attribute der Relationstabelle sind die Summe der Primärschlüsselattribute beider Tabellen, die durch die Beziehung verknüpft werden. Jede Zeile der Relationstabelle verbindet eine Zeile der einen Tabelle mit einer Zeile der anderen Tabelle.

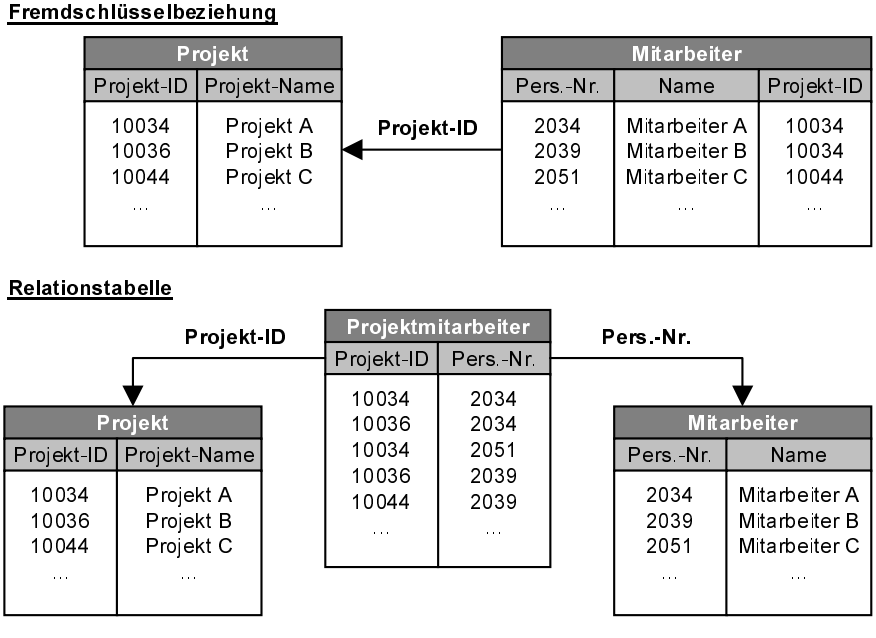

Abbildung 5.5: Fremdschlüssel versus Relationstabelle

Die möglichen Navigationsrichtungen haben keine Auswirkung auf die Modellierung. Abbildung 5.5 verdeutlicht die beiden Ansätze anhand des bereits erwähnten Beispiels *Projekt*. Falls ein Mitarbeiter in nur einem Projekt tätig sein kann, handelt es sich um eine 1:N Beziehung zwischen der Entität *Projekt* und der Entität *Mitarbeiter*. Die Beziehung zwischen den beiden Entitäten wird über einen Fremdschlüssel (Spalte *Projekt-*

ID) in der Entität *Mitarbeiter* gespeichert, der auf das entsprechende Projekt verweist. Falls ein Mitarbeiter in mehreren Projekten arbeiten kann, handelt es sich um eine N:M Beziehung zwischen der Entität *Projekt* und der Entität *Mitarbeiter*. Die Beziehung zwischen den beiden Entitäten wird in diesem Fall über die Tabelle *Projektmitarbeiter* (eine so genannte Relationstabelle) gespeichert. Diese Tabelle wird nicht als Entity-Bean modelliert. Sie wird vom EJB-Container lediglich für die Speicherung der Beziehungen zwischen den Entitäten *Projekt* und *Mitarbeiter* verwendet.

Den Persistenzmechanismus implementiert der Persistenz-Manager. Um seine Arbeit erledigen zu können, benötigt dieser genaue Angaben darüber, welches Attribut der Entity-Bean in welcher Datenbanktabelle und in welcher Spalte gespeichert werden soll. Gleiches gilt für Beziehungen. Der Persistenz-Manager muss wissen, über welche Attribute die Entity-Beans zueinander in Beziehung stehen, welche Kardinalität die Beziehung hat und wie die Beziehung gespeichert werden soll (über eine Fremdschlüsselbeziehung oder eine Relationstabelle). Diese Angaben macht der Deployer. Außerdem muss das Datenbankschema in der Datenbank definiert werden. Hier ist ein Datenbankexperte gefragt, der das beste Schema mit den richtigen Indizes unter Berücksichtigung der Auswirkungen auf Performanz und Speicherverbrauch definiert.

5.2.8 Lebenszyklus einer Bean-Instanz

Das Verständnis für den Lebenszyklus einer Entity-Bean-Instanz ist unbedingte Voraussetzung für die Entwicklung. Abbildung 5.6 zeigt die verschiedenen Zustände und Zustandsübergänge einer Entity-Bean. Bei den Zustandsübergängen wird unterschieden, ob der EJB-Container oder der Client den Zustandsübergang auslöst.

Die EJB-Spezifikation unterscheidet drei Zustände der Entity-Bean-Instanz. Zur Verdeutlichung der Aspekte bei der Programmierung teilen wir den Zustand *Ready* in drei Unterzustände auf:

- *Nicht existent*: Die Instanz existiert nicht.
- *Pooled*: Die Instanz existiert, ihr wurde aber noch keine Bean-Identität zugewiesen. Die Instanz wird verwendet, um die Zugriffe des Clients auf das Home-Interface auszuführen, da sich diese nicht auf eine spezielle Bean-Identität beziehen.
- *Ready*: Die Instanz hat eine Bean-Identität und kann vom Client verwendet werden. Wir teilen den Zustand *Ready* in drei Unterzustände auf:

  1. *Ready-Async*: Die Attributwerte sind evtl. nicht mit dem aktuellen Datenbankinhalt abgeglichen. Entweder wurden die Attribute noch nicht initialisiert, oder der Datenbankinhalt wurde durch einen parallelen Zugriff geändert.
  2. *Ready-Sync*: Die Attributwerte haben den aktuellen Inhalt.

3. *Ready-Update*: Die Attribute der Bean werden vom Client geändert. Normalerweise befindet sich die Bean in einer Transaktion. Die neuen Werte wurden noch nicht oder nur teilweise in die Datenbank geschrieben.

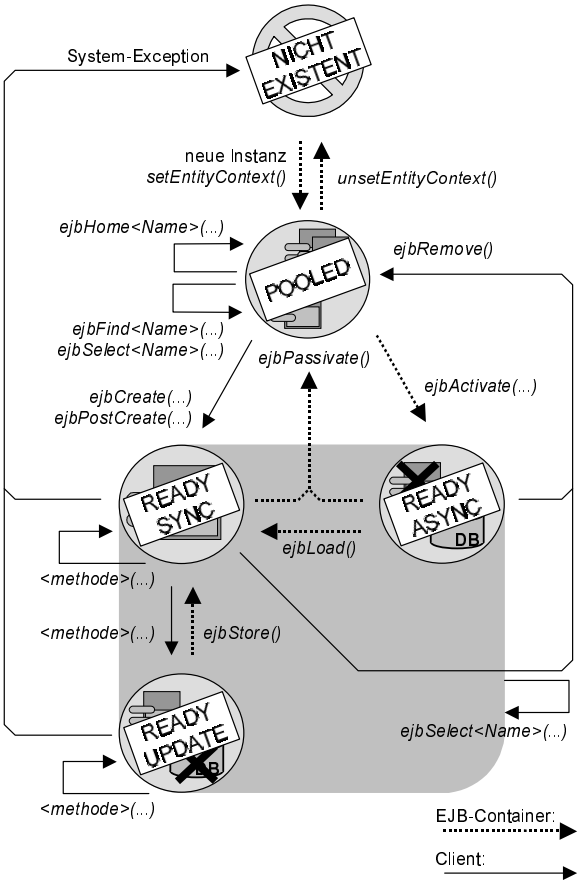

Abbildung 5.6: Zustandsgraph für eine Entity-Bean-Instanz

Für jede Entity-Bean-Klasse hat der EJB-Container einen Pool, in dem er die Instanzen verwaltet. Der Pool ist eine Performanzoptimierung, die vor allem durch eine Minimierung des Aufwands zur automatischen Speicherverwaltung (*Garbage-Collection*) erreicht wird. Der EJB-Container vermeidet das dauernde Erzeugen und Verwerfen von Java-Objekten. Die Instanzen aus dem Pool verwendet der EJB-Container nacheinander für unterschiedliche Bean-Identitäten. Da eine Instanz nur eine Bean-Identität zu einem Zeitpunkt haben kann, muss der Pool mindestens so viele Instanzen enthalten, wie unterschiedliche Bean-Identitäten parallel benutzt werden.

Benötigt der EJB-Container eine neue Instanz im Pool, so erzeugt er eine Instanz der Bean-Klasse und initialisiert diese mit *setEntityContext*. Die Instanz hat noch keine Bean-Identität. Will der EJB-Container zu einem anderen Zeitpunkt die Menge der Bean-Instanzen im Pool verringern, so ruft er die Methode *unsetEntityContext* auf und verwirft die Instanz danach.

Jetzt sucht der Client beispielsweise ein Bean-Objekt mit der *findByPrimaryKey*-Methode des Home-Interface. Der EJB-Container leitet den Aufruf an eine beliebige Bean-Instanz weiter. Dies ist möglich, weil die Bean-Instanz für die Ausführung der Methode keine Bean-Identität benötigt.

Die entsprechende Methode mit den gleichen Parametern wurde in der Entity-Bean-Klasse entweder vom Bean-Entwickler implementiert (Bean-Managed-Persistence) oder wird vom EJB-Container bereitgestellt (Container-Managed-Persistence).

Als Ergebnis erhält der Client das Remote- bzw. Local-Interface des gefunden Bean-Objekts (hat die Suche mehrere Ergebnisse, so wird eine *Collection* oder *Enumeration* für die Rückgabe mehrerer Remote- bzw. Local-Objekte verwendet). Der EJB-Container muss die entsprechende Bean-Instanz für den Aufruf von Methoden vorbereiten. Er aktiviert dazu eine beliebige Bean-Instanz und gibt ihr die Bean-Identität in Form des Primärschlüssels bekannt. Dann ruft er die Methode *ejbActivate* auf. Die Bean-Instanz ist jetzt im Zustand *Ready-Async*.

Wenn der Client eine Methode der Bean aufruft, muss der EJB-Container dafür sorgen, dass die Bean-Instanz die aktuellen Attributwerte hat. Der Persistenzmechanismus des EJB-Containers synchronisiert dazu die Bean-Instanz mit dem Persistenzmedium (Container-Managed-Persistence). Danach wird die Methode *ejbLoad* aufgerufen, um die Bean über die eben erfolgte Synchronisation zu informieren. Wird der Persistenzmechanismus selbst realisiert (Bean-Managed-Persistence), so wird die Synchronisation in der Methode *ejbLoad* vom Bean-Provider implementiert. Nach dem Aufruf von *ejbLoad* ist die Bean-Instanz im Zustand *Ready-Sync*. Alle lesenden Operationen des Clients werden in diesem Zustand ausgeführt.

Greift der Client schreibend auf die Entity-Bean zu und ändert ihre Attributwerte, so ist die Bean-Instanz vorübergehend nicht synchron mit der Datenbank. Die Bean-Instanz befindet sich im Zustand *Ready-Update*. Der EJB-Container veranlasst die Synchronisation zu gegebener Zeit. Der Persistenzmechanismus des EJB-Containers übernimmt das Schreiben der Attribute in die Datenbank (Container-Managed-Persistence). Danach wird die Methode *ejbStore* aufgerufen, um die Bean-Instanz über die eben erfolgte Synchronisation zu benachrichtigen. Wird der Persistenzmechanismus selbst realisiert (Bean-Managed-Persistence), wird die Synchronisation in der Methode *ejbStore* vom Bean-Provider selbst implementiert. Nach dem Aufruf von *ejbStore* befindet sich die Bean-Instanz wieder im Zustand *Ready-Sync*. Alle schreibenden Zugriffe

werden normalerweise durch eine Transaktion geschützt, so dass die Abweichung zwischen den Attributwerten und dem Datenbankinhalt keine Gefahr für die Konsistenz des Datenbestands darstellt (Details zu Transaktionen siehe Kapitel 7).

Alle Bean-Instanzen, die nicht gerade von einem Client benutzt werden und die sich in keiner Transaktion befinden, können vom EJB-Container für andere Bean-Identitäten wiederverwendet werden. Durch den Aufruf der Methode *ejbPassivate* gibt der EJB-Container der Bean-Instanz bekannt, dass sie in den Zustand *Pooled* wechselt. Wird die alte Bean-Identität vom Client wieder benötigt, so kann der EJB-Container eine beliebige Bean-Instanz dafür verwenden und aktivieren.

Für den Entwickler von Entity-Beans bedeutet das, dass die Bean-Instanz in der Methode *ejbPassivate* eventuell initialisiert werden muss. Da die Bean-Instanz unter Umständen längere Zeit im Pool verweilen kann, sollten auf jeden Fall zum Zeitpunkt des Passivierens alle von ihr belegten Ressourcen freigegeben werden. Außerdem ist zu beachten, dass die Bean für eine andere Bean-Identität wiederverwendet werden kann.

Das Erzeugen einer neuen Bean-Identität funktioniert ähnlich. Der Client ruft die Methode *create* des Home-Interface auf. Der EJB-Container leitet diesen Aufruf an die entsprechende Methode *ejbCreate* einer Bean-Instanz im Zustand *Pooled* weiter. Die Methode wertet die Parameter aus und setzt die Attributwerte der Bean-Instanz entsprechend. Danach wird die Methode *ejbPostCreate* vom EJB-Container mit den gleichen Parametern aufgerufen. Im Unterschied zu *ejbCreate* steht der Methode *ejbPostCreate* die Bean-Identität im Bean-Kontext zur Verfügung. Damit kann diese Methode weitere Initialisierungsschritte ausführen. Im Fall von Container-Managed-Persistence sorgt der EJB-Container für den entsprechenden Eintrag in der Datenbank. Im Fall von Bean-Managed-Persistence muss der Bean-Provider das Anlegen des neuen Datensatzes in der Methode *ejbCreate* der Bean-Klasse selbst programmieren. Nach der Ausführung von *ejbCreate* und *ejbPostCreate* befindet sich die Bean-Instanz im Zustand *Ready-Sync*.

Der Client hat zwei Möglichkeiten eine Bean-Identität zu löschen. Mit der Methode *remove* des (Local-) Home-Interfaces unter Angabe des Primärschlüssels oder durch den Aufruf der Methode *remove* am Local- bzw- Remote-Interface der Entity-Bean. Im ersten Fall übernimmt der EJB-Container die nötigen Zwischenschritte. Er aktiviert gegebenenfalls eine Bean-Instanz für die Bean-Identität und ruft *ejbRemove* auf. Die zugehörigen Daten werden im Fall der Container-Managed-Persistence vom EJB-Container (über den Persistenz-Manager) aus der Datenbank gelöscht. Im Fall der Bean-Managed-Persistence muss der Bean-Provider das Löschen des Datensatzes in der Methode *ejbRemove* selbst programmieren. Mit dem Löschen des Datensatzes in der Datenbank wird auch die zugehörige Entity-Bean-Identität gelöscht. Die Bean-Instanz geht danach in den Zustand *Pooled* über.

Im Lebenszyklus einer Entity-Bean-Instanz kann es auch zu Systemfehlern kommen. Beispielsweise kann ein Netzwerkfehler auftreten oder die Datenbank nicht verfügbar sein. Tritt in einer Entity-Bean-Instanz ein Systemfehler auf, so löscht der EJB-Container diese Instanz. Die Instanz wird dem Pool nicht wieder zugeführt. Für jeden folgenden Zugriff auf die zugehörige Bean-Identität muss der EJB-Container eine neue Bean-Instanz aus dem Pool aktivieren. Für den Bean-Entwickler bedeutet das, dass er im Fall eines Systemfehlers die Bean-Instanz nicht für den Pool initialisieren muss.

Nach dieser Einführung, die für alle Typen von Entity-Beans relevant war, werden wir nun in den folgenden Abschnitten detailliert auf die Besonderheiten der einzelnen Entity-Bean-Typen eingehen.

5.2.9 Entity-Kontext

Der Entity-Kontext (Interface *javax.ejb.EntityContext*) ist wie der Session-Kontext bei Session-Beans die Schnittstelle zwischen dem EJB-Container und der Entity-Bean-Instanz. Der Entity-Kontext wird der Bean-Instanz über die Callback-Methode *setEntityContext* zugewiesen, nachdem sie erzeugt wurde. Die Entity-Bean verwendet diesen Kontext während ihrer gesamten Lebensdauer. Der Kontext einer Entity-Bean wird während ihres Lebenszyklus vom EJB-Container häufiger verändert. Immer wenn die Entity-Bean vom Zustand *Pooled* in den Zustand *Ready* wechselt, wechselt auch der Kontext der Entity-Bean. Durch den Kontext erfährt die Entity-Bean vom EJB-Container, welche Identität sie momentan repräsentiert, die Identität des Clients der gerade eine Business-Methode aufruft und Informationen über die gerade laufende Transaktion.

Im weiteren Verlauf dieses Abschnitts werden die wichtigsten Methoden des Entity-Kontexts kurz vorgestellt.

`EJBHome getEJBHome()`

Diese Methode ermöglicht der Bean-Instanz den Zugriff auf ihr eigenes Home-Interface. Der Datentyp des Rückgabewerts muss in das spezielle Home-Interface der Bean-Klasse umgewandelt werden (Type-Casting). Die Methode löst eine Ausnahme vom Typ *java.lang.IllegalStateException* aus, wenn die Entity-Bean kein Home-Interface (sondern ein Local-Home-Interface) hat.

`EJBLocalHome getEJBLocalHome()`

Diese Methode ermöglicht der Bean-Instanz den Zugriff auf ihr eigenes Local-Home-Interface. Der Datentyp des Rückgabewerts muss in das spezielle Local-Home-Interface der Bean-Klasse umgewandelt werden. Die Methode löst eine Ausnahme vom Typ *java.lang.IllegalStateException* aus, wenn die Entity-Bean kein Local-Home-Interface (sondern ein Home-Interface) hat.

`Object getPrimaryKey()`

Diese Methode ermöglicht der Bean-Instanz den Zugriff auf den Primärschlüssel. Der Primärschlüssel spiegelt die Identität wieder, die die Entity-Bean gerade repräsentiert. Diese Methode kann nicht in der Methode *ejbCreate* aufgerufen werden. Wenn die Methode *ejbCreate* aufgerufen wird, ist der Übergang in den Zustand *Ready* noch nicht abgeschlossen. Der Primärschlüssel steht der Enterprise-Bean erst in der Methode *ejbPostCreate* zur Verfügung sowie in allen anderen Methoden (mit Ausnahme der Methode *unsetEntityContext*).

`void setRollbackOnly()`

Diese Methode dient zur Transaktionssteuerung (siehe auch Kapitel 7). Eine Entity-Bean-Instanz verwendet diese Methode im Fehlerfall, um sicherzustellen, dass die aktive Transaktion mit einem Rollback beendet wird. So ein Fehlerfall kann z.B. das Scheitern des Speicherns von persistenten Attributen bei Entity-Beans mit Bean-Managed-Persistence sein.

`boolean getRollbackOnly()`

Die Methode *getRollbackOnly* ist das lesende Gegenstück zu der Methode *setRollbackOnly*. Mit *getRollbackOnly* kann überprüft werden, ob es noch möglich ist, die aktive Transaktion mit *commit* zu beenden.

`UserTransaction getUserTransaction()`

Normalerweise übernimmt die Steuerung der Transaktionen der EJB-Container. Die Methode *getUserTransaction* benötigt eine Entity-Bean-Instanz nur, wenn sie die Transaktionssteuerung selbst übernehmen möchte. Der Rückgabewert der Methode ist ein Objekt der Klasse *javax.transaction.UserTransaction*, das Zugriff auf den Transaktionskontext ermöglicht (siehe Kapitel 7).

`Principal getCallerPrincipal()`

Die Methode erlaubt es der Entity-Bean-Instanz, die Identität des Clients zu bestimmen. Der Rückgabewert ist ein Objekt der Klasse *java.security.Principal* (siehe auch Kapitel 8).

`boolean isCallerInRole(java.lang.String roleName)`

Mit dieser Methode wird überprüft, ob der angemeldete Benutzer eine bestimmte Rolle im Sicherheitskonzept hat. Die möglichen Rollen werden im Deployment-Deskriptor definiert (siehe auch Kapitel 8).

5.3 Container-Managed-Persistence 2.0

5.3.1 Überblick

Abbildung 5.7 gibt einen vollständigen Überblick über die Klassen und Interfaces einer Entity-Bean mit Container-Managed-Persistence nach der Version 2.0, die den Remote-Client-View unterstützt. Rechts steht die Entity-Bean-Klasse, deren Objekte zur Laufzeit vom EJB-Container verwaltet werden. Auf der linken Seite befinden sich die Schnittstellen, über die der Client auf die Bean zugreift und die durch das Home- bzw. Remote-Objekt des EJB-Containers implementiert werden.

In der Abbildung werden alle symbolischen Namen, die bei der konkreten Programmierung durch die richtigen Begriffe ersetzt werden müssen, in spitzen Klammern (<abc>) dargestellt. Ein Plus (+) bezeichnet eine öffentliche (*public*) Methode. Abstrakte Methoden werden kursiv dargestellt.

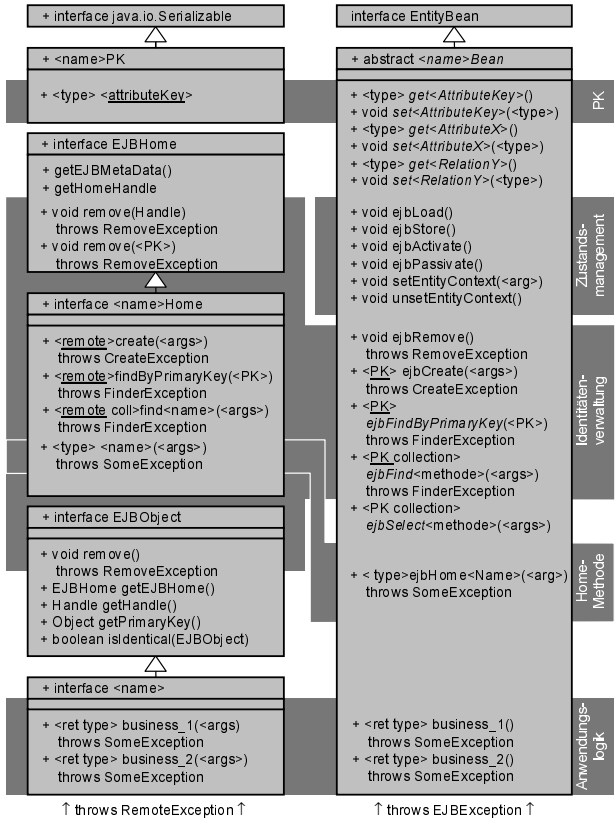

Abbildung 5.7: Remote-Interfaces und Klassen der CMP Entity-Bean (EJB 2.0)

Die Entity-Bean-Klasse implementiert das Interface *javax.ejb.EntityBean*. In dieser Entity-Bean-Klasse implementiert der Bean-Provider die eigentliche Funktionalität der Bean. Für jedes persistente Attribut werden zwei abstrakte Zugriffsmethoden (*get*- und *set*-) definiert.

Die *get*-Methode dient zum Lesen des Attributs. Sie besitzt keine Parameter und ihr Rückgabewert hat den Datentyp des persistenten Attributs. Die *set*-Methode dient zum Schreiben des Attributs. Sie hat einen Parameter mit dem Datentyp des persistenten Attributs und keinen Rückgabewert. Die Implementierung dieser Methoden übernimmt der EJB-Container beim Deployment.

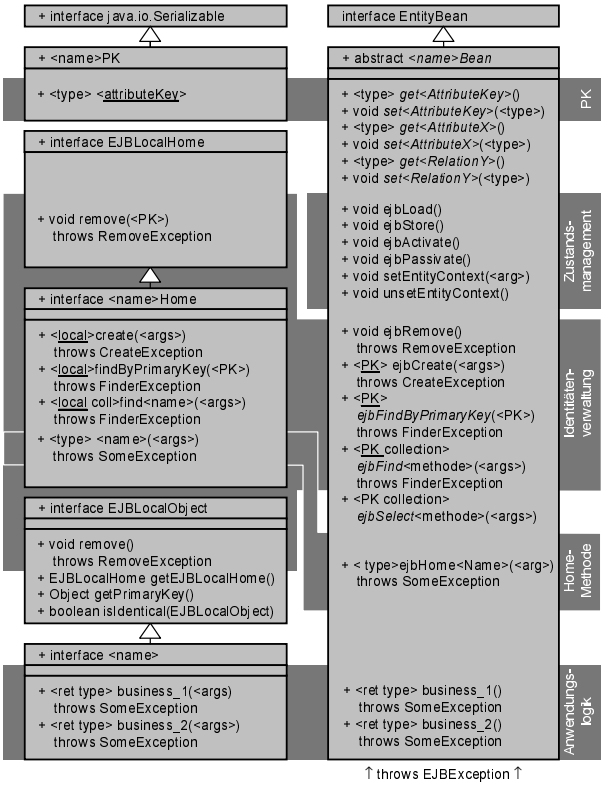

Abbildung 5.8: Local-Interfaces und Klassen der CMP Entity-Bean (EJB 2.0)

Oben links in Abbildung 5.7 steht die Klasse für den Primärschlüssel der Bean. Da Objekte dieser Klasse zwischen Server und Client über das Netzwerk transportiert werden, muss die Klasse serialisierbar sein. Sie implementiert das Interface *java.io. Serializable*. In der Primärschlüsselklasse werden die Attribute definiert, die den Primärschlüssel der zugehörigen Entity-Bean-Klasse bilden. Für jedes (virtuelle) persis-

tente Attribut der Bean wird ein *public* Attribut definiert. Der Datentyp ist der gleiche wie beim Rückgabewert der *get*-Methode. Wenn der Primärschlüssel nur aus einem einzelnen Attribut besteht, kann dessen Datentyp als Primärschlüsselklasse verwendet werden. In diesem Fall muss keine spezielle Primärschlüsselklasse implementiert werden.

Abbildung 5.8 gibt einen vollständigen Überblick über die Klassen und Interfaces einer Entity-Bean mit Container-Managed-Persistence nach der Version 2.0, die den Local-Client-View unterstützt. Rechts steht die Entity-Bean-Klasse, auf der linken Seite befinden sich die Schnittstellen, die vom Client benutzt werden und durch das Local-Home- bzw. Local-Objekt des EJB-Containers implementiert werden. Die Entity-Bean-Klasse des Local-Client-Views ist identisch mit der des Remote-Client-Views. Die Unterschiede zwischen den beiden Fällen werden vom EJB-Container durch die Implementierung der einschlägigen Interfaces abgebildet. Die Bean-Klasse bzw. die Bean-Instanz bleibt davon unberührt. Die Unterschiede für den Client einer Entity-Bean zwischen Local- und Remote-Client-View sind die gleichen wie bei Session-Beans. Die Unterschiede wurden bereits in Kapitel 4 diskutiert.

Die Primärschlüsselklasse müsste im Fall des Local-Client-View *java.io.Serializable* nicht implementieren, da Objekte dieser Klasse nur innerhalb eines Java-Prozesses verwendet werden. Da sich die Spezifikation dazu nicht ausdrücklich äussert, wurde diese Anforderung an den Primärschlüssel aus dem Remote-Client-View übernommen.

Jede Entity-Bean wird im Deployment-Deskriptor (siehe Listing 5.1) spezifiziert (*entity*). Hier wird ein eindeutiger Name für die Entity-Bean definiert (*ejb-name*). Außerdem werden das Home-Interface (*home*), das Remote-Interface (*remote*), die Bean-Klasse (*ejb-class*) und die Primärschlüsselklasse (*prim-key-class*) vollständig qualifiziert angegeben. Im Element *persistence-type* wird unterschieden, ob es sich um eine Entity-Bean mit Container-Managed-Persistence (*Container*) oder mit Bean-Managed-Persistence (*Bean*) handelt. Weiterhin muss angegeben werden, dass die Entity-Bean den Persistenzmechanismus der Version 2.0 der EJB-Spezifikation benutzt (*cmp-version*). Zuletzt muss ein Name für das Persistenz-Schema der Entity-Bean vergeben werden (*abstract-schema-name*). Dieser Name wird später vom Deployer für die Abbildung der Daten der Entity-Bean auf das jeweilige Persistenzmedium benutzt. Außerdem spielt dieser Name bei der EJB-QL (siehe unten) eine wichtige Rolle.

```
<?xml version="1.0" ?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0/
/EN" "http://java.sun.com/j2ee/dtds/ejb-jar_2_0.dtd">

<ejb-jar>

  <enterprise-beans>
    <entity>
      <ejb-name>Name der Bean</ejb-name>
```

```
        <home>Home-Interface</home>
        <remote>Remote-Interface</remote>
        <ejb-class>Bean-Klasse</ejb-class>
        <persistence-type>Container oder Bean</persistence-type>
        <prim-key-class>Primärschlüsselklasse</prim-key-class>
        <reentrant>True oder False</reentrant>
        <cmp-version>2.x</cmp-version>
        <abstract-schema-name>
            Name des Persistenz-Schemas
        </abstract-schema-name>
        ...
    </enterprise-beans>
    ...
</ejb-jar>
```

Listing 5.1: Schematischer Deployment-Deskriptor für eine Entity-Bean

Eine Entity-Bean kann, genauso wie eine Session-Bean, über das Bean-Environment von außen konfiguriert werden. Die Einträge für das Bean-Environment werden im Deployment-Deskriptor gemacht. Die Handhabung des Bean-Environments wurde im Kapitel 4, *Session-Beans*, behandelt und wird in diesem Kapitel nicht wiederholt.

Die folgenden Abschnitte stellen die Programmierung mit den entsprechenden Definitionen im Deployment-Deskriptor für die Aspekte der Entity-Bean mit Container-Managed-Persistence 2.0 vor. Dazu gehören die persistenten Attribute einer Entity-Bean, das Zustandsmanagement des EJB-Containers, das Erzeugen und Löschen einer Entity-Bean, die Suchmethoden (Finder- und Select-Methoden) und die Methoden der Anwendungslogik.

5.3.2 Attribute

Die Namen aller persistenten Attribute, für die in der Bean-Klasse abstrakte *getter-/setter*-Methoden definiert wurden, werden im Deployment-Deskriptor deklariert. Jeder Name muss mit einem kleinen Buchstaben beginnen. Dadurch können die Tools des EJB-Containers bei der Installation der Entity-Bean die für die Speicherung der Attribute zuständigen abstrakten Methoden identifizieren. Die Deklaration erfolgt innerhalb des *entity*-Abschnitts und wird in Listing 5.2 schematisch dargestellt.

```
...
    <entity>
      ...
      <cmp-field>
        <description>Beschreibung des Attributs 1</description>
        <field-name>Attributname 1 (z.B. name)</field-name>
      </cmp-field>
      <cmp-field>
        <description>Beschreibung des Attributs 2</description>
```

```
        <field-name> Attributname 2 (z.B. vorname)</field-name>
      </cmp-field>
      ...
    </entity>
...
```

Listing 5.2: Definition eines Attributs einer Entity-Bean im Deployment-Deskriptor

Für jedes persistente Attribut werden zwei abstrakte Zugriffsmethoden in der Entity-Bean-Klasse definiert. Ihr Name beginnt mit *get* (lesender Zugriff) bzw. *set* (schreibender Zugriff), gefolgt vom groß geschriebenen Namen des Attributs.

```
...
public abstract void set<Attributname>(<Typ>)
public abstract <Typ> get<Attributname>();
...
```

Listing 5.3: Zugriffsmethoden für ein Attribut einer Entity-Bean

Ein persistentes Attribut einer Entity-Bean, für das z.B. die abstrakten Methoden *getMyAttribute* und *setMyAttribute* definiert wurden, wird im Deployment-Deskriptor der Entity-Bean unter dem Element *field-name* mit dem Wert *myAttribute* deklariert.

Der Client hat keinen direkten Zugriff auf die *getter-/setter*-Methoden zum Lesen und Schreiben der persistenten Attribute einer Entity-Bean. Die Entity-Bean muss entsprechende Methoden im Local- oder Remote-Interface zur Verfügung stellen, die das Schreiben und Lesen der persistenten Attribute indirekt erlauben, indem sie Lese- und Schreibzugriffe an die *getter-/setter*-Methoden delegieren. Die Attribute des Primärschlüssels dürfen nach dem Erzeugen der Entity-Bean nicht mehr verändert werden, da sonst die Integrität der Daten gefährdet ist.

5.3.3 Zustandsmanagement

An oberster Stelle in Abbildung 5.7 bzw. Abbildung 5.8 stehen die Methoden für das Zustandsmanagement. Bei Container-Managed-Persistence ruft der EJB-Container diese Methoden auf, um die Bean-Instanz über Zustandsübergänge zu informieren. Hier sind die Methoden zur Verwaltung des Kontexts und für die Aktivierung und Passivierung zu nennen, die schon von den Session-Beans bekannt sind. Hinzu kommen die Methoden *ejbLoad* und *ejbStore*, die zum Persistenzmechanismus der Bean gehören.

```
void setEntityContext(EntityContext ctx)
```

Die Methode *setEntityContext* wird vom EJB-Container aufgerufen, wenn die Bean-Instanz erzeugt wird und in den Zustand *Pooled* wechselt. Sie dient zur Initialisierung der Bean. Der EJB-Container übergibt der Entity-Bean ihren Kontext. Diese Methode sollte nur den Kontext speichern und darüber hinaus keine Funktionalität besitzen.

`void unsetEntityContext()`

Diese Methode wird vom EJB-Container aufgerufen, wenn er die Bean-Instanz nicht mehr benötigt und aus dem Pool nehmen will. Der Aufruf teilt der Bean-Instanz mit, dass der assoziierte Kontext ungültig wird.

`void ejbActivate()`

Der EJB-Container ruft diese Methode auf, um der Entity-Bean-Instanz mitzuteilen, dass sie eine bestimmte Entity-Bean-Identität bekommen hat. Welche Identität sie nun hat, kann sie über den Entity-Kontext erfahren.

Die Bean-Instanz wechselt in den Zustand *Ready-Async*. In diesem Zustand kann im *EntityContext* schon der Primärschlüssel mit *getPrimaryKey* und das Remote- bzw. Local-Interface mit *getEJBObject* bzw *getEJBLocalObject* abgefragt werden. Die Bean-Instanz wurde jedoch noch nicht mit der Datenbank synchronisiert, und somit sind die Werte der persistenten Attribute noch nicht gesetzt.

Diese Methode kann genutzt werden, um Datenbankverbindungen zu öffnen oder andere Ressourcen zu reservieren. Zu beachten ist dabei, dass der Methode keine Transaktion zur Verfügung steht.

`void ejbPassivate()`

Diese Methode bildet das Gegenstück zu *ejbActivate*. Sie wird vom EJB-Container aufgerufen, wenn die Bean-Instanz vom Zustand *Ready* zurück in den Zustand *Pooled* wechselt. Die Bean-Instanz besitzt danach keine Bean-Identität mehr. In dieser Methode kann jedoch die Bean-Identität noch verwendet werden, da der Zustandswechsel beim Aufruf dieser Methode noch nicht abgeschlossen ist.

Diese Methode wird verwendet, um die Ressourcen wieder freizugeben, die mit *ejbActivate* oder später im Zustand *Ready* reserviert wurden.

`void ejbLoad()`

Die Methode *ejbLoad* wird vom EJB-Container aufgerufen, nachdem der Zustand der Bean-Instanz mit der Datenbank synchronisiert wurde. Dies ist erforderlich, falls die Bean-Instanz noch nicht initialisiert wurde oder falls der Datenbankinhalt sich durch einen parallelen Zugriff geändert hat. Der EJB-Container kann die Bean zu jedem Zeitpunkt im Zustand *Ready* mit dem Persistenzmedium synchronisieren. Häufig geschieht dies, bevor eine Methode der Anwendungslogik aufgerufen wird. Die Bean wechselt vom Zustand *Ready-Async* in den Zustand *Ready-Sync*.

Die Entity-Bean kann diese Methode nutzen, um z. B. transiente Attribute aus persistenten Attributen (die der EJB-Container soeben mit dem Persistenzmedium synchronisiert hat) neu zu berechnen.

```
void ejbStore()
```

Wenn ein Client den Zustand einer Entity-Bean verändert, müssen die veränderten Attributwerte in die Datenbank zurückgeschrieben werden. Dadurch wechselt die Bean-Instanz vom Zustand *Ready-Update* in den Zustand *Ready-Sync*.

5.3.4 Erzeugen und Löschen

Die Methoden zum Erzeugen und Löschen von Entity-Bean-Identitäten gehören zur Identitätenverwaltung und stehen in Abbildung 5.7 bzw. in Abbildung 5.8 in der zweiten Gruppe. Diese Methoden finden sich im (Local-) Home-Interface wieder und stehen so dem Client zur Verfügung, schon bevor er eine Bean-Instanz kennt.

```
<primKeyClass> ejbCreate(<args>)
```

Zum Erzeugen von Entity-Beans gibt es eine oder mehrere Methoden, die alle *ejbCreate* heißen und als Rückgabewert die Primärschlüsselklasse der Bean definieren. Die Methoden unterscheiden sich durch ihre Parameter und können vom Bean-Entwickler frei definiert werden. Die Methoden sind nicht Bestandteil des Entity-Bean-Interface, da die Parameter- und Rückgabetypen für jede Entity-Bean-Klasse unterschiedlich sind.

Die Methoden stehen dem Client zur Verfügung und werden deshalb auch im (Local-) Home-Interface der Entity-Bean deklariert. Die Signaturen der Methoden in der Bean-Klasse und im (Local-) Home-Interface unterscheiden sich durch den Rückgabewert und den Namen. Die Methoden im (Local-)Home-Interface haben das Remote- bzw. Local-Interface der Bean als Rückgabewert und heißen *create*. Die Methoden an der Bean-Klasse verwenden den Primärschlüssel als Rückgabewert und haben *ejb* als Präfix im Namen. Der EJB-Container leistet die erforderliche Konvertierung zwischen dem Primärschlüssel als Rückgabewert der *ejbCreate*-Methode der Entity-Bean und dem Remote- bzw. Local-Objekt der *create*-Methode des (Local-) Home-Interfaces.

Der Bean-Entwickler hat die Aufgabe in dieser Methode alle oder einen Teil der persistenten Attribute mit Hilfe der übergebenen Parameter zu initialisieren. Der Transaktionskontext für die *ejbCreate*-Methoden wird im Deployment-Deskriptor festgelegt.

```
void ejbPostCreate(<args>)
```

Zu jeder *ejbCreate*-Methode muss der Bean-Entwickler eine *ejbPostCreate*-Methode mit den gleichen Parametertypen und keinem Rückgabewert definieren. Die *ejbPostCreate*-Methode wird vom EJB-Container immer nach der entsprechenden *ejbCreate*-Methode mit dem gleichen Transaktionskontext ausgeführt.

In diesen Methoden können weitere Initialisierungsschritte ausgeführt werden. Im Gegensatz zu den *ejbCreate*-Methoden steht diesen Methoden die Bean-Identität zur Verfügung. Mit der Ausführung von *ejbCreate* und *ejbPostCreate* wechselt die Bean-Instanz vom Zustand *Pooled* in den Zustand *Ready-Update*.

```
void ejbRemove()
```

Zur Verwaltung von Bean-Identitäten gehören auch die Methoden zum Löschen von Beans. Der Client gibt den Auftrag zum Löschen einer Entity-Bean-Identität durch den Aufruf der Methode *remove* im (Local-) Home-, Remote- bzw. Local-Interface. Die Methode *ejbRemove* an der Bean-Instanz wird dann vom EJB-Container aufgerufen, bevor die Entity-Bean-Identität gelöscht wird. Die Bean-Instanz geht nach dem Löschen über in den Zustand *Pooled* über. Die Methode *ejbPassivate* wird bei diesem Zustandsübergang nicht aufgerufen.

Bei Container-Managed-Persistence muss der Bean-Entwickler das Löschen nicht selbst implementieren. Der EJB-Container sorgt für die Löschung der Daten. Die Methode *ejbRemove* gibt dem Bean-Entwickler die Möglichkeit, zusätzliche Ressourcen freizugeben, bevor die Entity-Bean gelöscht wird.

Auch die Methode *ejbRemove* kann vom EJB-Container mit einer Transaktion aufgerufen werden. Das zugehörige Transaktionsattribut im Deployment-Deskriptor definiert hier das Verhalten des EJB-Containers (siehe Kapitel 7).

5.3.5 Suchmethoden

Finder-Methoden

Es gibt eine oder mehrere Suchmethoden. Per definitionem kann eine Bean immer anhand ihres Primärschlüssels gefunden werden. Dementsprechend muss die zugehörige Methode *findByPrimaryKey* immer existieren. Diese Suche kann immer nur eine einzelne Entity-Bean als Treffer haben. Zusätzlich kann es andere Suchmethoden geben, die nach anderen Kriterien suchen. Per Namenskonvention beginnt deren Name immer mit *find*, gefolgt von einem beschreibenden Namen für die Suchmethode. Diese Methoden können für einzelne Treffer oder Treffermengen definiert werden.

Die Suchmethoden werden im (Local-) Home-Interface deklariert und die zugehörige Suche im Deployment-Deskriptor mit EJB-QL definiert. Eine Implementierung in der Bean-Klasse ist bei Container-Managed-Persistence nicht erforderlich, oder besser gesagt nicht möglich.

Suchen, die nur einen Treffer haben können, verwenden direkt das Remote- bzw. Local-Interface der Bean als Datentyp für den Rückgabewert. Kann eine Suche mehrere Treffer haben, so wird *java.util.Enumeration* (Java 1.1.) oder *java.util.Collection* (Java 1.2) als Datentyp für den Rückgabewert verwendet. Die Methode *findByPrimaryKey* besitzt genau einen Parameter, und zwar den Primärschlüssel. Für die anderen Suchmethoden können beliebige Parameter definiert werden.

Listing 5.4 zeigt ein fiktives Beispiel eines Home-Interfaces, welches die Suchmethoden *findByPrimaryKey* (muss) und *findByAttributeA* (kann) definiert. Letztere Methode kann mehrere Treffer liefern und verwendet deshalb *java.util.Collection* als Rückgabe-

wert. Eine Suchmethode deklariert immer eine Ausnahme vom Typ *javax.ejb.FinderException*. Handelt es sich um den Remote-Client-View, wird zusätzlich dazu die Ausnahme *java.rmi.RemoteException* deklariert. Eine Ausnahme vom Typ *FinderException* wird dann ausgelöst, wenn bei der Ausführung der Suchmethode ein Systemfehler auftritt. Bei Suchmethoden, die genau ein Element als Treffer zurückliefern, wird eine Ausnahme vom Typ *javax.ejb.ObjectNotFoundException* ausgelöst, wenn kein passendes Element gefunden wurde. *ObjectNotFoundException* ist von *FinderException* abgeleitet. Bei Suchmethoden, die mehrere Elemente zurückliefern, wird eine leere Menge zurückgegeben, falls keine passenden Elemente gefunden werden können.

```
...
public interface SomeHome extends EJBHome
{
    ...
    public Some findByPrimaryKey(SomePK pk)
        throws FinderException, RemoteException;

    public Collection findByAttributeA(double attrValue)
        throws FinderException, RemoteException;
    ...
}
```

Listing 5.4: Fiktives Beispiel für ein Home-Interface mit Find-Methoden

Im Deployment-Deskriptor wird für jede Suchmethode (außer der Methode *findByPrimaryKey*) eine Suche (*query*) definiert. Die Suchanfrage (*ejb-ql*) wird über das Element *query-method* mit der jeweiligen Suchmethode assoziiert. Der Name muss mit dem Namen der Suchmethode im Home-Interface übereinstimmen. Alle Parameter (*method-params*) werden in der Reihenfolge, wie sie in der Methodendefinition auftreten, angegeben. Für jeden Parameter muss der Datentyp (*method-param*) bestimmt werden.

Die Definition der Suche wir in EJB-QL formuliert, die später in diesem Kapitel im Detail behandelt wird. Listing 5.5 zeigt die Definition der Suche für die Methode *findByAttributeA* als Vorgriff auf diesen Abschnitt.

```
...
<enterprise-beans>
  <entity>
    ...
    <query>
      <query-method>
        <method-name>findByAttributeA</method-name>
        <method-params>
          <method-param>double</method-param>
        </method-params>
      </query-method>
      <ejb-ql>
```

```
          SELECT OBJECT(a) FROM SomeBean AS a WHERE a.attributeA = ?1
        </ejb-ql>
      </query>
    </entity>
  </enterprise-beans>
  ...
```

Listing 5.5: EJB-QL für die Methode findByAttributeA

Die definierten Parameter der Suchmethode können in der Suche verwendet werden. Auf sie wird über ihre Position in der Definition referiert. Der erste Parameter wird mit *?1* angesprochen, der zweite mit *?2* und der n-te mit *?<n>*.

Select-Methoden

Select-Methoden ähneln den Finder-Methoden, sind jedoch vielschichtiger und werden von einer Bean intern verwendet, z.B. in den Business-Methoden. Sie sind nicht für die Benutzung durch den Client gedacht und werden weder im (Local-) Home-, Remote- oder Local-Interface veröffentlicht. Die Select-Methoden werden als abstrakte Methoden mit dem Präfix *ejbSelect* in der Bean-Klasse deklariert und die zugehörige Suche im Deployment-Deskriptor mit EJB-QL definiert. Die Tools des EJB-Containers sorgen zum Zeitpunkt der Installation im EJB-Container für die Generierung einer konkreten Methode, die dann zur Laufzeit von der Entity-Bean-Instanz verwendet werden kann.

Select-Methoden können wie Finder-Methoden eine oder mehrere Referenzen auf andere Entity-Beans zurückliefern. Sie können aber auch Werte persistenter Attribute von Entity-Beans zurückliefern. Select-Methoden können beliebige Parameter deklarieren. Die Parameter werden wie bei den Finder-Methoden zur Einschränkung der Ergebnismenge benutzt.

Listing 5.6 zeigt ein fiktives Beispiel einer Entity-Bean, die die Select-Methoden *ejbSelectAllABeansByName* und *ejbSelectAllABeanNames* definiert. Erstere Methode soll alle Entity-Beans vom Typ *ABean* zurückliefern, deren Name dem übergebenen Attribut entspricht. Letztere Methode liefert alle Namen zurück, die von *ABean*-Instanzen verwendet werden. Sie liefert also eine Menge von *String*-Objekten zurück, bei denen es sich um Werte des persistenten Attributes *name* der Entity-Bean *ABean* handelt.

```
...
public abstract BBean implements EntityBean
{
  ...
  public abstract Collection ejbSelectAllABeansByName(String name)
    throws FinderException;

  public abstract Collection ejbSelectAllABeanNames()
```

```
    throws FinderException
  ...
}
```

Listing 5.6: Pseudocode für die Bean-Klasse einer Select-Methode

Listing 5.7 zeigt die Definition der Suchen für die Select-Methoden im Deployment-Deskriptor mittels EJB-QL. Wie bereits erwähnt, wird EJB-QL im Detail in einem Abschnitt weiter unten in diesem Kapitel behandelt. Dieses Beispiel ist ein kleiner Vorgriff auf die Ausführungen zu EJB-QL.

```
...
<enterprise-beans>
  <entity>
    ...
    <query>
      <query-method>
        <method-name>ejbSelectAllABeansByName</method-name>
        <method-params>
          <method-param>java.lang.String</method-param>
        </method-params>
      </query-method>
      <ejb-ql>
        SELECT OBJECT(a) FROM ABean AS a WHERE a.name = ?1
      </ejb-ql>
    </query>
    <query>
      <query-method>
        <method-name> ejbSelectAllABeanNames</method-name>
        <method-params>
           <!-- Diese Methode hat keine Parameter -->
        </method-params>
      </query-method>
      <ejb-ql>
        SELECT DISTINCT a.name FROM ABean AS a
      </ejb-ql>
    </query>
  </entity>
</enterprise-beans>
...
```

Listing 5.7: EJB-QL für die Select-Methoden der Bean

Die Select-Methode *ejbSelectAllABeansByName* liefert Referenzen auf andere Entity-Beans zurück. Dabei kann es sich um Entity-Beans handeln, die den Local- oder den Remote-Client-View unterstützen. Hat eine Entity-Bean ein Remote- und ein Local-Interface, wird standardmäßig eine Referenz auf das Local-Interface der Entity-Bean zurückgeliefert. Mit dem Element *result-type-mapping* innerhalb des Elements *query* kann ausdrücklich festgelegt werden, ob eine Referenz auf das Local- oder das

Remote-Interface der Entity-Bean zurückgeliefert wird. Zulässige Werte für das Element *result-type-mapping* sind *Local* für eine Referenz auf das Local-, *Remote* für eine Referenz auf das Remote-Interface der Bean.

Über Select-Methoden kann nach allen Entity-Beans und deren Attributen gesucht werden, die innerhalb des gleichen Deployment-Deskriptors definiert sind, wie die Entity-Bean, welche die jeweilige Select-Methode definiert. Select-Methoden können von einer Entity-Bean in den Home-, den Business-Methoden und in den Methoden *ejbLoad* und *ejbStore* verwendet werden.

Eine Select-Methode hat keinen eigenen transaktionalen Kontext. Ob sie in einer Transaktion ausgeführt wird hängt vom transaktionalen Kontext der Methode ab, die die Select-Methode aufruft (zu Transaktionen siehe Kapitel 7).

5.3.6 Anwendungslogik und Home-Methoden

Die letzte Gruppe von Methoden ist die Anwendungslogik. Hier wird die eigentliche Funktionalität der Entity-Bean implementiert. Normalerweise beziehen sich diese Methoden auf eine bestimmte Bean-Instanz. In den Anwendungsmethoden (Business-Methoden) werden die Zugriffsmethoden zum Lesen und Schreiben von Attributen und Beziehungen verwendet. Die Business-Methoden werden vom Bean-Entwickler in der Bean-Klasse implementiert und im Remote- bzw. Local-Interface veröffentlicht. Die Definition in der Bean-Klasse hat die gleiche Signatur wie die Definition im Remote- bzw Local-Interface (vergleiche Abbildung 5.7 bzw. Abbildung 5.8). Der Name der Methoden darf nicht einem vorbelegten Methodennamen entsprechen (nicht *get*<name>, *set*<name> oder *ejb*<name>).

In Ausnahmefällen gibt es jedoch auch Anwendungslogik, die sich nicht auf eine bestimmte Instanz bezieht. Diese kann als sogenannte Home-Methode realisiert werden. Home-Methoden werden im (Local-) Home-Interface veröffentlicht und stehen dem Client zur Verfügung, auch wenn er keine Entity-Bean-Instanz zur Verfügung hat. Ihr Name in der Bean-Klasse beginnt immer mit *ejbHome* gefolgt von der Bezeichnung für die Methode. Im Home-Interface entfällt das Präfix (vgl. auch Abbildung 5.7 bzw. Abbildung 5.8). Die Bezeichnung für die Methode darf nicht einem vorbelegten Methodennamen entsprechen (nicht *create*<name> oder *find*<name>).

Bei der Implementierung einer Home-Methode ist zu beachten, dass das Objekt keine Identität hat. Die jeweilige *ejbHome*-Methode wird an einer Instanz im Zustand *Pooled* aufgerufen. Deshalb dürfen keine Zugriffsmethoden außer den Select-Methoden aufgerufen werden. Auch das Erzeugen oder Löschen von Bean-Instanzen ist nicht erlaubt.

Die Anwendungsmethoden im Home- und Remote-Interface des Remote-Client-Views definieren immer die *RemoteException*, während die Methoden in der Bean-Klasse die *EJBException* verwenden, um Systemfehler zu signalisieren. Die *EJBException* wird vom

EJB-Container in eine *RemoteException* umgewandelt. Anwendungsmethoden des Local-Home- und des Local-Interfaces definieren keine Ausnahme vom Typ *RemoteException*. Tritt ein Systemfehler auf, wird die *EJBException* direkt an den lokalen Client weitergegeben.

Der EJB-Container kontrolliert alle Aufrufe der Methoden einer Bean. Er delegiert die Methodenaufrufe am (Local-) Home-, Remote bzw. Local-Interface an die entsprechenden Methoden der Entity-Bean-Instanz. Der Client greift niemals direkt auf die Bean-Instanz zu. Diese Position als Vermittler zwischen Client und Bean-Instanz nutzt der EJB-Container für vielseitige Aufgaben. Beispielsweise stellt der EJB-Container sicher, dass die Methode mit dem richtigen Transaktionskontext ausgeführt wird. Er wertet das Transaktionsattribut der Methode im Deployment-Deskriptor aus, vergleicht dieses mit dem Transaktionskontext des Clients und führt dann die nötigen Aktionen durch, um den transaktionalen Bedürfnissen der Methode gerecht zu werden (siehe Kapitel 7). Andere Beispiele für die Aufgaben des EJB-Containers sind die Realisierung des Zugriffsschutzes für Beans (siehe Kapitel 8) oder das Protokollieren von Fehlern.

5.3.7 Beispiel *Counter*

Zur Veranschaulichung der grundlegenden Aspekte einer Entity-Bean mit Container-Managed-Persistence 2.0, die in diesem Abschnitt behandelt wurden, wollen wir ein einfaches Beispiel betrachten. Eine Anwendung mit vielen Remote-Clients muss Zählerstände für diverse Dinge verwalten. Die Zählerstände sollen allen Clients zur Verfügung stehen und zentral verwaltet werden. Jeder Client muss die Möglichkeit haben, einen bestimmten Zählerstand zu verändern. Der Zählerstand darf dabei nicht kleiner als 0 und nicht größer als 100 sein.

Die Entity-Bean *Counter* erfüllt diese Anforderungen. Sie unterstützt den Remote-Client-View. Mit ihrer Hilfe ist es möglich, auf dem Applikationsserver diverse Zählerstände zu verwalten, die allen Clients zugänglich sind. Jeder Client kann den Stand eines bestimmten Zählers lesen und verändern.

Zunächst wird das Remote-Interface der Counter-Bean definiert, welches die Funktionalität gegenüber den Remote-Clients repräsentiert (siehe Listing 5.8). Es definiert die Methode *inc* zum Inkrementieren eines bestimmten Zählers, die Methode *dec* zum Dekrementieren eines bestimmten Zählers und die Methode *getValue*, die den aktuellen Stand eines bestimmten Zählers zurückgibt.

```
package ejb.counter;

import java.rmi.RemoteException;
import javax.ejb.EJBObject;

public interface Counter extends EJBObject {
```

```
    public void inc()
        throws RemoteException, CounterOverflowException;

    public void dec()
        throws RemoteException, CounterOverflowException;

    public int getValue()
        throws RemoteException;

}
```

Listing 5.8: Remote-Interface der Counter-Bean

Neben der RemoteException deklarieren die Methoden *inc* und *dec* die Ausnahme *CounterOverflowException* (siehe Listing 5.9). Sie wird dann ausgelöst, wenn der Stand des Zählers überläuft, d.h. wenn der Wert nach einer Dekrementierung unter 0 sinken bzw. nach einer Inkrementierung über 100 steigen würde.

```
package ejb.counter;

public class CounterOverflowException extends Exception {

    public CounterOverflowException() {
        super();
    }

    public CounterOverflowException(String message) {
        super(message);
    }

}
```

Listing 5.9: Ausnahme CounterOverflowException

Ein weiterer Bestandteil der Counter-Bean ist das Home-Interface (siehe Listing 5.10). Es definiert die Methode *create*, mit deren Hilfe ein neuer Zähler erzeugt werden kann. Der Zähler hat eine ID in Form eines String-Objektes. Anhand dieser ID kann der Zähler eindeutig identifiziert und über die Methode *findByPrimaryKey* wiedergefunden werden. Zudem wird dem Zähler in der Methode *create* der initiale Zählerstand übergeben. Außerdem definiert das Home-Interface der Counter-Bean eine Home-Methode mit dem Namen *getAllCounterIds*. Sie liefert die IDs aller existierenden Zählerstände in Form von String-Objekten zurück. Diese Funktionalität ist nicht an einen bestimmten Zähler gekoppelt und wird deshalb in einer Home-Methode implementiert. Ein Client hat über die Home-Methode die Möglichkeit herauszufinden, welche Zähler es gibt. Falls er Interesse an einem bestimmten Zähler hat, kann er sich mit Hilfe der ID über die Methode *findByPrimaryKey* Zugriff auf den Zähler verschaffen.

```
package ejb.counter;

import javax.ejb.CreateException;
import javax.ejb.EJBHome;
import javax.ejb.FinderException;
import java.rmi.RemoteException;

public interface CounterHome extends EJBHome {

    public Counter create(String counterId, int initCounterValue)
        throws CreateException, RemoteException;

    public Counter findByPrimaryKey(String counterId)
        throws FinderException, RemoteException;

    public java.util.Collection getAllCounterIds()
        throws RemoteException;

}
```

Listing 5.10: Home-Interface der Counter-Bean

Nachdem Home- und Remote-Interface definiert worden sind, folgt nun die Bean-Klasse (siehe Listing 5.11). Sie ist eine abstrakte Klasse und implementiert das Interface *javax.ejb.EntityBean*. Sie definiert die zur *create* Methode des Home-Interfaces passende *ejbCreate*-Methode (plus der von der Spezifikation vorgeschriebenen *ejbPostCreate*-Methode). Die Counter-Bean hat zwei persistente Attribute, nämlich *counterId* und *counterValue*. Das Attribut *counterId* ist zugleich der Primärschlüssel der Counter-Bean. Für jedes Attribut werden die abstrakten *getter-/setter*-Methoden definiert.

```
package ejb.counter;

import javax.ejb.CreateException;
import javax.ejb.EntityBean;
import javax.ejb.EntityContext;
import javax.ejb.FinderException;

public abstract class CounterBean implements EntityBean {

    public final static int VALUE_MAX = 100;
    public final static int VALUE_MIN = 0;

    private transient EntityContext ctx;

    //Die Create Methode des Home-Interfaces

    public String ejbCreate(String counterId,
                            int initCounterValue)
        throws CreateException
```

```
{
    if(counterId == null) {
        throw new CreateException("counterId is null!");
    }
    if(initCounterValue < VALUE_MIN ||
       initCounterValue > VALUE_MAX)
    {
        String s = "initCounterValue out of range: "
                 + initCounterValue;
        throw new CreateException(s);
    }
    this.setCounterId(counterId);
    this.setCounterValue(initCounterValue);
    return null;
}

public void ejbPostCreate(String accountId,
                          int initCounterValue)
{}

//Abstrakte getter-/setter-Methoden
public abstract void setCounterId(String id);
public abstract String getCounterId();
public abstract void setCounterValue(int value);
public abstract int getCounterValue();
//Abstrakte select-Methoden
public abstract java.util.Collection ejbSelectAllCounterIds()
    throws FinderException;

//Methoden des Remote-Interfaces

public void inc() throws CounterOverflowException {
    if(this.getCounterValue() < VALUE_MAX) {
        this.setCounterValue(this.getCounterValue() + 1);
    } else {
        String s = "Cannot increase above " + VALUE_MAX;
        throw new CounterOverflowException(s);
    }
}

public void dec() throws CounterOverflowException {
    if(this.getCounterValue() > VALUE_MIN) {
        this.setCounterValue(this.getCounterValue() - 1);
    } else {
        String s = "Cannot decrease below " + VALUE_MIN;
        throw new CounterOverflowException(s);
    }
}

public int getValue() {
    return this.getCounterValue();
```

```
    }

    //Methoden des Entity-Bean-Interface

    public void ejbActivate() {}

    public void ejbPassivate() {}

    public void setEntityContext(EntityContext ctx) {
        this.ctx = ctx;
    }

    public void unsetEntityContext() {
        this.ctx = null;
    }

    public void ejbLoad() {}

    public void ejbStore() {}

    public void ejbRemove() {}

    // Home-Methoden

    public java.util.Collection ejbHomeGetAllCounterIds() {
        java.util.ArrayList al = new java.util.ArrayList();
        try {
            java.util.Collection col =
                this.ejbSelectAllCounterIds();
            java.util.Iterator it = col.iterator();
            while(it.hasNext()) {
                al.add(it.next());
            }
        } catch(FinderException ex) {
            ex.printStackTrace();
        }
        return al;
    }

}
```

Listing 5.11: Bean-Klasse der Counter-Bean

Neben den abstrakten Methoden für die persistenten Attribute wird eine abstrakte *ejbSelect*-Methode definiert. Sie liefert alle Counter-IDs in Form von String-Objekten zurück und wird von der Home-Methode *getAllCounterIds* zur Erfüllung ihrer Aufgabe benutzt. Die Home-Methode *getAllCounterIds* des Home-Interfaces bekommt in der Bean-Klasse das Präfix *ejbHome*, so wie es die Spezifikation verlangt.

Die Methoden *inc*, *dec* und *getValue* des Remote-Interface werden ebenfalls in der Bean-Klasse implementiert. Die Callback-Methoden des Entity-Bean-Interfaces bleiben weitgehend leer. Bei Container-Managed-Persistence übernimmt der EJB-Container den Großteil der Arbeit, so dass für die Bean-Klasse selbst wenig zu implementieren bleibt.

Zu guter Letzt wird der Deployment-Deskriptor der Counter-Bean definiert (siehe Listing 5.12). Als Besonderheit ist das Element *query* zu nennen, in dem die Methode *ejbSelectAllCounterIds* mittels EJB-QL deklarativ implementiert wird. EJB-QL wird weiter unten in diesem Kapitel im Detail behandelt. Außerdem wird im Assembly-Deskriptor festgelegt, dass die Methoden *inc* und *dec* in einer Transaktion aufgerufen werden müssen. Auch das ein kleiner Vorgriff auf Kapitel 7, in dem Transaktionen ausführlich behandelt werden.

```
<?xml version="1.0" ?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0/
/EN" "http://java.sun.com/j2ee/dtds/ejb-jar_2_0.dtd">

<ejb-jar>
  <enterprise-beans>
    <entity>
      <ejb-name>Counter</ejb-name>
      <home>ejb.counter.CounterHome</home>
      <remote>ejb.counter.Counter</remote>
      <ejb-class>ejb.counter.CounterBean</ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>java.lang.String</prim-key-class>
      <reentrant>False</reentrant>
      <cmp-version>2.x</cmp-version>
      <abstract-schema-name>CounterBean</abstract-schema-name>
      <cmp-field>
        <description>Die Id des Zaehlers</description>
        <field-name>counterId</field-name>
      </cmp-field>
      <cmp-field>
        <description>Der Wert des Zaehlers</description>
        <field-name>counterValue</field-name>
      </cmp-field>
      <primkey-field>counterId</primkey-field>
      <query>
        <query-method>
          <method-name>ejbSelectAllCounterIds</method-name>
          <method-params>
          </method-params>
        </query-method>
        <ejb-ql>
          SELECT DISTINCT cb.counterId FROM CounterBean AS cb
        </ejb-ql>
      </query>
    </entity>
```

```
  </enterprise-beans>

  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>Counter</ejb-name>
        <method-name>inc</method-name>
      </method>
      <method>
        <ejb-name>Counter</ejb-name>
        <method-name>dec</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
  </assembly-descriptor>

</ejb-jar>
```

Listing 5.12: Deployment-Deskriptor der Counter-Bean

Damit wären die Bestandteile der Entity-Bean Counter aus Sicht des Bean-Entwicklers komplett. Eine Angabe, die zum Deployment noch fehlt, ist die Abbildung der persistenten Attribute der Bean auf die Datenbank. Diese Angaben werden i.d.R. vom Deployer gemacht. Wie diese Angaben zu machen sind, unterscheidet sich von Applikationsserver zu Applikationsserver. Listing 5.13 zeigt ein fiktives Beispiel, wie solche Angaben gemacht werden könnten.

```
<database-mapping>
    <bean-mapping>
        <ejb-name>Counter</ejb-name>
        <data-source-name>postgres</data-source-name>
        <table-name>counter</table-name>
        <field-map>
            <cmp-field>counterId</cmp-field>
            <dbms-column>id</dbms-column>
        </field-map>
        <field-map>
            <cmp-field>counterValue</cmp-field>
            <dbms-column>value</dbms-column>
        </field-map>
    </bean-mapping>
</database-mapping>
```

Listing 5.13: Fiktives Datenbank-Mapping für die Counter-Bean

Die Attribute der Entity-Bean Counter werden in einer Tabelle mit dem Namen *counter* gespeichert. Das Bean-Attribut *counterId* wird in der Spalte *id*, das Attribut *counterValue* in der Spalte *value* gespeichert. Der Persistenz-Manager soll für die Verbindung zur

Datenbank die bereits vorkonfigurierte JDBC-Datasource mit dem JNDI-Namen *postgres* verwenden. Listing 5.14 zeigt das SQL-Statement, mit Hilfe dessen die Tabelle *counter* erzeugt werden kann.

```
CREATE TABLE COUNTER (
  ID VARCHAR(128) NOT NULL UNIQUE,
  VALUE INTEGER NOT NULL
)
```

Listing 5.14: SQL-Statement zum Erzeugen der Tabelle counter

Abschließend zeigt Listing 5.15, wie ein fiktiver Client die Counter-Bean benutzen könnte.

```
...
  InitialContext ctx = new InitialContext();
  Object o = ctx.lookup("Counter");
  CounterHome counterHome =
    (CounterHome)PortableRemoteObject.narrow(o, CounterHome.class);
  Counter counter = counterHome.create("carsInGarage", 0);

  ...

  try {
      counter.inc();
  } catch(CounterOverflowException ex) {
      throw new IllegalStateException(„garage is full");
  }

  ...

  try {
      counter.dec();
  } catch(CounterOverflowException ex) {
      this.garageIsEmpty = true;
  }

...
```

Listing 5.15: Fiktives Beispiel eines Clients der Counter-Bean

5.4 Beziehungen zwischen Entity-Beans (EJB 2.0)

Persistente, vom EJB-Container verwaltete Beziehungen (Container-Managed-Relationships) zwischen Entity-Beans sind eine der wesentlichen Neuerungen der Version 2.0 der EJB-Spezifikation. Sie ermöglichen die Abbildung komplexer Datenstrukturen durch die Verwendung mehrerer Entity-Beans mit Container-Managed-Persistence, die zueinander in Beziehung stehen.

Beziehungen sind gerichtet, d.h. es gibt ein Ausgangsobjekt und ein oder mehrere Zielobjekte. Unidirektionale Beziehungen sind Beziehungen, bei denen nur immer der eine Teil von der Beziehung weiß. Die Navigation ist nur in eine Richtung möglich. Bidirektionale Beziehungen sind Beziehungen, bei denen beide Teile von der Beziehung wissen. Die Navigation ist in beide Richtungen möglich.

Die Entity-Beans, die zueinander in Relation gesetzt werden, müssen im gleichen Deployment-Deskriptor definiert sein. Entity-Beans, die zueinander in Beziehung stehen, sind voneinerander abhängig. Nur wenn sie im gleichen Deployment-Deskriptor definiert sind, kann der EJB-Container zur Laufzeit sicherstellen, dass die verbundenen Entity-Beans verfügbar sind. Wären die verbundenen Entity-Beans in unterschiedlichen Deployment-Deskriptoren definiert, könnten sie in unterschiedlichen Applikationsservern installiert sein. Damit könnte der EJB-Container zur Laufzeit die Verfügbarkeit der verbundenen Entity-Beans nicht mehr garantieren. Der andere Applikationsserver könnte nicht laufen oder die verbundenen Entity-Beans könnten deinstalliert worden sein.

Die Entity-Beans, die zum navigierbaren Teil der Beziehung gehören, müssen den Local-Client-View unterstützen. Andernfalls kann auf sie nicht über eine Beziehung referiert werden. Da Entity-Beans, die zueinander in Beziehung stehen, ohnehin im selben Deployment-Deskriptor definiert sein müssen, befinden sie sich zur Laufzeit auch im selben Applikationsserverprozess. Die Verwendung des Local-Client-View sorgt durch den Wegfall des Netzwerkoverheads für eine optimale Performanz beim Navigieren über Beziehungen. Außerdem ist die Verwaltung der Beziehungen für den EJB-Container durch die Verwendung des Local-Client-Views und die damit verbundene call-by-reference Semantik wesentlich einfacher.

Aus Sicht des Entwicklers besteht eine Beziehung zwischen zwei Entity-Bean-Typen im Sinne dieses Abschnitts aus 3 Komponenten:

Die erste Komponente sind die abstrakten Beziehungsmethoden in der Entity-Bean-Klasse. Eine Beziehung zu einer anderen Entity-Bean bekommt einen Namen ähnlich dem eines persistenten Attributs. Je nach Art der Beziehung (unidirektional oder bidirektional) definiert die Bean-Klasse ein Paar abstrakter Methoden, eine Methode *set<Beziehungsname>* und eine Methode *get<Beziehungsname>*. Die *set*-Methode dient zum Herstellen oder Verändern einer Beziehung auf eine bestimmte Instanz bzw. auf bestimmte Instanzen des anderen Bean-Typs, die *get*-Methode dient zur Navigation über die Beziehung.

Die zweite Komponente ist die formale Beschreibung der Beziehung im Deployment-Deskriptor. Dort wird definiert, welche Entity-Beans zueinander in Beziehung stehen, über welche Attribute (sprich über welche abstrakten Beziehungsmethoden) die Beziehung hergestellt wird, ob es sich um eine uni- oder eine bidirektionale Beziehung handelt und von welcher Kardinalität die Beziehung ist.

Die dritte Komponente ist die Abbildung der Beziehungen auf das Persistenzmedium. Wir werden im weiteren Verlauf dieses Abschnitts davon ausgehen, dass als Persistenzmedium eine relationale Datenbank zum Einsatz kommt. Es muss festgelegt werden, wie die Beziehung gespeichert wird, d.h. ob ein Fremdschlüssel oder eine Relationstabelle verwendet wird. Außerdem muss festgelegt werden, über welche Attribute der Entity-Bean, sprich über welche Tabellenspalten, eine Beziehung definiert wird. Die Beschreibung der dritten Komponente ist Aufgabe des Deployers und in ihrer Form von der Spezifikation nicht festgelegt. Die Beschreibung der Abbildung von Beziehungen auf das Persistenzmedium ist demnach von Applikationsserver zu Applikationsserver verschieden.

In den folgenden Abschnitten wollen wir die Grundlagen der verschiedenen Beziehungstypen diskutieren.

5.4.1 Eins-zu-eins-Beziehungen (One to One)

Unidirektional

Ein Beispiel für eine eins-zu-eins-Beziehung wäre im Fall einer Fahrzeugvermietung die Beziehung zwischen dem Fahrzeug und dem Kunden, der das Fahrzeug gerade gemietet hat. Ein Fahrzeug kann zu einem Zeitpunkt nur an genau einen Kunden vermietet sein. Ein Kunde kann zu einem bestimmten Zeitpunkt nur genau ein Fahrzeug benutzen. Besteht keine Beziehung zwischen einem Fahrzeug und einem Kunden, so ist das Fahrzeug nicht vermietet. Abbildung 5.9 stellt den Sachverhalt graphisch dar. Die Abbildung zeigt eine unidirektionale Beziehung, die über eine Fremdschlüsselbeziehung gespeichert wird. Die Tabelle *fahrzeug* enthält die Spalte *kundennr* (=Fremdschlüssel), die die Beziehung zum jeweiligen Kunden in der Tabelle *kunde* herstellt.

Für den Fall der unidirektionalen Beziehung zwischen der Fahrzeug- und der Kunde-Bean definiert nur die Fahrzeug-Bean die abstrakten Beziehungsmethoden in der Bean-Klasse (siehe Listing 5.16). Der Datentyp der Beziehung ist dabei immer der Typ des Local-Interfaces der bezogenen Entity-Bean.

```
...
    //Persistente Attribute
    public abstract void setKennzeichen(String z);
    public abstract String getKennzeichen();
    //Persistente Beziehungen
    public abstract void setKunde(KundeLocal kunde);
    public abstract KundeLocal getKunde();
...
```

Listing 5.16: Abstrakte Beziehungmethoden der Fahrzeug-Bean

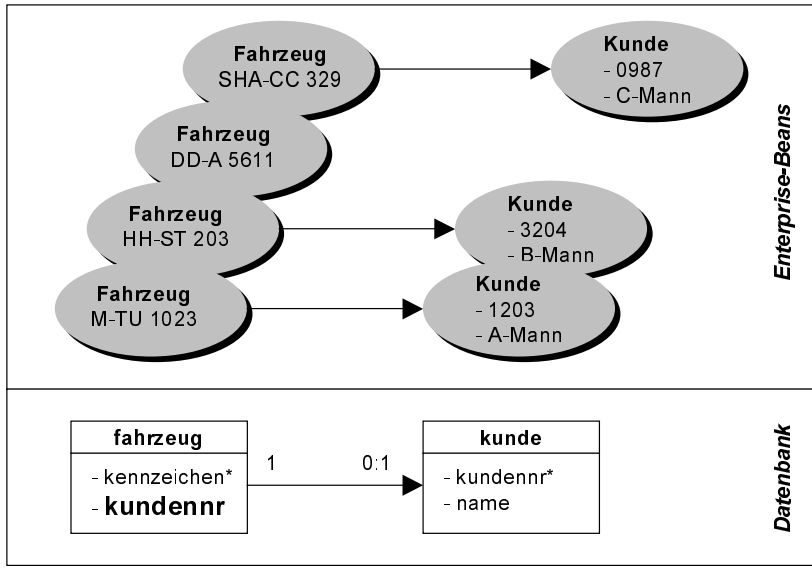

Abbildung 5.9: Beispiel unidirektionale eins-zu-eins-Beziehung

Die Beziehungsrichtung ist von der Fahrzeug-Bean auf die Kunde-Bean. Die Kunde-Bean muss daher den Local-Client-View unterstützen, da sie von der Fahrzeug-Bean aus über Navigation erreichbar ist. Die Fahrzeug-Bean muss im Fall einer unidirektionalen Beziehung den Local-Client-View nicht unterstützen, da sie durch Navigation über die Beziehung nicht erreichbar ist. Besteht keine Beziehung zur Kunde-Bean, so liefert ein Aufruf der Methode *getKunde* den Rückgabewert *null*.

Listing 5.17 zeigt die formale Beschreibung der Beziehung im Deployment-Deskriptor. Das Element *ejb-relation* beinhaltet das Element *ejb-relationship-name* und immer zwei Elemente *ejb-relationship-role* für jeweils eine Seite der Beziehung. Jede Seite der Beziehung bekommt über das Element *ejb-relationship-role-name* einen Namen zugewiesen. Dieser Name ist später für die Abbildung auf die Datenbank von Bedeutung. Für jeden Teil der Beziehung wird über das Element *multiplicity* die Kardinalität angegeben. Mittels *relationship-role-source* wird die Entity-Bean angegeben, die an der Beziehung beteiligt ist. Das Element *cmr-field* benennt das Attribut der Bean, über das die Beziehung gespeichert wird. Fehlt dieses Feld auf einer der beiden Seiten der Beziehung, handelt es sich automatisch um eine unidirektionale Beziehung; von wo nach wo die Beziehung gerichtet ist hängt davon ab, auf welcher Seite das Element *cmr-field* definiert ist. Die Fahrzeug-Bean hat für die Speicherung und Navigation der Beziehung die Methoden *setKunde* und *getKunde* definiert. Damit muss der Wert für das Element *cmr-field-name kunde* sein, da sonst der EJB-Container die Zuordnung zu den Beziehungsmethoden nicht herstellen kann.

```
...
  <enterprise-beans>
    <entity>
      <ejb-name>Fahrzeug</ejb-name>
      ...
    </entity>
    <entity>
      <ejb-name>Kunde</ejb-name>
      <local-home>KundeLocalHome</local-home>
      <local>KundeLocal</local>
      ...
    </entity>
    ...
  </enterprise-beans>
...
  <ejb-relation>
    <ejb-relation-name>Fahrzeug-Kunde</ejb-relation-name>
    <ejb-relationship-role>
      <ejb-relationship-role-name>
        fahrzeug-hat-kunde
      </ejb-relationship-role-name>
      <multiplicity>One</multiplicity>
      <relationship-role-source>
        <ejb-name>Fahrzeug</ejb-name>
      </relationship-role-source>
      <cmr-field>
        <cmr-field-name>kunde</cmr-field-name>
      </cmr-field>
    </ejb-relationship-role>
    <ejb-relationship-role>
      <ejb-relationship-role-name>
        kunde-benutzt-fahrzeug
      </ejb-relationship-role-name>
      <multiplicity>One</multiplicity>
      <relationship-role-source>
        <ejb-name>Kunde</ejb-name>
      </relationship-role-source>
    </ejb-relationship-role>
  </ejb-relation>
...
```

Listing 5.17: Formale Beschreibung der Beziehung Fahrzeug-Kunde (unidirektional)

Die fehlende Komponente ist die Beschreibung dafür, wie die Beziehung auf die Datenbank abgebildet wird. Wie bereits erwähnt, gibt es bezüglich der Form von Seiten der Spezifikation keine Vorgaben. Jeder Applikationsserver hat sein eigenes Format, wie diese Abbildung auf das Persistenzmedium beschrieben wird. Listing 5.18 zeigt ein fiktives Beispiel, anhand dessen das Prinzip der Zuordnung verdeutlicht wird.

```
<bean-mapping>
  <ejb-name>Fahrzeug</ejb-name>
  <data-source-name>postgres</data-source-name>
  <table-name>fahrzeug</table-name>
  <field-map>
    <cmp-field>kennzeichen</cmp-field>
    <dbms-column>kennzeichen</dbms-column>
  </field-map>
</bean-mapping>
<bean-mapping>
  <ejb-name>Kunde</ejb-name>
  <data-source-name>postgres</data-source-name>
  <table-name>kunde</table-name>
  <field-map>
    <cmp-field>kundennr</cmp-field>
    <dbms-column>kundennr</dbms-column>
  </field-map>
  <field-map>
    <cmp-field>name</cmp-field>
    <dbms-column>name</dbms-column>
  </field-map>
</bean-mapping>

...
<relation-mapping>
  <relation-name>Fahrzeug-Kunde</relation-name>
    <relationship-role>
      <relationship-role-name>
        fahrzeug-hat-kunde
      </relationship-role-name>
      <column-map>
        <foreign-key-column>kundennr</foreign-key-column>
        <key-column>kundennr</key-column>
      </column-map>
    </relationship-role>
</relation-mapping>
```

Listing 5.18: Abbildung der Fahrzeug-Kunde-Beziehung auf die Datenbank

Zunächst wird die Abbildung der Entity-Beans *Fahrzeug* und *Kunde* auf die Datenbank beschrieben (*bean-mapping*). Anschließend wird die Abbildung der Beziehung beschrieben. Aus dem Deployment-Deskriptor weiß der EJB-Container, dass die Fahrzeug- und die Kunde-Bean an der Beziehung beteiligt sind. Ferner weiß er, dass es sich um eine eins-zu-eins-Beziehung handelt und dass die Beziehung unidirektional (von der Fahrzeug- zur Kunde-Bean) ist. Aus der Beschreibung der Beziehung (*relation-mapping*) weiß er nun auch, dass der Fremdschlüssel in der Tabelle der Fahrzeug-Bean gespeichert wird. Es wird der Teil der Beziehung mit dem Namen *fahrzeug-hat-kunde* beschrieben wird (dessen *relationship-role-source* die Fahrzeug-Bean ist) und nicht der Teil *kunde-benutzt-fahrzeug*. Der EJB-Container bildet demnach die Beziehung über die

Spalte *kundennr* der Tabelle *fahrzeug* (Fremdschlüssel) auf die Spalte *kundennr* der Tabelle *kunde* ab. Mit diesen Informationen sind die Tools des EJB-Containers bzw. des Persistenz-Managers in der Lage, den für die Verwaltung der Beziehung notwendigen Code zu generieren bzw. die Beziehung zur Laufzeit richtig zu handhaben.

Bidirektional

Damit die in Abbildung 5.9 dargestellte unidirektionale Beziehung zu einer bidirektionalen Beziehung erweitert werden kann, muss die Fahreug-Bean ebenfalls den Local-Client-View unterstützen. Andernfalls kann auf sie nicht von der Kunde-Bean referiert werden. Die Kunde-Bean muss wie die Fahrzeug-Bean abstrakte Beziehungsmethoden definieren (vgl. Listing 5.19).

```
...
    //Persistente Attribute
    public abstract void setKundennr(String kn);
    public abstract String getKundennr();
    public abstract void setName(String name);
    public abstract String getName();
    //Persistente Beziehungen
    public abstract void setFahrzeug(FahrzeugLocal fl);
    public abstract FahrzeugLocal getFahrzeug();
...
```

Listing 5.19: Abstrakte Beziehungsmethoden der Kunde-Bean

Wird eine Beziehung zwischen der Fahrzeug- und der Kunde-Bean durch die Methode *setKunde* auf Seiten der Fahrzeug-Bean hergestellt, so ist die Beziehung auch sofort von der Kunde-Bean aus sichtbar. Gleiches gilt umgekehrt. Wird eine Beziehung zwischen Fahrzeug- und Kunde-Bean durch den Aufruf der Methode *setFahrzeug* auf Seiten der Kunde-Bean hergestellt, so ist die Beziehung sofort auch von der Fahrzeug-Bean aus sichtbar. Im Fall einer bidirektionalen Beziehung genügt es also, die Beziehung auf einer der beiden Seiten herzustellen. Sie ist damit automatisch auch von der anderen Seite aus navigierbar. Besteht keine Beziehung zwischen Enterprise-Beans, die mit der Kardinalität eins-zu-eins verbunden sind, so liefert ein Aufruf der einschlägigen *getter*-Methoden den Rückgabewert *null*.

Die Bidirektionalität der Beziehung wird im Deployment-Deskriptor dadurch deklariert, dass nun für beide Seiten der Beziehung das Element *cmr-field* deklariert wird (siehe Listing 5.20). Die Deklaration im Deployment-Deskriptor ist letztlich ausschlaggebend dafür, dass die Beziehung in beide Richtungen navigierbar ist.

```
...
  <enterprise-beans>
    <entity>
      <ejb-name>Fahrzeug</ejb-name>
      <local-home>FahrzeugLocalHome</local-home>
```

```
      <local>FahrzeugLocal</local>

      ...
    </entity>
    <entity>
      <ejb-name>Kunde</ejb-name>
      <local-home>KundeLocalHome</local-home>
      <local>KundeLocal</local>
      ...
    </entity>
    ...
  </enterprise-beans>
...
  <ejb-relation>
    <ejb-relation-name>Fahrzeug-Kunde</ejb-relation-name>
    <ejb-relationship-role>
      <ejb-relationship-role-name>
        fahrzeug-hat-kunde
      </ejb-relationship-role-name>
      <multiplicity>One</multiplicity>
      <relationship-role-source>
        <ejb-name>Fahrzeug</ejb-name>
      </relationship-role-source>
      <cmr-field>
        <cmr-field-name>kunde</cmr-field-name>
      </cmr-field>
    </ejb-relationship-role>
    <ejb-relationship-role>
      <ejb-relationship-role-name>
        kunde-benutzt-fahrzeug
      </ejb-relationship-role-name>
      <multiplicity>One</multiplicity>
      <relationship-role-source>
        <ejb-name>Kunde</ejb-name>
      </relationship-role-source>
      <cmr-field>
        <cmr-field-name>fahrzeug</cmr-field-name>
      </cmr-field>
    </ejb-relationship-role>
  </ejb-relation>
...
```

Listing 5.20: Formale Beschreibung der Beziehung Fahrzeug-Kunde (bidirektional)

Die Beschreibung zur Abbildung auf die Datenbank muss nicht geändert werden. Der EJB-Container weiss bereits, wie die Beziehung gespeichert wird. Die in Abbildung 5.9 dargestellte Fremdschlüsselbeziehung ermöglicht die Navigation in beide Richtungen. Ob uni- oder bidirektional macht für die Programmierung wenig Unterschied. Für den EJB-Container ist die Navigation in die eine oder andere Richtung der Beziehung lediglich eine Spiegelung der Datenbankabfrage.

5.4.2 Eins-zu-N-Beziehungen (One to Many)

Unidirektional

Ein Beispiel für eine eins-zu-N-Beziehung ist die Beziehung zwischen Abteilung und Mitarbeiter, die in Abbildung 5.10 dargestellt wird. In einer Abteilung arbeiten mehrere Mitarbeiter. Ein Mitarbeiter ist immer genau einer Abteilung zugeordnet (Könnte ein Mitarbeiter mehreren Abteilungen zugeordnet sein, wäre es keine eins-zu-N-Beziehung mehr). Auch in diesem Fall wird die Beziehung mittels einer Fremdschlüsselbeziehung zwischen den beteiligten Tabellen gespeichert. Die Tabelle *mitarbeiter* enthält die Spalte *kostenstellennr* (=Fremdschlüssel), die die Beziehung zur jeweiligen Abteilung in der Tabelle *abteilung* herstellt. Alternativ könnte man auch eine Relationstabelle verwenden. Die Verwendung einer Relationstabelle wird im nächsten Abschnitt veranschaulicht.

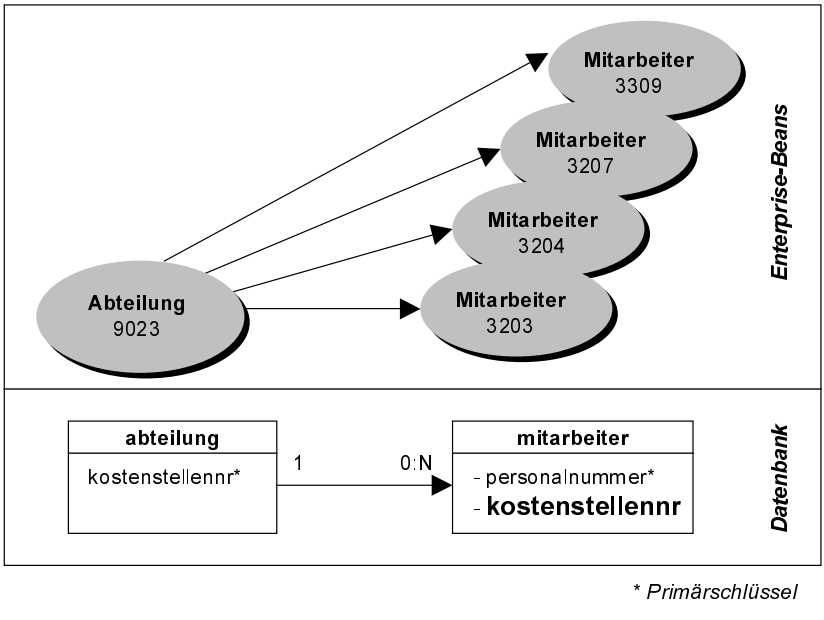

Abbildung 5.10: Beispiel unidirektionale eins-zu-N-Beziehung

Für den Fall der unidirektionalen Beziehung zwischen der Abteilung- und der Mitarbeiter-Bean definiert nur die Abteilung-Bean die abstrakten Beziehungsmethoden in der Bean-Klasse (siehe Listing 5.21). Da die Abteilung-Bean eine Beziehung zu mehreren Mitarbeiter-Beans haben kann, ist der Datentyp der Beziehung vom Type *java.util.Collection* oder *java.util.Set*.

```
...
    //Persistente Attribute
    public abstract void setKostenstellennr(String knr);
    public abstract String getKostenstellennr();
    //Persistente Beziehungen
    public abstract void setMitarbeiter(Collection ma);
    public abstract Collection getMitarbeiter();
...
```

Listing 5.21: Abstrakte Beziehungmethoden der Abteilung-Bean

Der Unterschied zwischen *java.util.Collection* und *java.util.Set* ist, dass eine Collection grundsätzlich Duplikate enthalten kann, ein Set nicht. In unserem Beispiel spielt es keine Rolle, welche der Klassen verwendet wird. Ein Mitarbeiter kann nur einer Abteilung zugeordnet sein. Die Personalnummer des Mitarbeites ist eindeutig, da sie der Primärschlüssel der Tabelle ist. Daher stellt schon die Datenbank sicher, dass es keine Duplikate gibt.

Die Mitarbeiter-Bean muss den Local-Client-View unterstützen, da auf sie über Navigation von der Abteilung-Bean referiert werden kann. Die Abteilung-Bean muss den Local-Client-View nicht unterstützen, da die Beziehung unidirektional ist und von der Abteilung- auf die Mitarbeiter-Bean zeigt.

Die Beschreibung der Beziehung im Deployment-Deskriptor unterscheidet sich von eins-zu-eins-Beziehungen nur durch die Kardinalität (siehe Listing 5.22). Außerdem muss durch das Element *cmr-field-type* angegeben werden, ob *java.util.Collection* oder *java.util.Set* als Datencontainer benutzt wird.

```
...
  <enterprise-beans>
    <entity>
      <ejb-name>Abteilung</ejb-name>
      ...
    </entity>
    <entity>
      <ejb-name>Mitarbeiter</ejb-name>
      <local-home>MitarbeiterLocalHome</local-home>
      <local>MitarbeiterLocal</local>
      ...
    </entity>
    ...
  </enterprise-beans>
...
  <ejb-relation>
    <ejb-relation-name>Abteilung-Mitarbeiter</ejb-relation-name>
    <ejb-relationship-role>
      <ejb-relationship-role-name>
        abteilung-hat-mitarbeiter
      </ejb-relationship-role-name>
```

```
      <multiplicity>One</multiplicity>
      <relationship-role-source>
        <ejb-name>Abteilung</ejb-name>
      </relationship-role-source>
      <cmr-field>
        <cmr-field-name>mitarbeiter</cmr-field-name>
        <cmr-field-type>java.util.Collection</cmr-field-type>
      </cmr-field>
    </ejb-relationship-role>
    <ejb-relationship-role>
      <ejb-relationship-role-name>
        mitarbeiter-ist-in-abteilung
      </ejb-relationship-role-name>
      <multiplicity>Many</multiplicity>
      <relationship-role-source>
        <ejb-name>Mitarbeiter</ejb-name>
      </relationship-role-source>
    </ejb-relationship-role>
  </ejb-relation>
...
```

Listing 5.22: Formale Beschreibung der Beziehung Abteilung-Mitarbeiter (unidirektional)

Die Beschreibung zur Abbildung auf die Datenbank (siehe Listing 5.23) ist nahezu identisch mit der von eins-zu-eins-Beziehungen. Vorsicht ist bei der Beschreibung der Fremdschlüsselbeziehung geboten: Man muss die Fremdschlüsselbeziehung für die richtige Seite der Beziehung beschreiben. In diesem Fall handelt es sich um die Seite *mitarbeiter-ist-in-abteilung*, da die *relationship-role-source* die Mitarbeiter-Bean ist. Sie wird auf die Tabelle *mitarbeiter* abgebildet, welche den Fremdschlüssel enthält.

```
<bean-mapping>
  <ejb-name>Abteilung</ejb-name>
  <data-source-name>postgres</data-source-name>
  <table-name>abteilung</table-name>
  <field-map>
    <cmp-field>kostenstellennr</cmp-field>
    <dbms-column>kostenstellennr</dbms-column>
  </field-map>
</bean-mapping>
<bean-mapping>
  <ejb-name>Mitarbeiter</ejb-name>
  <data-source-name>postgres</data-source-name>
  <table-name>mitarbeiter</table-name>
  <field-map>
    <cmp-field>personalnr</cmp-field>
    <dbms-column>personalnr</dbms-column>
  </field-map>
</bean-mapping>

...
```

```
<relation-mapping>
  <relation-name>Abteilung-Mitarbeiter</relation-name>
    <relationship-role>
      <relationship-role-name>
        mitarbeiter-ist-in-abteilung
      </relationship-role-name>
      <column-map>
        <foreign-key-column>kostenstellennr</foreign-key-column>
        <key-column>kostenstellennr</key-column>
      </column-map>
    </relationship-role>
</relation-mapping>
```

Listing 5.23: Abbildung der Abteilung-Mitarbeiter-Beziehung auf die Datenbank

Die Besonderheit für den Bean-Entwickler bei der Verwendung einer eins-zu-N-Beziehung liegt im Umgang mit der Kardinalität n der Beziehung, die durch den Datentyp *java.util.Collection* oder *java.util.Set* repräsentiert wird.

Im Vergleich zur Kardinalität eins liefert die *get*-Methode der Kardinalität n niemals den Wert *null* zurück. Wenn keine Beziehungen zwischen den Bean-Instanzen existieren, ist der Rückgabewert ein *Collection*- oder *Set*-Objekt, welches keine Elemente beinhaltet (also leer ist).

Um Beziehungen zwischen der Abteilung- und der Mitarbeiter-Bean herzustellen, hat der Bean-Entwickler zwei Möglichkeiten. Die Variante, die Listing 5.24 zeigt, benutzt die Methode *setMitarbeiter* der Abteilung-Bean. Sie stellt eine Beziehung zu den Mitarbeitern A, B und C her. Hätte die Abteilung-Bean bereits Beziehungen zu dem Mitarbeitern D und E gehabt, würden diese Beziehungen gelöscht. Die Semantik einer *set*-Methode ist, bereits bestehende Werte zu überschreiben. Dies ist auch bei Beziehungen der Fall. Die Beziehung zu den Mitarbeitern D und E würde mit der Beziehung zu den Mitarbeitern A, B und C überschrieben. Die Beziehung zu den Mitarbeitern D und E würde nicht mehr bestehen.

```
...

MitarbeiterLocalHome mitarbeiterHome = ...
AbteilungHome abteilungHome = ...

Abteilung abteilung = abteilungHome.findByPrimaryKey(...);

MitarbeiterLocal mA = mitarbeiterHome.create("A", ...);
MitarbeiterLocal mB = mitarbeiterHome.create("B", ...);
MitarbeiterLocal mC = mitarbeiterHome.create("C", ...);

java.util.ArrayList al = new java.util.ArrayList();
al.add(mA);
al.add(mB);
```

```
al.add(mC);

abteilung.setMitarbeiter(al);

...
```

Listing 5.24: Setzen von Beziehungen über die Methode setMitarbeiter()

Listing 5.25 zeigt, wie Beziehungen zu Mitarbeitern hergestellt werden können, ohne die bestehenden Beziehungen zu überschreiben.

```
...

MitarbeiterLocalHome mitarbeiterHome = ...
AbteilungHome abteilungHome = ...

Abteilung abteilung = abteilungHome.findByPrimaryKey(...);

java.util.Collection col = abteilung.getMitarbeiter();

MitarbeiterLocal mA = mitarbeiterHome.create("A", ...);
MitarbeiterLocal mB = mitarbeiterHome.create("B", ...);
MitarbeiterLocal mC = mitarbeiterHome.create("C", ...);

col.add(mA);
col.add(mB);
col.add(mC);

...
```

Listing 5.25: Hinzufügen von Beziehungen über die Methode Collection.add()

Das Objekt, das die Methode *getMitarbeiter* der Abteilung-Bean zurückliefert, ist eine Implementierung des Interfaces *java.util.Collection*, die der EJB-Container bereitstellt. Durch das *Collection*-Objekt gewährt der EJB-Container dem Bean-Entwickler Zugriff auf die Kardinalität n der Beziehung mit der Mitarbeiter-Bean. Methodenaufrufe an diesem *Collection*-Objekt werden vom EJB-Container bearbeitet. Änderungen werden vom EJB-Container sofort an die Datenbank weitergegeben.

Der Bean-Entwickler muss beachten, dass eine Mitarbeiter-Bean mit nur einer Abteilung-Bean verbunden sein kann. Ist Mitarbeiter H in Abteilung B, würde ein Hinzufügen des Mitarbeiters H zur Abteilung C bewirken, dass die Verbindung mit der Abteilung B gelöscht wird. Dieses Verhalten ist durch die Kardinalität eins-zu-N bestimmt. Kann ein Mitarbeiter mit zwei oder mehr Abteilungen verbunden sein, handelt es sich um die Kardinalität N-zu-M.

Listing 5.26 zeigt, wie der Bean-Entwickler über eine Beziehung der Kardinalität n zu einer bestimmten Bean navigieren kann.

```
...

java.util.Collection col = abteilung.getMitarbeiter();

java.util.Iterator it = col.iterator();

while(it.hasNext()) {
    MitarbeiterLocal ma = (MitarbeiterLocal)it.next();
    if(ma.getName().equals(nameToSearchFor)) {
       ...
    }
    ...
}

...
```

Listing 5.26: Navigation über die Kardinalität n der Abteilung-Bean

Die Objekte, die die Collection beinhaltet, sind immer vom Typ des Local-Interfaces der verbundenen Bean. Deswegen ist auch kein Type-Narrowing über die Methode *narrow* der Klasse *javax.rmi.PortableRemoteObject* notwendig.

Listing 5.27 zeigt, wie der Bean-Entwickler eine Beziehung der Kardinalität n zu einer Bean löschen kann.

```
...

java.util.Collection col = abteilung.getMitarbeiter();
java.util.Iterator it = col.iterator();
while(it.hasNext()) {
    MitarbeiterLocal ma = (MitarbeiterLocal)it.next();
    if(ma.isFired()) {
        it.remove();
    }
}
...
```

Listing 5.27: Löschen einer Beziehung der Kardinalität n der Abteilung-Bean

Das Löschen einer Beziehung bedeutet natürlich nicht das Löschen der verbundenen Bean-Instanz. Gelöscht wird lediglich die Beziehung der beiden Entity-Beans. Der EJB-Container setzt dazu den Wert der Fremdschlüsselspalte auf *null*.

Bidirektional

Um eine unidirektionale eins-zu-N-Beziehung in eine bidirektionale eins-zu-N-Beziehung zu verwandeln, sind die gleichen Schritte notwendig, wie bei den eins-zu-eins-Beziehungen. Die Abteilung-Bean muss ebenfalls den Local-Client-View unterstützen, da auf sie sonst von der Mitarbeiter-Bean nicht referiert werden kann. Außerdem muss

die Mitarbieter-Bean abstrakte Beziehungsmethoden definieren (siehe Listing 5.28). Da die Kardinalität auf Seiten der Abteilung-Bean mit eins deklariert ist, sind die Beziehungsmethoden vom Typ *AbteilungLocal*.

```
...
    //Persistente Attribute
    public abstract void setPersonalnummer(String pn);
    public abstract String getPersonalnummer();
    //Persistente Beziehungen
    public abstract void setAbteilung(AbteilungLocal al);
    public abstract AbteilungLocal getAbteilung();
...
```

Listing 5.28: Abstrakte Beziehungsmethoden der Mitarbeiter-Bean

Die abstrakten Beziehungsmethoden der Mitarbeiter-Bean müssen noch im Deployment-Deskriptor deklariert werden (siehe Listing 5.29). Erst durch die Deklaration der Beziehungsmethoden im Deployment-Deskriptor ist die Beziehung bidirektional.

```
...
  <enterprise-beans>
    <entity>
      <ejb-name>Abteilung</ejb-name>
      <local-home>AbteilungLocalHome</local-home>
      <local>AbteilungLocal</local>
      ...
    </entity>
    <entity>
      <ejb-name>Mitarbeiter</ejb-name>
      <local-home>MitarbeiterLocalHome</local-home>
      <local>MitarbeiterLocal</local>
      ...
    </entity>
    ...
  </enterprise-beans>
...
  <ejb-relation>
    <ejb-relation-name>Abteilung-Mitarbeiter</ejb-relation-name>
    <ejb-relationship-role>
      <ejb-relationship-role-name>
        abteilung-hat-mitarbeiter
      </ejb-relationship-role-name>
      <multiplicity>One</multiplicity>
      <relationship-role-source>
        <ejb-name>Abteilung</ejb-name>
      </relationship-role-source>
      <cmr-field>
        <cmr-field-name>mitarbeiter</cmr-field-name>
        <cmr-field-type>java.util.Collection</cmr-field-type>
      </cmr-field>
```

```
      </ejb-relationship-role>
      <ejb-relationship-role>
        <ejb-relationship-role-name>
          mitarbeiter-ist-in-abteilung
        </ejb-relationship-role-name>
        <multiplicity>Many</multiplicity>
        <relationship-role-source>
          <ejb-name>Mitarbeiter</ejb-name>
        </relationship-role-source>
        <cmr-field>
          <cmr-field-name>abteilung</cmr-field-name>
        </cmr-field>
      </ejb-relationship-role>
    </ejb-relation>
...
```

Listing 5.29: Formale Beschreibung der Beziehung Abteilung-Mitarbeiter (bidirektional)

Die Beschreibung der Abbildung auf die Datenbank ist (wie bei den eins-zu-eins-Beziehungen) identisch mit der der unidirektionalen eins-zu-N-Beziehung. Die Beziehung auf Datenbankebene ist bereits in beide Richtungen navigierbar. Die Navigation in die andere Richtung ist lediglich eine spiegelbildliche Formulierung der Datenbankabfrage.

5.4.3 N-zu-M-Beziehungen (Many to Many)

Unidirektional

Ein Beispiel für eine N-zu-M-Beziehung ist die Beziehung zwischen einem Produkt und einem Zukaufteil. Abbildung 5.11 verdeutlicht diesen Sachverhalt. Ein Zukaufteil kann in mehreren Produkten verwendet werden. Für die Speicherung der Beziehung kann in diesem Fall aber keine Fremdschlüsselbeziehung mehr verwendet werden. Voraussetzung für die Verwendung einer Fremdschlüsselbeziehung ist, dass wenigstens in einer Richtung der Beziehung Eindeutigkeit in der Zuordnung besteht. Bei N-zu-M-Beziehungen ist das nicht mehr der Fall. Deshalb muss für die Speicherung der Beziehung eine sogenannte Relationstabelle benutzt werden. Sie existiert nur für den Zweck der Speicherung von Beziehungen. Sie enthält alle Felder des Primärschlüssels der beteiligten Entitäten.

Für den Fall der unidirektionalen N-zu-M-Beziehung definiert nur die Produkt-Bean die abstrakten Beziehungsmethoden in der Bean-Klasse (siehe Listing 5.30). Da die Produkt-Bean Beziehungen zu mehreren Zukaufteil-Beans haben kann, sind die abstrakten Beziehungsmethoden vom Typ *java.util.Collection* bzw. *java.util.Set*.

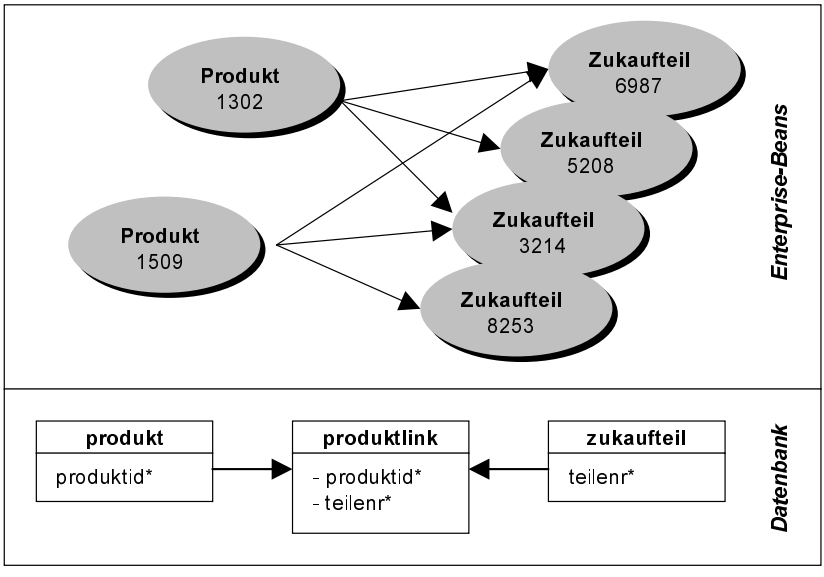

Abbildung 5.11: Beispiel unidirektionale N-zu-M-Beziehung

```
...
    //Persistente Attribute
    public abstract void setProduktId(String pid);
    public abstract String getProduktId();
    //Persistente Beziehungen
    public abstract void setZukaufteil(Collection zt);
    public abstract Collection getZukaufteil();
...
```

Listing 5.30: Abstrakte Beziehungsmethoden der Produkt-Bean

Die Unterschiede zwischen *java.util.Collection* und *java.util.Set* sowie deren Verwendung zum Setzen, Hinzufügen und Löschen von Beziehungen wurden bereits im vorhergehenden Abschnitt behandelt. Es existiert jedoch ein nennenswerter Unterschied zu den eins-zu-N-Beziehungen, der durch die Semantik von N-zu-M-Beziehungen begründet wird. Hat zum Beispiel das Produkt A eine Beziehung zum Zukaufteil X und wird Zukaufteil X nun auch in Beziehung zu Produkt D gesetzt, hat dies keine Auswirkungen auf die Beziehung zu Produkt A. Beide Produkte A und D haben nun eine Beziehung zum Zukaufteil X. Bei einer eins-zu-N-Beziehung wäre die Beziehung zu Produkt A durch das Setzen der Beziehung zu Produkt D gelöscht worden.

Listing 5.31 zeigt die Beschreibung der unidirektionalen N-zu-M-Beziehung zwischen der Produkt- und der Zukaufteil-Bean im Deployment-Deskriptor. Die Zukaufteil-Bean muss den Local-Client-View unterstützen, da auf sie durch Navigation von der

Produkt-Bean aus referiert wird. Die Produkt-Bean muss im Falle der unidirektionalen Beziehung den Local-Client-View nicht unterstützen.

```
...
  <enterprise-beans>
    <entity>
      <ejb-name>Produkt</ejb-name>
      ...
    </entity>
    <entity>
      <ejb-name>Zukaufteil</ejb-name>
      <local-home>ZukaufteilLocalHome</local-home>
      <local>ZukaufteilLocal</local>
      ...
    </entity>
    ...
  </enterprise-beans>
  ...
  <ejb-relation>
    <ejb-relation-name>Produkt-Zukaufteil</ejb-relation-name>
    <ejb-relationship-role>
      <ejb-relationship-role-name>
        produkt-hat-zukaufteil
      </ejb-relationship-role-name>
      <multiplicity>Many</multiplicity>
      <relationship-role-source>
        <ejb-name>Produkt</ejb-name>
      </relationship-role-source>
      <cmr-field>
        <cmr-field-name>zukaufteil</cmr-field-name>
        <cmr-field-type>java.util.Collection</cmr-field-type>
      </cmr-field>
    </ejb-relationship-role>
    <ejb-relationship-role>
      <ejb-relationship-role-name>
        zukaufteil-ist-in-produkt
      </ejb-relationship-role-name>
      <multiplicity>Many</multiplicity>
      <relationship-role-source>
        <ejb-name>Zukaufteil</ejb-name>
      </relationship-role-source>
    </ejb-relationship-role>
  </ejb-relation>
...
```

Listing 5.31: Formale Beschreibung der Beziehung Produkt-Zukaufteil (unidirektional)

Listing 5.32 zeigt ein fiktives Beispiel, wie die Abbildung der Beziehung auf die Datenbank beschrieben werden könnte. Neu im Vergleich zu den bisherigen Beispielen ist die Notwendigkeit, eine Relationstabelle verwenden zu müssen.

```
<bean-mapping>
  <ejb-name>Produkt</ejb-name>
  <data-source-name>postgres</data-source-name>
  <table-name>produkt</table-name>
  <field-map>
    <cmp-field>produktId</cmp-field>
    <dbms-column>produktid</dbms-column>
  </field-map>
</bean-mapping>
<bean-mapping>
  <ejb-name>Zukaufteil</ejb-name>
  <data-source-name>postgres</data-source-name>
  <table-name>zukaufteil</table-name>
  <field-map>
    <cmp-field>teileNummer</cmp-field>
    <dbms-column>teilenr</dbms-column>
  </field-map>
</bean-mapping>

...
<relation-mapping>
  <relation-name>Produkt-Zukaufteil</relation-name>
  <relation-table>produktlink</relation-table>
    <relationship-role>
      <relationship-role-name>
        produkt-hat-zukaufteil
      </relationship-role-name>
      <column-map>
        <foreign-key-column>produktid</foreign-key-column>
        <key-column>produktid</key-column>
      </column-map>
    </relationship-role>
    <relationship-role>
      <relationship-role-name>
        zukaufteil-ist-in-produkt
      </relationship-role-name>
      <column-map>
        <foreign-key-column>teilenr</foreign-key-column>
        <key-column>teilenr</key-column>
      </column-map>
    </relationship-role>
</relation-mapping>
```

Listing 5.32: Abbildung der Produkt-Zukaufteil-Beziehung auf die Datenbank

Zunächst wird die Abbildung der persistenten Attribute der Entity-Beans auf die jeweiligen Tabellen beschrieben. Bei der Beschreibung der Beziehung wird zunächst angegeben, dass eine Relations-Tabelle verwendet wird und wie sie heißt. Anschließend muss die Abbildung beider Seiten der Beziehung auf die Relationstabelle beschrieben werden. Die Beschreibung von nur einer Seite der Beziehung, wie bei der

Verwendung eines Fremdschlüssels, ist nicht ausreichend, da beide Seiten der Beziehung eine Fremdschlüsselbeziehung zur Relationstabelle haben.

Bidirektional

Wie bei den eins-zu-eins- und eins-zu-N-Beziehungen, so ist auch bei den unidirektionalen N-zu-M-Beziehungen die Erweiterung in eine bidirektionale N-zu-M-Beziehung relativ einfach. Auch die Produkt-Bean muss nun den Local-Client-View unterstützen, damit auf sie von der Zukaufteil-Bean referiert werden kann. Die Zukaufteil-Bean muss abstrakte Beziehungsmethoden deklarieren (siehe Listing 5.33), um die Navigation in die andere Richtung (von der Zukaufteil-Bean auf die Produkt-Bean) zu ermöglichen. Da eine Zukaufteil-Bean mehreren Produkten zugeordnet sein kann, ist die Beziehung vom Typ *java.util.Collection* (alternativ *java.util.Set*).

```
...
    //Persistente Attribute
    public abstract void setTeileNummer(String tn);
    public abstract String getTeileNummer();
    //Persistente Beziehungen
    public abstract void setProdukt(Collection p);
    public abstract Collection getProdukt();
...
```

Listing 5.33: Abstrakte Beziehungsmethoden der Zukaufteil-Bean

Der nächste Baustein für die Erweiterung in eine bidirektionale N-zu-M-Beziehung ist die Deklaration der abstrakten Beziehungsmethoden der Zukaufteil-Bean im Deployment-Deskriptor (siehe Listing 5.34).

```
...
  <enterprise-beans>
    <entity>
      <ejb-name>Produkt</ejb-name>
      <local-home>ProduktLocalHome</local-home>
      <local>ProduktLocal</local>
      ...
    </entity>
    <entity>
      <ejb-name>Zukaufteil</ejb-name>
      <local-home>ZukaufteilLocalHome</local-home>
      <local>ZukaufteilLocal</local>
      ...
    </entity>
    ...
  </enterprise-beans>
  ...
  <ejb-relation>
    <ejb-relation-name>Produkt-Zukaufteil</ejb-relation-name>
```

```
      <ejb-relationship-role>
        <ejb-relationship-role-name>
          produkt-hat-zukaufteil
        </ejb-relationship-role-name>
        <multiplicity>Many</multiplicity>
        <relationship-role-source>
          <ejb-name>Produkt</ejb-name>
        </relationship-role-source>
        <cmr-field>
          <cmr-field-name>zukaufteil</cmr-field-name>
          <cmr-field-type>java.util.Collection</cmr-field-type>
        </cmr-field>
      </ejb-relationship-role>
      <ejb-relationship-role>
        <ejb-relationship-role-name>
          zukaufteil-ist-in-produkt
        </ejb-relationship-role-name>
        <multiplicity>Many</multiplicity>
        <relationship-role-source>
          <ejb-name>Zukaufteil</ejb-name>
        </relationship-role-source>
        <cmr-field>
          <cmr-field-name>produkt</cmr-field-name>
          <cmr-field-type>java.util.Collection</cmr-field-type>
        </cmr-field>
      </ejb-relationship-role>
    </ejb-relation>
...
```

Listing 5.34: Formale Beschreibung der Beziehung Produkt-Zukaufteil (bidirektional)

Die Beschreibung der Abbildung auf die Datenbank ist (wie bei den anderen Beziehungsarten) identisch mit der der unidirektionalen N-zu-M-Beziehung. Die Verwendung der Relationstabelle ermöglicht von Haus aus die Navigation in beide Richtungen.

5.4.4 Cascade-Delete

Wird eine Beziehung zwischen Entity-Beans gelöscht, bleiben die Entity-Beans selbst davon unberührt. Nur die Beziehung zwischen den Entity-Beans wird gelöscht, nicht die Entity-Beans selbst. In manchen Fällen ist die Beziehung zwischen zwei Entity-Beans von der Natur, dass die Existenz der einen ohne die Existenz der anderen wenig Sinn macht. Ein Beispiel dafür wäre die bidirektionale eins-zu-eins-Beziehung zwischen der Entity-Bean *Benutzer* und der Entity-Bean *Benutzerprofil*. Die Entity-Bean *Benutzer* speichert Daten wie den Benutzernamen, das verschlüsselte Passwort oder die Rechte die der Benutzer hat. Diese Daten sind Voraussetzung dafür, dass der Benutzer im System arbeiten kann. Die Entity-Bean *Benutzerprofil* speichert Daten wie den vollen Namen des Benutzers, die Abteilung in der er arbeitet oder seine Telefon-

nummer. Diese Daten sind ein Zusatz, die man z.B. für Auswertungen heranziehen kann. Wird die Benutzer-Bean eines bestimmten Benutzers gelöscht (z.B. beim Ausscheiden des Mitarbeiters aus der Firma), so wird vom EJB-Container auch die Beziehung zur Benutzerprofil-Bean gelöscht. Die Benutzerprofil-Bean dieses Benutzers ist damit eine Leiche im System, sofern sie nicht explizit mitgelöscht wird. Auf das Benutzerprofil kann ohne Benutzer nicht mehr referiert werden.

Für solche Fälle kann man den EJB-Container anweisen, die mit der zu löschenden Entity-Bean in Beziehung stehenden Entity-Beans ebenfalls zu löschen. Diesen Mechanismus nennt man *kaskadierendes Löschen* (*cascading delete*). Listing 5.35 zeigt, wie man den EJB-Container anweisen kann, ein kaskadierendes Löschen durchzuführen.

```
...
  <enterprise-beans>
    <entity>
      <ejb-name>Benutzer</ejb-name>
      ...
    </entity>
    <entity>
      <ejb-name>Benutzerprofil</ejb-name>
      ...
    </entity>
    ...
  </enterprise-beans>
  ...
  <ejb-relation>
    <ejb-relation-name>Benutzer-Benutzerprofil</ejb-relation-name>
    <ejb-relationship-role>
      <ejb-relationship-role-name>
        benutzer-hat-benutzerprofil
      </ejb-relationship-role-name>
      <multiplicity>One</multiplicity>
      <relationship-role-source>
        <ejb-name>Benutzer</ejb-name>
      </relationship-role-source>
      <cmr-field>
        <cmr-field-name>profil</cmr-field-name>
      </cmr-field>
    </ejb-relationship-role>
    <ejb-relationship-role>
      <ejb-relationship-role-name>
        benutzerprofil-gehoert-zu-benutzer
      </ejb-relationship-role-name>
      <multiplicity>One</multiplicity>
      <cascade-delete/>
      <relationship-role-source>
        <ejb-name>Benutzerprofil</ejb-name>
      </relationship-role-source>
      <cmr-field>
```

```
          <cmr-field-name>benutzer</cmr-field-name>
        </cmr-field>
      </ejb-relationship-role>
    </ejb-relation>
...
```

Listing 5.35: Deklaration kaskadierendes Löschen

Wird eine bestimmte Bentzer-Bean gelöscht, so wird durch die Anweisung das Element *cascade-delete* in Listing 5.35 die verbundene Benutzerprofil-Bean vom EJB-Container automatisch mitgelöscht. Kaskadierendes Löschen kann nur von einer Entity-Bean ausgehen, deren Kardinalität eins ist. Andernfalls kann nicht sichergestellt werden, dass auf die gelöschte Entity-Bean von keiner anderen Beziehung mehr referiert wird. Die Konsistenz der gespeicherten Daten könnte dadurch gefährdet werden.

Cascade-delete ist ein mächtiges Werkzeug, das mit Bedacht eingesetzt werden sollte. Die Benutzerprofil-Bean könnte zum Beispiel noch eine Beziehung zu einer anderen Entity-Bean haben, in der auch ein *cascade-delete* deklariert ist etc. D.h. durch kaskadierendes Löschen können beim Entfernen einer Entity-Bean aus der Datenbank richtige Kettenreaktionen hervorgerufen werden. Bei komplexeren Anwendungen mit vielen Beziehungen kann es schwer werden, den Überblick über die Abhängigkeiten zu behalten um *cascade-delete* richtig einzusetzen.

Abschließend zum Abschnitt über Beziehungen zwischen Entity-Beans möchten wir noch auf die Ausführungen zu Beziehungen zwischen Entity-Beans in Kapitel 10.3.6 und 10.3.7 der EJB-Spezifikation (vgl. [Sun Microsystems, 2001]) hinweisen.

5.5 EJB-QL (EJB 2.0)

Mit der Version 2.0 wurde die Abfragesprache EJB-QL eingeführt. EJB-QL steht dabei für *EJB Query Language for Container-Managed-Persistence Query Methods*. Wie der Name bereits sagt, handelt es sich um eine EJB-spezifische Abfragesprache für die Formulierung von Suchen über Entity-Beans mit *Container-Managed-Persistence*. Die Syntax von EJB-QL ist sehr stark an die Syntax der Abfragesprache SQL 92 angelehnt. Die Abschnitte über Finder- und Select-Methoden haben bereits einen kleinen Vorgeschmack auf die EJB-QL gegeben.

Suchabfragen, die mit EJB-QL formuliert sind, können unverändert in verschiedenen EJB-Containern und unterschiedlichsten Persistenzsystemen eingesetzt werden. Für die Formulierung von Finder-Methoden bei Container-Managed-Persistence haben viele Applikationsserver ihre eigene, proprietäre Sprache entwickelt, da die Spezifikation vor der Version 2.0 keine unabhängige Abfragesprache definiert hat. Bei der Portierung einer Entity-Bean mit Container-Managed-Persistence mussten dann die Suchabfragen für die Finder-Methoden auf den jeweiligen Applikationsserver bzw. EJB-Container angepasst werden.

EJB-QL wird nicht zur Laufzeit interpretiert, sondern zum Zeitpunkt des Deployments vom EJB-Container in eine andere Abfragesprache übersetzt. Wurde z. B. einer Entity-Bean mit Container-Managed-Persistence beim Deployment eine relationale Datenbank als Persistenzsystem zugeordnet, würde der EJB-Container die Suchabfrage von EJB-QL in SQL übersetzen. Wenn die Suchabfrage aufgerufen wird, führt der EJB-Container die entsprechenden SQL-Befehle aus.

EJB-QL Suchanfragen beziehen sich immer auf einen bestimmten Abfrageraum. Die Grenzen eines solchen Abfrageraums werden durch den Deployment-Deskriptor festgelegt. Bei der Formulierung einer Suchabfrage, die ja immer im Umfeld eines bestimmten Entity-Bean-Typs definiert wird, kann auf alle Entity-Beans mit Container-Managed-Persistence zugegriffen werden, die im gleichen Deployment-Deskriptor definiert sind.

EJB-QL wird konkret eingesetzt für die Definition von:

- *Finder-Methoden*
  Finder-Methoden werden im Home- bzw. im Local-Home-Interface einer Entity-Bean mit Container-Managed-Persistence definiert. Implementiert werden sie durch die Angabe einer EJB-QL-Abfrage im Deployment-Deskriptor. Ist die Finder-Methode im Home-Interface einer Entity-Bean definiert, so muss der Typ des Rückgabewertes der zugehörigen EJB-QL-Abfrage vom Typ des Remote-Interfaces der Entity-Bean sein. Wurde die Finder-Methode im Local-Home-Interface der Entity-Bean defniert, so muss der Typ des Rückgabewertes der zugehörigen Suchabfrage vom Typ des Local-Interfaces der Entity-Bean sein.
- *Select-Methoden*
  Sie werden als abstrakte Methoden in der Bean-Klasse deklariert und dienen zum Zugriff auf den persistenten Zustand anderer Entity-Beans. Implementiert werden sie (wie die Finder-Methoden) durch die Angabe einer EJB-QL-Abfrage im Deployment-Deskriptor. Der Typ des Rückgabewertes einer solchen Abfrage kann vom Typ des Remote-, des Local-Interfaces oder eines persistenten Attrbuts einer Entity-Bean sein.

Listing 5.36 zeigt einen Ausschnitt aus einem Deployment-Deskriptor, in dem die Entity-Bean *Person* und die Entity-Bean *Adresse* definiert werden. Beide Entity-Beans stehen zueinander in einer bidirektionalen eins-zu-eins-Beziehung. Die Person-Bean definiert die persistenten Attribute *name* und *vorname*, die zusammen mit der persistenten Beziehung zur Adresse-Bean zu dem abstrakten Persistenzschema der Person-Bean gehören, welchem über das Element *abstract-schema-name* der Name *PersonAPS* gegeben wird. Das Postfix APS steht für *abstract persistence schema* und wird verwendet, damit es für den Leser nicht zu Verwechslungen zwischen dem Bean-Namen und dem Namen des abstrakten Persistenzschemas kommt (üblicherweise verwendet man für beide Elemente den gleichen Namen). Die Adresse-Bean definiert die persistenten

Attribute *name* (=Fremdschlüssel für die Beziehung zu Person-Bean), *strasse*, *plz* und *stadt*. Sie gehören zusammen mit der persistenten Beziehung zur Person-Bean zum abstrakten Persistenzschema der Adresse-Bean, welches mit *AdresseAPS* benannt wird. Wir werden uns im weiteren Verlauf dieses Abschnitts bei den EJB-QL-Beispielen immer wieder auf diesen Deployment-Deskriptor beziehen.

```
...
<entity>
    <ejb-name>PersonBean</ejb-name>
    <local-home>PersonLocalHome</local-home>
    <local>PersonLocal</local>
    <ejb-class>PersonBean</ejb-class>
    <persistence-type>Container</persistence-type>
    <prim-key-class>java.lang.String</prim-key-class>
    <reentrant>False</reentrant>
    <cmp-version>2.x</cmp-version>
    <abstract-schema-name>PersonAPS</abstract-schema-name>
    <cmp-field>
        <field-name>name</field-name>
    </cmp-field>
    <cmp-field>
        <field-name>vorname</field-name>
    </cmp-field>
    <primkey-field>name</primkey-field>
    </entity>
<entity>
    <ejb-name>AdresseBean</ejb-name>
    <local-home>AdresseLocalHome</local-home>
    <local>AdresseLocal</local>
    <ejb-class>AdresseBean</ejb-class>
    <persistence-type>Container</persistence-type>
    <prim-key-class>java.lang.String</prim-key-class>
    <reentrant>False</reentrant>
    <cmp-version>2.x</cmp-version>
    <abstract-schema-name>AdresseAPS</abstract-schema-name>
    <cmp-field>
        <field-name>name</field-name>
    </cmp-field>
    <cmp-field>
        <field-name>strasse</field-name>
    </cmp-field>
    <cmp-field>
        <field-name>plz</field-name>
    </cmp-field>
    <cmp-field>
        <field-name>stadt</field-name>
    </cmp-field>
    <primkey-field>name</primkey-field>
</entity>
...
```

```
<ejb-relation>
    <ejb-relation-name>Person-Adresse</ejb-relation-name>
    <ejb-relationship-role>
        <ejb-relationship-role-name>
            person-hat-adresse
        </ejb-relationship-role-name>
        <multiplicity>One</multiplicity>
        <relationship-role-source>
            <ejb-name>PersonBean</ejb-name>
        </relationship-role-source>
        <cmr-field>
            <cmr-field-name>adresse</cmr-field-name>
        </cmr-field>
    </ejb-relationship-role>
    <ejb-relationship-role>
        <ejb-relationship-role-name>
            adresse-gehoert-zu-person
        </ejb-relationship-role-name>
        <multiplicity>One</multiplicity>
        <relationship-role-source>
            <ejb-name>AdresseBean</ejb-name>
        </relationship-role-source>
        <cmr-field>
            <cmr-field-name>person</cmr-field-name>
        </cmr-field>
    </ejb-relationship-role>
</ejb-relation>
...
```

Listing 5.36: EJB-QL Beispiel für die Entity-Bean Person

5.5.1 Aufbau der Suchabfrage

Eine Suchabfrage besteht aus drei Teilen:

1. SELECT-Klausel
   Sie bestimmt den Typ des Rückgabewertes der Suchabfrage. Dabei kann es sich um eine Referenz auf eine Entity-Bean oder um ein persistentes Attribut einer Entity-Bean handeln.

2. FROM-Klausel
   Sie bestimmt, auf welchen Bereich sich die Angaben in der SELECT- und der optionalen WHERE-Klausel beziehen. Abfragen beziehen sich auf im Deployment-Deskriptor definierte CMP 2.0 Entity-Beans bzw. auf deren persistente Attribute und Beziehungen.

3. WHERE-Klausel (optional)
   Sie dient zur Einschränkung der Ergebnismenge.

Das einfachste Beispiel einer EJB-QL-Abfrage zeigt Listing 5.37. Es wird angenommen, dass die Person-Bean in ihrem Home-Interface eine Finder-Methode mit dem Namen *findAllePersonen* definiert. Die Abfrage, die über das Element *ejb-ql* definiert wird, wird der Methode *findAllePersonen* über das Element *query-method* zugeordnet.

```
...
<entity>
    <ejb-name>PersonBean</ejb-name>
    ...
    <query>
        <query-method>
            <method-name>findAllePersonen</method-name>
        </query-method>
        <ejb-ql>
            SELECT OBJECT(p) FROM PersonAPS AS p
        </ejb-ql>
    </query>
</entity>
...
```

Listing 5.37: Einfache EJB-QL-Abfrage für findAllePersonen

Die Abfrage liefert alle Objekte, die dem abstrakten Persistenzschema PersonAPS entsprechen, also alle existierenden Person-Entity-Beans. Der Client, der die *findAllePersonen*-Methode aufruft, bekommt ein Objekt vom Typ *java.util.Collection*, das Local-Referenzen auf die gefundenen Person-Beans enthält.

Folgende Punkte gelten generell für Suchabfragen in EJB-QL:

- Abfragen beziehen sich immer auf das abstrakte Persistenzschema einer oder mehrerer Entity-Beans mit Container-Managed-Persistence 2.0.
- Zum abstrakten Persistenzschema einer Entity-Bean gehören die persistenten Attribute und die persistenten Beziehungen zu anderen Entity-Beans.
- EJB-QL-Abfragen werden im Deployment-Deskriptor definiert, und zwar im Bereich einer bestimmten Entity-Bean.
- EJB-QL-Abfragen werden immer einer bestimmten Finder- oder Select-Methode zugeordnet.
- In die Suchabfrage dürfen nur solche Entity-Beans einbezogen werden, die im gleichen Deployment-Deskriptor definiert sind wie die Entity-Bean, in deren Bereich die Suchabfrage definiert wird.

5.5.2 Attributsuche

Die Ergebnismenge einer Abfrage kann über die Attribute einer Entity-Bean (die Bestandteil des abstrakten Persistenzschemas sind) eingeschänkt werden. Nehmen wir an, die Person-Bean definiert in ihrem Local-Home-Interface die Finder-Methode

```
java.util.Collection findByVorname(java.lang.String vorname)
     throws javax.ejb.FinderException;
```

um nur die Personen zu finden, die einen bestimmten Vornamen haben. Listing 5.38 zeigt die entsprechende Definition im Deployment-Deskriptor.

```
...
<entity>
    <ejb-name>PersonBean</ejb-name>
    ...
    <abstract-schema-name>PersonAPS</abstract-schema-name>
    ...
    <cmp-field>
        <field-name>vorname</field-name>
    </cmp-field>
    ...
    <query>
        <query-method>
            <method-name>findByVorname</method-name>
            <method-params>
                <method-param>java.lang.String</method-param>
            </method-params>
        </query-method>
        <ejb-ql>
            SELECT OBJECT(p) FROM PersonAPS AS p WHERE p.vorname=?1
        </ejb-ql>
    </query>
</entity>
...
```

Listing 5.38: EJB-QL-Abfrage für findByVorname

Durch die WHERE-Klausel wird die Ergebnismenge auf alle Objekte eingeschränkt, deren Vorname dem Wert des ersten Übergabeparameters (*?1*) der Methode *findByVorname* entspricht (in dem Fall dem Parameter *vorname*). Über das Element *method-params* müssen die Typen der Übergabeparameter festgelegt werden.

Neben der Einschränkung des Suchergebnises durch Attribute können die Werte persistenter Attribute auch zurückgegeben werden. Dies ist allerdings den Select-Methoden vorbehalten, da Finder-Methoden nur Referenzen auf Entity-Beans zurückgeben dürfen. Finder-Methoden werden vom Client aufgerufen, Select-Methoden nicht.

Angenommen, die Person-Bean definiert eine Select-Methode

```
public abstract Collection ejbSelectNamenProStadt(String stadt)
    throws FinderException;
```

um herauszufinden, welche Personen in einer bestimmten Stadt wohnen. Die entsprechende Deklaration im Deployment-Deskriptor zeigt Listing 5.39.

```
...
<entity>
    <ejb-name>PersonBean</ejb-name>
    ...
    <query>
        <query-method>
            <method-name>ejbSelectNamenProStadt</method-name>
            <method-params>
                <method-param>java.lang.String</method-param>
            </method-params>
        </query-method>
        <ejb-ql>
            SELECT p.name FROM AdresseAPS AS p WHERE p.stadt=?1
        </ejb-ql>
    </query>
</entity>
<entity>
    <ejb-name>AdresseBean</ejb-name>
    ...
    <abstract-schema-name>AdresseAPS</abstract-schema-name>
    <cmp-field>
        <field-name>name</field-name>
    </cmp-field>
    ...
    <cmp-field>
        <field-name>stadt</field-name>
    </cmp-field>
    ...
</entity>
...
```

Listing 5.39: EJB-QL Abfrage für ejbSelectNameProStadt

Der Typ des Rückgabewerts ist nun keine Bean-Objekt mehr, sondern der Typ des Attributes *name* des abstrakten Persistenzschemas *AdresseAPS*. Da die Abfrage mehr als ein Ergebnis liefern kann, definiert die Select-Methode den Rückgabewert vom Typ *Collection*.

5.5.3 Suche über Beziehungen

Neben den Attributen gehören auch die persistenten Beziehungen zum abstrakten Persistenzschema einer Entity-Bean. Sie können wie die Attribute in Suchabfragen verwendet werden. Zum einen können die Attribute bezogener Entity-Beans ebenfalls zur Einschränkung der Ergebnismengen dienen, zum anderen können die bezogenen Entity-Bean-Objekte bzw. die Werte ihrer persistenten Attribute von der Suchanfrage zurückgegeben werden.

Nehmen wir an, die Person-Bean definiert folgende Select-Methoden:

```
public abstract AdresseLocal ejbSelectAdresse()
    throws FinderException;

public abstract String ejbSelectStadt()
    throws FinderException;

public abstract Collection ejbSelectPersonMitPlz(Integer plz)
    throws FinderException;
```

Die Methode *ejbSelectAdresse* soll die Adresse liefern, die über die persistente eins-zu-eins-Beziehung mit der jeweiligen Person-Bean verbunden ist. Die Methode *ejbSelectStadt* soll nur das persistente Attribut *stadt* der Adresse-Bean für alle existierenden Personen liefern. Die Methode *ejbSelectPersonMitPlz* liefert alle Personen, die in einer bestimmten Stadt wohnen. Listing 5.40 zeigt die zu den Methoden gehörenden Abfragen.

```
...
<entity>
    <ejb-name>PersonBean</ejb-name>
    ...
    <abstract-schema-name>PersonAPS</abstract-schema-name>
    ...
    <query>
        <query-method>
            <method-name>ejbSelectAdresse</method-name>
        </query-method>
        <ejb-ql>
            SELECT p.adresse FROM PersonAPS AS p
        </ejb-ql>
    </query>
    <query>
        <query-method>
            <method-name>ejbSelectStadt</method-name>
        </query-method>
        <ejb-ql>
            SELECT p.adresse.stadt FROM PersonAPS AS p
        </ejb-ql>
    </query>
    <query>
        <query-method>
            <method-name>ejbSelectPersonMitPlz</method-name>
            <method-params>
                <method-param>java.lang.Integer</method-param>
            </method-params>
        </query-method>
        <ejb-ql>
            SELECT OBJECT(p) FROM PersonAPS AS p
            WHERE p.adresse.plz = ?1
```

```
        </ejb-ql>
    </query>
</entity>
<entity>
    <ejb-name>AdresseBean</ejb-name>
    ...
    <abstract-schema-name>AdresseAPS</abstract-schema-name>
    ...
    <cmp-field>
        <field-name>plz</field-name>
    </cmp-field>
    <cmp-field>
        <field-name>stadt</field-name>
    </cmp-field>
    ...
</entity>
...
<ejb-relation>
    <ejb-relation-name>Person-Adresse</ejb-relation-name>
    <ejb-relationship-role>
        ...
        <multiplicity>One</multiplicity>
        <relationship-role-source>
            <ejb-name>PersonBean</ejb-name>
        </relationship-role-source>
        <cmr-field>
            <cmr-field-name>adresse</cmr-field-name>
        </cmr-field>
    </ejb-relationship-role>
    <ejb-relationship-role>
        ...
    </ejb-relationship-role>
</ejb-relation>
...
```

Listing 5.40: EJB-QL und persistente Beziehungen

Die Beziehung zur Adresse-Bean wird im Fall der Kardinalität eins behandelt wie ein persistentes Attribut. Zur Referenz der bezogenen Entity-Bean wird der Punkt-Operator verwendet (*SELECT p.adresse*). Die Navigation kann mittels Punkt-Operator bis auf die persistenten Attribute der bezogenen Entity-Bean ausgedehnt werden (*SELECT p.adresse.stadt*).

Wäre die Beziehung zur Adresse-Bean nicht von der Kardinalität eins sondern von der Kardinalität n (d.h. eine Person kann z. B. mehrere Wohnsitze haben), wäre die Definition der Select-Methoden sowie der in Listing 5.40 gezeigten Abfragen nicht mehr gültig. Die Signaturen der Select-Methoden müssten wie folgt geändert werden:

```
public abstract Collection ejbSelectAdresse()
    throws FinderException;

public abstract Collection ejbSelectStadt()
    throws FinderException;

public abstract Collection ejbSelectPersonMitPlz(Integer plz)
    throws FinderException;
```

Da die Beziehung von der Kardinalität eins-zu-N ist, kann die Person-Bean mit mehreren Adresse-Beans in Beziehung stehen. Daher muss der Rückgabewert der Methoden *ejbSelectAdresse* und *ejbSelectStadt* vom Typ *java.util.Collection* (alternativ wäre *java.util.Set* möglich) sein. Soll in einer Abfrage über eine Beziehung mit der Kardinalität n navigiert werden, muss statt des Punkt-Operators der IN-Operator verwendet werden. Listing 5.41 zeigt die Anwendung des IN-Operators.

```
...
<entity>
    <ejb-name>PersonBean</ejb-name>
    ...
    <abstract-schema-name>PersonAPS</abstract-schema-name>
    ...
    <query>
        <query-method>
            <method-name>ejbSelectAdresse</method-name>
        </query-method>
        <ejb-ql>
          SELECT OBJECT(a) FROM PersonAPS AS p, IN(p.adresse) AS a
        </ejb-ql>
    </query>
    <query>
        <query-method>
            <method-name>ejbSelectStadt</method-name>
        </query-method>
        <ejb-ql>
          SELECT a.stadt FROM PersonAPS AS p, IN(p.adresse) AS a
        </ejb-ql>
    </query>
    <query>
        <query-method>
            <method-name>ejbSelectPersonMitPlz</method-name>
            <method-params>
                <method-param>java.lang.Integer</method-param>
            </method-params>
        </query-method>
        <ejb-ql>
          SELECT OBJECT(p) FROM PersonAPS AS p,
                           IN(p.adresse) AS a
            WHERE a.plz = ?1
        </ejb-ql>
```

```
    </query>
</entity>
<entity>
    <ejb-name>AdresseBean</ejb-name>
    ...
</entity>
...
<ejb-relation>
    <ejb-relation-name>Person-Adresse</ejb-relation-name>
    <ejb-relationship-role>
        ...
        <multiplicity>One</multiplicity>
        <relationship-role-source>
            <ejb-name>PersonBean</ejb-name>
        </relationship-role-source>
        <cmr-field>
            <cmr-field-name>adresse</cmr-field-name>
        </cmr-field>
    </ejb-relationship-role>
    <ejb-relationship-role>
        ...
        <multiplicity>Many</multiplicity>
        <relationship-role-source>
            <ejb-name>AdresseBean</ejb-name>
        </relationship-role-source>
    </ejb-relationship-role>
</ejb-relation>
```

Listing 5.41: Anwendung des IN-Operators

Die Navigation darf im Fall einer Beziehung mit der Kardinalität n nicht mehr über den Punkt-Operator erfolgen. Die Objekte, die über die Beziehung mit der Entity-Bean verbunden sind, müssen zuerst über den IN-Operator an eine Variable gebunden werden. Über diese Variable kann dann im weiteren Verlauf der Abfrage auf die persistenten Attribute der verbundenen Entity-Beans referiert werden. Die Variable wird auch verwendet, wenn die verbundenen Objekte selbst zurückgegeben werden.

5.5.4 Weitere Operatoren und Ausdrücke

Tabelle 5.1 zeigt die Datentypen, die für Konstanten in EJB-QL-Anweisungen verwendet werden können.

Datentyp	Syntax für Konstanten
Strings	Strings werden in einfache Hochkommata (') eingeschlossen.
Ganze Zahlen	Ziffern

Tabelle 5.1: Datentypen für Konstanten in EJB-QL-Abfragen

Datentyp	Syntax für Konstanten
Fließkommazahlen	Ziffern mit einem Dezimalpunkt
Bool'sche Werte	TRUE oder FALSE

Tabelle 5.1: Datentypen für Konstanten in EJB-QL-Abfragen (Fortsetzung)

Tabelle 5.2 führt alle Operatoren auf. Die Reihenfolge entspricht der aufsteigenden Bindungsstärke (*Präzedenz*) der Operatoren.

Operator	Beschreibung
NOT	logisches nicht
AND	logisches oder
OR	logisches oder
=	Gleich
>	Größer
>=	größer oder gleich
<	Kleiner
<=	kleiner gleich
<>	Ungleich
+ (unär)	Inkrementieren eines Zahl um eins
- (unär)	Dekrementieren einer Zahl um eins
*	Multiplikationszeichen
+ (binär)	Additionszeichen
- (binär)	Substraktionszeichen
.	Navigationsoperator für Attribute und Referenzen innerhalb des Deployment-Deskriptors

Tabelle 5.2: EJB-QL-Operatoren

Das Größer- und das Kleiner-Zeichen, das in EJB-QL-Abfragen verwendet werden kann, gehören zur XML-Syntax. Es sind Zeichen, die innerhalb von XML-Elementen nicht verwendet werden dürfen. Damit es bei der Verwendung solcher Zeichen nicht zu Schwierigkeiten mit dem XML-Parser kommt, muss die EJB-QL-Abfrage in eine so genannte *CDATA*-Sektion eingebunden werden:

```
<query>
    <query-method>
        <method-name>...</method-name>
    </query-method>
    <ejb-ql>
      <![CDATA[
        SELECT l.summe FROM Rechnung AS r WHERE r.summe > 1000
```

```
        ]]>
      </ejb-ql>
    </query>
```

Der XML-Parser interpretiert keine Daten, die sich innerhalb einer CDATA-Section befinden. Er gibt sie uninterpretiert an die darüber liegende Anwendungsschicht weiter.

Tabelle 5.3 zeigt Ausdrücke, die im WHERE-Teil verwendet werden können.

Ausdruck	Beschreibung
BETWEEN	prüft, ob eine Zahl zwischen zwei gegebenen Werten liegt.
	Syntax: <Zahlenwert> [NOT] BETWEEN <Zahlenwert> AND <Zahlenwert>
	Beispiel: … WHERE p.adresse.plz BETWEEN 80000 AND 9000
IN	prüft, ob ein String in einer Menge von Strings vorkommt (nicht zu verwechseln mit dem IN-Operator im FROM-Teil einer Abfrage!).
	Syntax: <String-Wert> [NOT] IN (<String-Wert>, <String-Wert>, ...)
	Beispiel: ... WHERE p.adresse.stadt IN ('Muenchen', 'Hamburg', 'Dresden')
LIKE	vergleicht einen String mit einem einfachen regulären Ausdruck, um ähnliche Strings zu erkennen. Die Syntax des regulären Ausdrucks berücksichtigt zwei Sonderzeichen: Der Unterstrich (_) steht für ein beliebiges Zeichen und das Prozent-Zeichen (%) für eine Sequenz von 0 – n beliebigen Zeichen. Alle anderen Zeichen stehen für sich selbst. Um ein Prozent oder einen Unterstrich als Zeichen zu verwenden, wird ihnen ein Backslash (\) vorangestellt.
	Syntax: <String-Wert> [NOT] LIKE <regulärer Ausdruck>
	Beispiel: ... WHERE p.name LIKE ('%M__er%')
IS NULL	prüft, ob ein Attribut, eine eins-zu-eins- oder N-zu-eins-Beziehung für eine Instanz gesetzt ist.
	Syntax: <cmp/cmr-Feld> IS [NOT] NULL
	Beispiel: ... WHERE p.adresse IS NOT NULL AND …
IS EMPTY	prüft, ob eine eins-zu-N- oder N-zu-M-Beziehung für eine Instanz gesetzt ist. Wenn die Beziehung auf keine Instanz verweist, ist das Ergebnis TRUE.
	Syntax: <cmr-Feld> IS [NOT] EMPTY
	Beispiel: ... WHERE p.adresse IS NOT EMPTY (eins-zu-N- Fall)
MEMBER OF	prüft, ob ein Objekt Bestandteil einer Menge von Objekten ist.
	Syntax: <cmp/cmr-Feld/Eingabeparameter> [NOT] MEMBER [OF]
	Beispiel: … WHERE p.adresse MEMBER OF …

Tabelle 5.3: Ausdrücke der WHERE-Klausel

Tabelle 5.4 zeigt Funktionen, die in der Abfrage verwendet werden können.

CONCAT	fügt zwei Strings zusammen.
	Syntax: CONCAT(<String1>, <String2>)
	Beispiel: CONCAT ('abc', 'defg') ergibt den String 'abcdefg'
SUBSTRING	bestimmt einen Teilstring.
	Syntax: SUBSTRING(<String>, <Startposition>, <Länge>)
	Beispiel: SUBSTRING ('abcdefg', 2, 3) ergibt den String 'cde'
LOCATE	sucht einen Substring in einem anderen String.
	Syntax: LOCATE(<String>, <Substring>, [<start>])
	Beispiel: LOCATE('abcdefg', 'cde') ergibt den Wert 2 (integer)
LENGTH	bestimmt die Länge eines Strings.
	Syntax: LENGTH(<String>)
	Beispiel: LENGTH('abcdefg') ergibt den Wert 7 (integer)
ABS	berechnet den absoluten Wert für die Datentypen *int*, *float* und *double*
	Syntax: ABS (<int>) oder ABS (<float>) oder ABS (<double>)
	Beispiel: ABS(- 11.72) ergibt 11.72
SQRT	bestimmt die Quadratwurzel.
	Syntax: SQRT(<double>)
	Beispiel: SQRT(16) ergibt den Wert 4 (double)

Tabelle 5.4: Eingebaute EJB-QL-Funktionen

Abschließend bleibt anzumerken, dass EJB-QL einen großen Beitrag zur Portabilität von Entity-Beans mit Container-Managed-Persistence leistet. Leider bietet EJB-QL noch nicht den Funktionsumfang, den man von SQL gewohnt ist. So fehlt z.B. der ORDER BY Operator, um die Suchergebnisse sortieren zu lassen. Außerdem wird der Datentyp *java.util.Date* nicht unterstützt. Man kann sich kaum eine Anwendung vorstellen, bei der man nicht in irgendeiner Form mit einem persistenten Datum zu tun hat. Die nächsten Versionen der EJB-Spezifikation werden an diesen Stellen sicher Abhilfe schaffen.

5.6 Beispiel: Lagerverwaltung (EJB 2.0)

Ziel dieses Abschnitts ist es, anhand eines Beipiels zu veranschaulichen, wie die in den vorangegangenen Abschnitten diskutierten Technologien (persistente Attribute, persistente Beziehungen in Verbindung mit dem Local-Client-View, EJB-QL) zu einer

Gesamtlösung kombiniert werden können. Im Vergleich zu den anderen Beispielen in diesem Buch, ist dieses Beipiel sehr komplex. Wir wollen aus Gründen der Übersichtlichkeit und des Umfangs darauf verzichten, den kompletten Source-Code dieses Beispiels zu zeigen. Der komplette Code für dieses Beispiel kann von der auf dem Umschlag angegebenen Quelle bezogen werden. Wir wollen uns auf die Teile beschränken, die das unterstreichen, was wir mit diesem Beispiel zeigen wollen.

5.6.1 Problemstellung

Die Buchhaltungsabteilung benötigt zu Zwecken des Controllings Zugriff auf die Bestandsdaten der Materialläger. Die Lagerverwaltung ist teilautomatisiert. Das Lagerverwaltungssystem hat keine Client-Anwendung, die den Anforderungen der Buchhaltung gerecht wird. Deswegen soll für die Buchhaltungsabteilung eine Client-Anwendung entwickelt werden, die lesenden Zugriff auf die Datenbank des Lagerverwaltungssystems bietet.

Abbildung 5.12 zeigt den Kern des Datenmodells des Lagerverwaltungssystems. Die Tabelle *STORES* beinhaltet alle Läger. Das sind z.B. ein Lager für Zukaufteile, ein Lager für Halbfertigprodukte, ein Lager für Endprodukte, ein Lager für Garantiefälle etc. In jedem Lager stehen Regale. Diese werden in der Tabelle *RACKS* gespeichert. Welche Regale in welchem Lager stehen, wird über einen Fremdschlüssel (*STOREID*) in der Tabelle *RACKS* gespeichert.

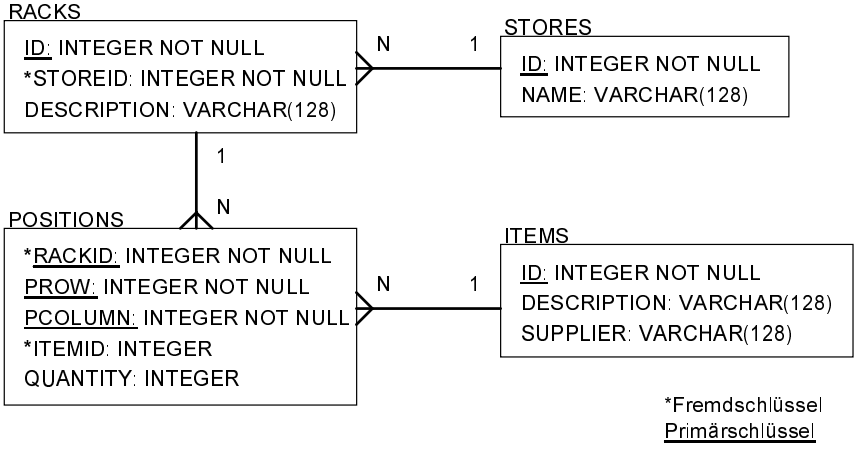

Abbildung 5.12: Datenbankschema Lagerverwaltung

Jedes Regal hat mehrere Positionen (Fächer), in denen Teile eingelagert werden können. Das Lagerverwaltungsystem setzt voraus, dass an einer Position in einem Regal immer nur ein Artikel (in beliebiger Menge) abgelegt werden kann. Eine bestimmte

Position in einem Regal wird durch die Ebene (*PROW*) und die Reihe (*PCOLUMN*) des Regals bestimmt. Welche Positionen welches Regal hat, wird über einen Fremdschlüssel (*RACKID*) in der Tabelle *POSITIONS* gespeichert. In den Positionen eines Regals befinden sich Artikel (*ITEMS*). Welche Artikel es gibt, wird in der Tabelle *ITEMS* gespeichert. Welche Artikel in welchen Positionen lagern, wird über einen Fremdschlüssel (*ITEMID*) in der Tabelle *POSITIONS* gespeichert.

Das Client-Programm wurde auf der Basis einer Anforderungsanalyse, die zusammen mit den Mitarbeitern aus der Buchhaltungsabteilung durchgeführt wurde, entwickelt. Abbildung 5.13 zeigt das Client-Programm (*ejb.store.client.StoreManager*). Im oberen Teil der Anwendung befindet sich eine Auswahlbox mit der Beschriftung *Lager.* Dort kann das Lager ausgewählt werden, dessen Bestände eingesehen werden sollen. Standardmäßig wird das erste Lager aus der Liste aller vorhandenen Läger angezeigt. In der linken Spalte werden alle Regale anzeigt, die sich in dem oben ausgewählten Lager befinden. Im Lager *Produkte* stehen z.B. die Regale mit den Namen *Bildschirme*, *Gehäuse* und *Diverse*.

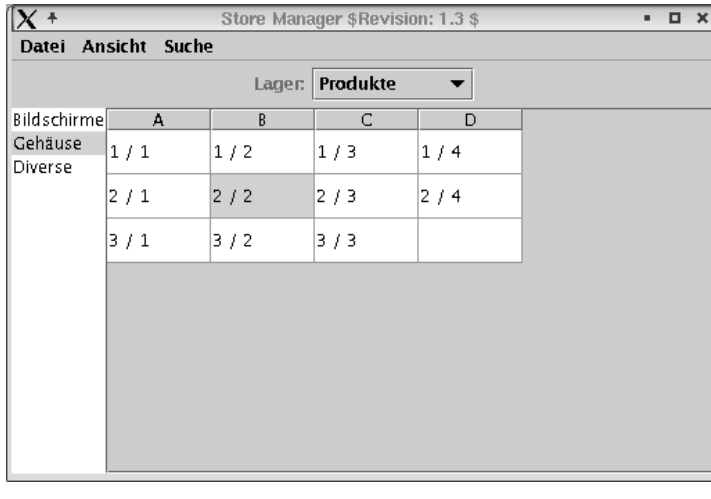

Abbildung 5.13: StoreManager

Wird ein Regal ausgewählt, erscheint in der Mitte der Anwendung eine Tabelle, die die Regalpositionen des jeweiligen Regals anzeigt. Ist eine Position belegt, wird die Ebene und die Reihe der jeweiligen Position angezeigt. Ist diese Position nicht belegt, wird ein leeres Feld angezeigt. In Abbildung 5.13 wurde die Position *2/2* ausgewählt (Regalfach in Ebene 2, Reihe 2). Um zu sehen, welcher Artikel sich in Position *2/2* des Regals *Gehäuse* befindet, kann über den Menüpunkt ANSICHT - ARTIKEL ein Dialog angezeigt werden, der den in dieser Position befindlichen Artikel anzeigt. Aus Abbildung 5.14 ist ersichtlich, dass in der Position *2/2* des Regals *Gehäuse* der Artikel *Gehäuse Big Tower* des Lieferanten *Plangate Inc.* lagert.

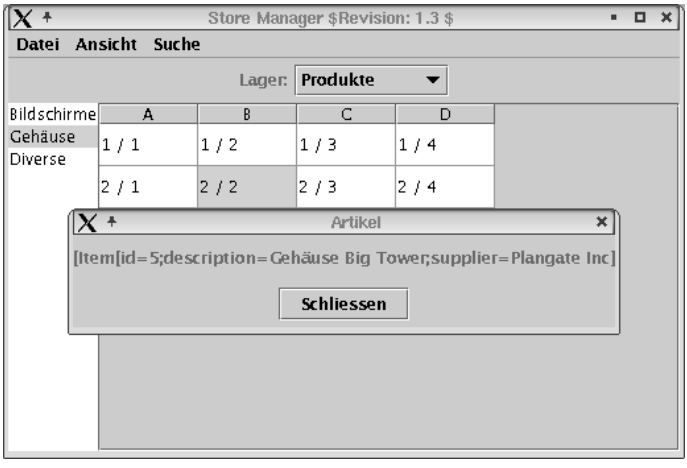

Abbildung 5.14: StoreManager – Artikelanzeige

Statt die einzelnen Positionen eines Regals durchzusehen, kann nach den Lagerpositionen eines bestimmten Artikels gesucht werden. Dazu wird der Menüpunkt SUCHE - ARTIKEL benutzt, der den in Abbildung 5.15 gezeigten Dialog aufblendet. Im oberen Teil des Dialogs werden alle im System vorhandenen Artikel aufgelistet. Um zu sehen, wo der Artikel *TFT Bildschirm 15 inch* gelagert ist, wird er in der Liste selektiert und der Button SUCHEN gedrückt. Daraufhin erscheint im unteren Teil des Dialogs eine Liste mit Regalpositionen, an denen der Artikel lagert. Im Beispiel in Abbildung 5.15 ist der Artikel im Regal *Bildschirme* an den Positionen *1/1*, *1/2* und *1/3* zu finden.

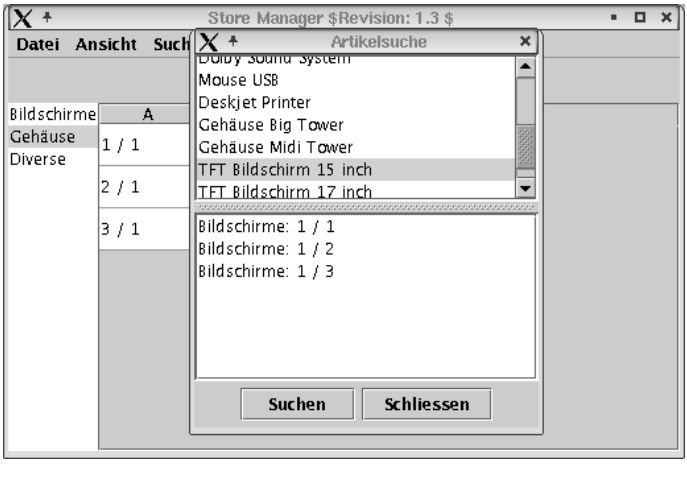

Abbildung 5.15: StoreManager – Artikelsuche

Nachdem feststeht, wie das Clientprogramm und die Benutzerführung aussehen muss, kann nun mit der Problemlösung begonnen werden.

5.6.2 Problemlösung

Jede der vier Datenbanktabellen wird über eine Entity-Bean repräsentiert. Die Tabellenspalten werden auf persistente Attribute der jeweiligen Entity-Bean abgebildet. Die Fremdschlüsselbeziehungen der Tabellen werden über persistente Beziehungen auf die Entity-Beans übertragen. Abbildung 5.16 zeigt die Entity-Beans und deren Beziehung zueinander.

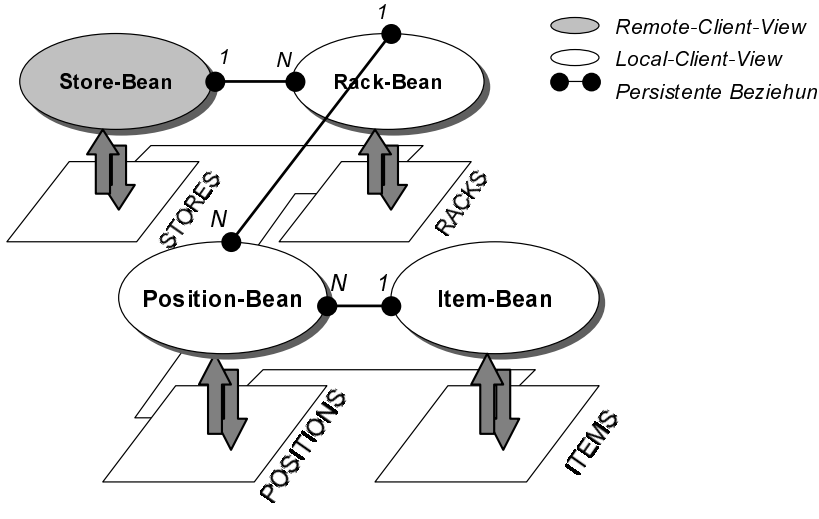

Abbildung 5.16: Modellierung des Datenmodells mit Entity-Beans

Die Store-Bean ist die einzige Bean, die den Remote-Client-View unterstützt. Alle anderen Beans unterstützen nur den Local-Client-View. Wegen der Beziehungen müssen sie den Local-Client-View unterstützen, könnten jedoch zusätzlich auch ein Remote-Interface anbieten. Ziel dieses Designansatzes ist, dem Client nur eine Bean, die Store-Bean, zugänglich zu machen. Das hat für den Client den Vorteil, dass er die internen Strukturen und die Abhängigkeiten auf der Serverseite nicht kennen muss. Für ihn gibt es nur eine Stelle, mit der er kommunizieren muss, nämlich mit der Store-Bean (»mit dem Lager«). Sie alleine versorgt ihn mit den Informationen, die er braucht. Damit ist die Benutzung der Serverschnittstelle für den Client auch wesentlich einfacher. Die internen Zusammenhänge bleiben ihm verbogen. Dadurch entstehen auch zwischen dem Client und den Internas des Servers keine Abhängigkeiten. Solange die Store-Bean ihre Schnittstelle beibehält bleibt der Client von Änderungen auf der Serverseite völlig unberührt.

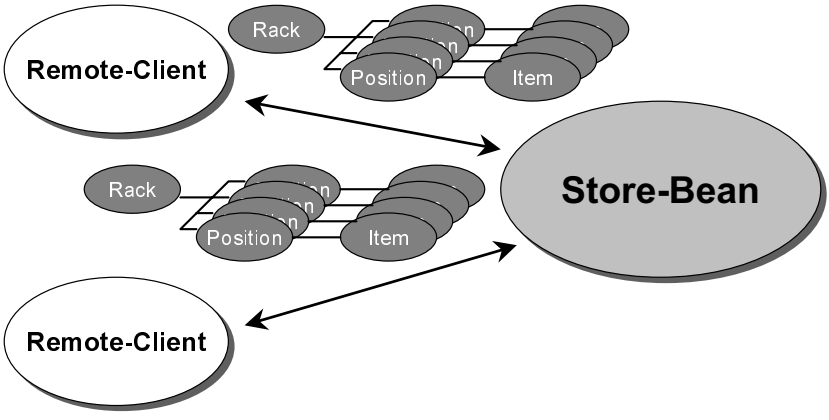

Abbildung 5.17: Kommunikation mit dem Remote-Client

Regale (*Rack*), Positionen (*Position*) und Artikel (*Item*) bekommt der Client nur in Form von einfachen Java-Klassen zu Gesicht (siehe Abbildung 5.17). Diese Klassen sind Containerklassen für die Daten der Rack-, Position- und Item-Bean. Die Store-Bean versorgt den Client über diese Klassen mit den jeweiligen Informationen. Sie füllt die Instanzen dieser Containerklassen mit den Daten aus den entsprechenden Entity-Beans. Der Client weiß nichts von der Existenz der Rack-, Position- und Item-Bean. Die Containerklassen sind damit auch Bestandteil der öffentlichen Schnittstelle der Store-Bean. Listing 5.42 zeigt die Implementierung der Klasse *Rack*. Die Klassen *Position* und *Item* sind nach exakt dem gleichen Muster entwickelt und werden deshalb nicht gezeigt.

```
package ejb.store;

public class Rack implements java.io.Serializable {

    private Integer id;
    private String  description;
    private Integer storeId;

    public Rack(Integer id, String desc, Integer storeId) {
        if(id == null || id.intValue() < 0)
            throw new IllegalArgumentException("id");
        if(desc == null)
            throw new IllegalArgumentException("desc == null!");
        if(storeId == null || storeId.intValue() < 0)
            throw new IllegalArgumentException("storeId");

        this.id          = id;
        this.description = desc;
        this.storeId     = storeId;
    }
```

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

    public String getDescription() {
        return this.description;
    }

    public Integer getStoreId() {
        return this.storeId;
    }

    public String toString() {
        return "[Rack[id=" + this.id +
               ";description=" + this.description + "]";
    }

}
```

Listing 5.42: Containerklasse Rack

Listing 5.45 am Ende dieses Abschnitts zeigt den Deployment-Deskriptor. Er ist der maßgebliche Teil der Problemlösung. In ihm fließen alle Informationen zusammen. Listing 5.43 zeigt das Home-Interface, Listing 5.44 das Remote-Interface der Store-Bean. Die Klassen *Rack*, *Position* und *Item* sowie die beiden Interfaces *Store* und *StoreHome* bilden zusammen die Serverschnittstelle für den Client.

```
package ejb.store;

import javax.ejb.CreateException;
import javax.ejb.EJBHome;
import javax.ejb.FinderException;

import java.rmi.RemoteException;
import java.util.Collection;

public interface StoreHome extends EJBHome {

    public Store create(Integer id, String name)
        throws CreateException, RemoteException;

    public Store findByPrimaryKey(Integer key)
        throws FinderException, RemoteException;

    public Collection findAllStores()
        throws FinderException, RemoteException;

}
```

Listing 5.43: Home-Interface der Store-Bean

Die Methode *findAllStores* des Home-Interfaces wird benutzt, um die Auswahlbox mit der Beschriftung *Lager* mit Werten zu füllen (siehe Abbildung 5.13). Wählt der Benutzer ein bestimmtes Lager aus, verschafft sich die Client-Anwendung mit *findByPrimaryKey* Zugriff auf das jeweilige Lager.

Um die Liste zu füllen, die die Regale anzeigt, verwendet die Client-Anwendung die Methode *getStoreRacks* des Remote-Interfaces. Die Store-Bean steht mit der Rack-Bean in einer eins-zu-N-Beziehung (siehe Listing 5.45, Beziehung *Store-Rack*). Damit ist es für die Store-Bean ein Leichtes herauszufinden, welche Regale sich in diesem Lager befinden. Sie muss lediglich über die Beziehung navigieren, um Zugriff auf die entsprechenden Rack-Beans zu bekommen. Die Daten der Rack-Beans werden in *Rack*-Objekte verpackt und an die Client-Anwendung als Ergebnis des Methodenaufrufs zurückgegeben.

```
package ejb.store;

import javax.ejb.EJBObject;

import java.rmi.RemoteException;
import java.util.Collection;

public interface Store extends EJBObject {

    public Integer getStoreId()
        throws RemoteException;

    public String getStoreName()
        throws RemoteException;

    public Collection getStoreRacks()
        throws RemoteException;

    public Collection getStoreRackPositions(Rack rack)
        throws RemoteException;

    public Item getItemInPosition(Position pos)
        throws RemoteException;

    public Collection getAllStoreItems()
        throws RemoteException;

    public Collection getPositionsForItem(Item item)
        throws RemoteException;

}
```

Listing 5.44: Remote-Interface der Store-Bean

Wählt der Benutzer ein bestimmtes Regal aus, müssen alle Regalpositionen in der Tabelle angezeigt werden. Dazu ruft die Client-Anwendung die Methode *getStoreRackPositions* auf und übergibt als Parameter ein *Rack*-Objekt, für das die Positionen gewünscht werden. Die Store-Bean verschafft sich zunächst durch Navigation über die Beziehung Zugriff auf die entsprechende Rack-Bean. Die Rack-Bean ihrerseits steht mit der Position-Bean in einer eins-zu-N-Beziehung (siehe Listing 5.45, Beziehung *RackPosition*). Die Rack-Bean wird von der Store-Bean angewiesen, alle zu ihr gehörigen Position-Beans durch Navigation über die Beziehung zu bestimmen und in *Position*-Objekte zu vewandeln. Diese *Position*-Objekte werden von der Store-Bean an den Client als Ergebnis des Methodenaufrufs zurückgegeben.

Wählt der Benutzer eine bestimmte Position aus, um den darin enthaltenen Artikel anzuzeigen (siehe Abbildung 5.14), ruft die Client-Anwendung die Methode *getItemInPosition* auf. Sie übergibt das *Position*-Objekt, für das der Artikel angezeigt werden soll. Die Store-Bean veschafft sich durch einen *findByPrimaryKey*-Aufruf am Local-Home-Interface der Position-Bean Zugriff auf die entsprechende Position-Bean. Die Position-Bean steht mit der Item-Bean in einer N-zu-eins-Beziehung (siehe Listing 5.45, Beziehung *Position-Item*). Die Position-Bean wird angewiesen, die zu ihr gehörige Item-Bean durch Navigation über die Beziehung zu bestimmen und in ein *Item*-Objekt zu verwandeln. Dieses *Item*-Objekt wird von der Store-Bean an den Client als Ergebnis des Methodenaufrufs zurückgegeben und kann von der Client-Anwendung angezeigt werden. Alternativ hätte die Store-Bean die entsprechende Item-Bean auch über eine Select-Methode, die im Deployment-Deskriptor mit einer EJB-QL Abfrage verknüpft wird, finden können.

Um dem Benutzer die Suche nach den Lagerpositionen eines besimmten Artikels zu ermöglichen (siehe Abbildung 5.15), muss der Suchdialog zunächst mit den im System vorhandenen Artikeln gefüllt werden. Dazu ruft die Client-Anwendung die Methode *getAllStoreItems* auf. Die Store-Bean benutzt dazu eine Select-Methode, die im Deployment-Deskriptor mit einer EJB-QL-Abfrage verknüpft wurde (siehe Listing 5.45, Methode *ejbSelectAllItems*). Die Select-Methode liefert alle im System befindlichen Item-Beans zurück. Die Store-Bean verwandelt die lokalen Entity-Bean Referenzen in *Item*-Objekte und gibt sie als Ergebnis des Methodenaufrufs an das Client-Programm zurück. Das Client-Programm benutzt diese *Item*-Objekte, um die Artikelliste im Suchdialog zu füllen.

Wählt der Benutzer einen bestimmte Artikel aus und drückt den SUCHEN-Button, ruft das Client-Programm die Methode *getPositionsForItem* auf. Die Store-Bean benutzt auch hier eine Select-Methode, die im Deployment-Deskriptor mit einer EJB-QL-Abfrage verknüpft wurde (siehe Listing 5.45, Methode *ejbSelectPositionsForItem*). Die Select-Methode liefert die Position-Beans zurück, in denen der jeweilige Artikel lagert. Die Store-Bean verwandelt die lokalen Referenzen in *Position*-Objekte und gibt sie an

das Client-Programm als Ergebnis des Methodenaufrufs zurück. Das Client-Programm benutzt die *Position*-Objekte, um die Liste mit den Suchergebnissen im Suchdialog zu füllen.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0/
/EN" "http://java.sun.com/dtd/ejb-jar_2_0.dtd">
<ejb-jar>
    <enterprise-beans>
        <entity>
            <ejb-name>Store</ejb-name>
            <home>ejb.store.StoreHome</home>
            <remote>ejb.store.Store</remote>
            <ejb-class>ejb.store.StoreBean</ejb-class>
            <persistence-type>Container</persistence-type>
            <prim-key-class>java.lang.Integer</prim-key-class>
            <reentrant>False</reentrant>
            <cmp-version>2.x</cmp-version>
            <abstract-schema-name>Store</abstract-schema-name>
            <cmp-field>
              <field-name>id</field-name>
            </cmp-field>
            <cmp-field>
              <field-name>name</field-name>
            </cmp-field>
            <primkey-field>id</primkey-field>
            <query>
              <query-method>
                <method-name>findAllStores</method-name>
                <method-params>
                </method-params>
              </query-method>
              <ejb-ql>
                SELECT OBJECT(s) FROM Store AS s
              </ejb-ql>
            </query>
            <query>
              <query-method>
                <method-name>ejbSelectAllItems</method-name>
                <method-params>
                </method-params>
              </query-method>
              <result-type-mapping>Local</result-type-mapping>
              <ejb-ql>
                SELECT DISTINCT p.item FROM Store AS s,
                                            IN (s.racks) AS r,
                                            IN(r.positions) AS p
              </ejb-ql>
            </query>
            <query>
```

```
        <query-method>
          <method-name>ejbSelectPositionsForItem
          </method-name>
          <method-params>
            <method-param>java.lang.Integer</method-param>
          </method-params>
        </query-method>
        <result-type-mapping>Local</result-type-mapping>
        <ejb-ql>
          SELECT OBJECT(p) FROM Store AS s,
                                IN (s.racks) AS r,
                                IN(r.positions) AS p
                        WHERE p.itemId = ?1
        </ejb-ql>
      </query>
    </entity>
    <entity>
      <ejb-name>RackLocal</ejb-name>
      <local-home>ejb.store.rack.RackLocalHome</local-home>
      <local>ejb.store.rack.RackLocal</local>
      <ejb-class>ejb.store.rack.RackBean</ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>java.lang.Integer</prim-key-class>
      <reentrant>False</reentrant>
      <cmp-version>2.x</cmp-version>
      <abstract-schema-name>Rack</abstract-schema-name>
      <cmp-field>
        <field-name>id</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>description</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>storeId</field-name>
      </cmp-field>
      <primkey-field>id</primkey-field>
    </entity>
    <entity>
      <ejb-name>PositionLocal</ejb-name>
      <local-home>
          ejb.store.rack.position.PositionLocalHome
      </local-home>
      <local>ejb.store.rack.position.PositionLocal</local>
      <ejb-class>
          ejb.store.rack.position.PositionBean
      </ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>
          ejb.store.rack.position.PositionPK
      </prim-key-class>
      <reentrant>False</reentrant>
```

```
            <cmp-version>2.x</cmp-version>
            <abstract-schema-name>Position</abstract-schema-name>
            <cmp-field>
              <field-name>rackId</field-name>
            </cmp-field>
            <cmp-field>
              <field-name>row</field-name>
            </cmp-field>
            <cmp-field>
              <field-name>column</field-name>
            </cmp-field>
            <cmp-field>
              <field-name>itemId</field-name>
            </cmp-field>
            <cmp-field>
              <field-name>quantity</field-name>
            </cmp-field>
        </entity>
        <entity>
            <ejb-name>ItemLocal</ejb-name>
            <local-home>ejb.store.item.ItemLocalHome</local-home>
            <local>ejb.store.item.ItemLocal</local>
            <ejb-class>ejb.store.item.ItemBean</ejb-class>
            <persistence-type>Container</persistence-type>
            <prim-key-class>java.lang.Integer</prim-key-class>
            <reentrant>False</reentrant>
            <cmp-version>2.x</cmp-version>
            <abstract-schema-name>Item</abstract-schema-name>
            <cmp-field>
              <field-name>id</field-name>
            </cmp-field>
            <cmp-field>
              <field-name>description</field-name>
            </cmp-field>
            <cmp-field>
              <field-name>supplier</field-name>
            </cmp-field>
            <primkey-field>id</primkey-field>
        </entity>
    </enterprise-beans>
    <relationships>
        <ejb-relation>
            <ejb-relation-name>Store-Rack</ejb-relation-name>
            <ejb-relationship-role>
                <ejb-relationship-role-name>
                    store-has-racks
                </ejb-relationship-role-name>
                <multiplicity>One</multiplicity>
                <relationship-role-source>
                    <ejb-name>Store</ejb-name>
                </relationship-role-source>
```

```
            <cmr-field>
                <cmr-field-name>racks</cmr-field-name>
                <cmr-field-type>
                    java.util.Collection
                </cmr-field-type>
            </cmr-field>
        </ejb-relationship-role>
        <ejb-relationship-role>
            <ejb-relationship-role-name>
                rack-belongs-to-store
            </ejb-relationship-role-name>
            <multiplicity>Many</multiplicity>
            <cascade-delete/>
            <relationship-role-source>
                <ejb-name>RackLocal</ejb-name>
            </relationship-role-source>
        </ejb-relationship-role>
    </ejb-relation>
    <ejb-relation>
        <ejb-relation-name>Rack-Position</ejb-relation-name>
        <ejb-relationship-role>
            <ejb-relationship-role-name>
                rack-has-positions
            </ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <relationship-role-source>
                <ejb-name>RackLocal</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>positions</cmr-field-name>
                <cmr-field-type>
                    java.util.Collection
                </cmr-field-type>
            </cmr-field>
        </ejb-relationship-role>
        <ejb-relationship-role>
            <ejb-relationship-role-name>
                position-belongs-to-rack
            </ejb-relationship-role-name>
            <multiplicity>Many</multiplicity>
            <cascade-delete/>
            <relationship-role-source>
                <ejb-name>PositionLocal</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>rack</cmr-field-name>
            </cmr-field>
        </ejb-relationship-role>
    </ejb-relation>
    <ejb-relation>
        <ejb-relation-name>Position-Item</ejb-relation-name>
```

```
            <ejb-relationship-role>
                <ejb-relationship-role-name>
                    position-has-item
                </ejb-relationship-role-name>
                <multiplicity>Many</multiplicity>
                <relationship-role-source>
                    <ejb-name>PositionLocal</ejb-name>
                </relationship-role-source>
                <cmr-field>
                    <cmr-field-name>item</cmr-field-name>
                </cmr-field>
            </ejb-relationship-role>
            <ejb-relationship-role>
                <ejb-relationship-role-name>
                    item-is-in-position(s)
                </ejb-relationship-role-name>
                <multiplicity>One</multiplicity>
                <relationship-role-source>
                    <ejb-name>ItemLocal</ejb-name>
                </relationship-role-source>
            </ejb-relationship-role>
        </ejb-relation>
    </relationships>
</ejb-jar>
```

Listing 5.45: Deployment-Deskriptor der Lagerverwaltung

5.6.3 Zusammenfassung

Store-, Rack-, Position- und Item-Bean können als eine Art Superkomponente verstanden werden. Sie funktioniert nur durch das Zusammenspiel aller Enterprise-Bean Komponenten. Die Store-Bean ist die Kontrollinstanz dieser Superkomponente und zugleich die Schnittstelle für den Client. Die anderen Entity-Beans treten nach außen hin nicht in Erscheinung. Die Kopplung der Enterprise-Beans zu einer Superkomponente übernimmt der EJB-Container durch die persistenten Beziehungen. Er weiß, welche Instanzen der jeweiligen Enterprise-Beans zusammengehören und sorgt für die Auflösung der Beziehungen zur Laufzeit. Er erzeugt die Bean-Instanzen und versorgt sie mit den Daten aus der Datenbank, wann immer über eine Beziehung navigiert wird. Die Kommunikation der Enterprise-Beans untereinander erfolgt dabei über den Local-Client-View, was für die nötige Effizienz sorgt. Die Abfragesprache EJB-QL in Verbindung mit Select- oder Finder-Methoden ist eine Möglichkeit, um innerhalb der Superkomponente wahlfrei auf bestimmte Instanzen der verbundenen Enterprise-Beans zuzugreifen. Mit Hilfe von persistenten Beziehungen (in Verbindung mit dem Local-Client-View), Select- und Finder-Methoden sowie der Abfragesprache EJB-QL hat sich dieses Problem sehr elegant und mit relativ wenig Programmieraufwand auf der Server-Seite lösen lassen. Eine zentrale Rolle spielt dabei der Deployment-Deskriptor. Er beinhaltet Teile der Implementierung (EJB-QL Abfragen) und legt den Grund-

stein für die Kopplung der Enterprise-Bean-Instanzen über persistente Beziehungen. Somit können in diesen Bereichen der Anwendung Änderungen gemacht werden, ohne den Code der Enterprise-Beans verändern zu müssen. Die Lösung sollte außerdem ohne weiteres auf andere (EJB 2.0 konforme) Applikationsserver und andere Datenbanken portiert werden können.

5.7 Container-Managed-Persistence 1.1

Die EJB-Spezifikation 2.0 beinhaltet nach wie vor Container-Managed-Persistence nach der Version 1.1, um abwärtskompatibel zu bleiben. Die Entity-Beans bestehender Anwendungen können dadurch schrittweise nach 2.0 migriert werden. Die Änderungen in der Version 2.0 im Bereich Container-Managed-Persistence sind sehr umfangreich und haben viele Vorteile gegenüber der Version 1.1. Dazu zählen vor allem persistente Beziehungen und der Local-Client-View. Für die Entwicklung neuer Anwendungen und Anwendungsmodule, in denen Persistenz eine Rolle spielt, sollten auf jeden Fall CMP Entity-Beans nach der Version 2.0 verwendet werden.

5.7.1 Überblick

Abbildung 5.18 gibt einen vollständigen Überblick über die Klassen und Interfaces von Entity-Beans mit Container-Managed-Persistence der Version 1.1. Eine EJB 1.1 Entity-Bean ist auf die Benutzung des Remote-Client-Views beschränkt. Sie kann kein Local-Home- und kein Local-Interface haben. Außerdem gibt es bei EJB 1.1 Entity-Beans keine Home-Methoden und keine Select-Methoden.

Rechts in Abbildung 5.18 steht die Entity-Bean-Klasse. Instanzen dieser Klasse werden zur Laufzeit durch den EJB-Container verwaltet. Auf der linken Seite befinden sich die Schnittstellen, die auch auf dem Client verfügbar sind und durch das Home- bzw. Remote-Objekt des EJB-Containers implementiert werden (vgl. dazu auch Kapitel 3).

Jede Entity-Bean-Klasse implementiert das Interface *javax.ejb.EntityBean*. Im Gegensatz zu EJB 2.0 Entity-Beans ist die Bean-Klasse einer EJB 1.1 Entity-Bean eine konkrete Klasse. Alle öffentlichen Attribute der Klasse sind per definitionem persistent, wenn sie auch im Deployment-Deskriptor als persistente Attribute definiert sind (siehe unten). In der Entity-Bean-Klasse implementiert der Bean-Entwickler die eigentliche Funktionalität der Bean.

Oben links in Abbildung 5.18 steht die Klasse für den Primärschlüssel der Bean. Da Objekte dieser Klasse zwischen Server und Client über das Netzwerk transportiert werden, muss die Klasse serialisierbar sein. Sie implementiert das Interface *java.io.Serializable*. Alle persistenten Attribute der Entity-Bean-Klasse, die zum Primärschlüssel gehören, werden vom Bean-Entwickler in die Primärschlüsselklasse übertragen. Der Name und der Datentyp dieser Attribute ist in der Entity-Bean-Klasse und der Primärschlüsselklasse identisch.

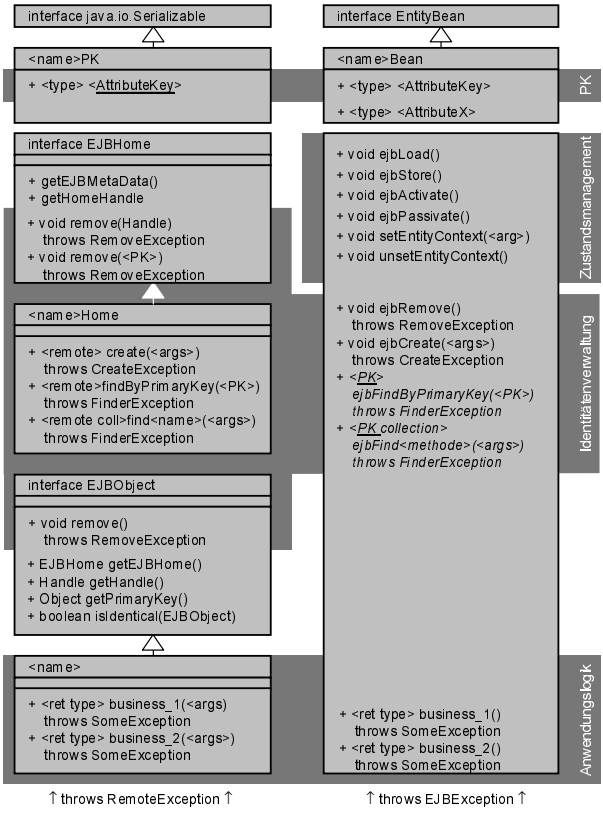

Abbildung 5.18: Remote-Interfaces und Klassen der CMP Entity-Bean (EJB 1.1)

Die Bestandteile der Entity-Bean-Klasse lassen sich in vier Gruppen aufteilen:

- Attribute,
- Zustandsmanagement,
- Identitätenverwaltung,
- Anwendungslogik.

Wir wollen im weiteren Verlauf dieses Abschnitts die einzelnen Bestandteile im Detail betrachten.

5.7.2 Attribute

Jedes persistente Attribut wird in der (konkreten) Bean-Klasse als Member-Variable definiert. Die persistenten Attribute müssen *public* und dürfen nicht *final* sein, da sie sonst vom EJB-Container nicht modifiziert werden können. Sie dürfen folgende Datentypen haben:

- primitive Datentypen der Programmiersprache Java (*int*, *float*, *long* etc.),
- serialisierbare Datentypen der Programmiersprache Java (dazu zählen insbesondere *java.lang.String*, *java.lang.Integer* etc.),
- Referenzen auf das Remote- oder Home-Interface anderer Enterprise-Beans.

Listing 5.46 zeigt ein Beispiel, wie eine EJB 1.1 Entity-Bean persistente Attribute in der Bean-Klasse definiert.

```
public class AddressBean implements EntityBean {

    public String name;
    public String firstName;
    public String street;
    public int    postalCode;
    public String city;

    ...

}
```

Listing 5.46: Persistente Attribute einer EJB 1.1 Entity-Bean

Die Attribute sind letztlich erst dann persistente Attribute, wenn sie im Deployment-Deskriptor als solche deklariert sind. Listing 5.47 zeigt, wie die persistenten Attribute aus Listing 5.46 im Deployment-Deskriptor deklariert werden.

```
...
  <enterprise-beans>
    <entity>
      <ejb-name>Address</ejb-name>
      <home>AddressHome</home>
      <remote>Address</remote>
      <ejb-class>AddressBean</ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>java.lang.String</prim-key-class>
      <reentrant>False</reentrant>
      <cmp-version>1.x</cmp-version>
      <cmp-field>
        <field-name>name</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>firstName</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>street</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>postalCode</field-name>
```

```
        </cmp-field>
        <cmp-field>
          <field-name>city</field-name>
        </cmp-field>
        <primkey-field>name</primkey-field>
      </entity>
    </enterprise-beans>
...
```

Listing 5.47: Deplyoment-Deskriptor einer EJB 1.1 Entity-Bean

Bei EJB 1.1 Entity-Beans wird im Element *cmp-version* der Wert *1.x* eingetragen. Die Elemente *abstract-schema-name* und *query* gibt es bei EJB 1.1 Entity-Beans nicht. Sonst sind die Angaben weitgehend identisch mit EJB 2.0 Entity-Beans.

Für die Initialisierung und die Synchronisation der persistenten Attribute mit dem Persistenzmedium ist der EJB-Container verantwortlich. Die Anwendungsmethoden der Entity-Bean (die vom Client aufgerufen werden), können lesend und schreibend auf die Attribute zugreifen.

5.7.3 Zustandsmanagement

Die Methoden des Zustandsmanagements (sieheAbbildung 5.18) sind sogenannte Callback-Methoden, die vom EJB-Container aufgerufen werden, um die Entity-Bean über einen Zustandswechsel zu informieren. Die einzelnen Zustände und Zustandsübergänge sind in Abbildung 5.6 dargestellt. Die Methoden haben die gleiche Funktion wie bei EJB 2.0 Entity-Beans, sollen hier aber nochmal kurz erläutert werden:

```
void setEntityContext(EntityContext ctx)
```

Die Methode *setEntityContext* wird vom EJB-Container aufgerufen, wenn die Bean-Instanz erzeugt wird und in den Zustand *Pooled* wechselt. Sie dient zur Initialisierung der Bean. Der EJB-Container übergibt der Entity-Bean ihren *EntityContext*. Diese Methode sollte nur den *EntityContext* speichern und darüber hinaus keine Funktionalität besitzen.

```
void unsetEntityContext()
```

Diese Methode wird vom EJB-Container aufgerufen, wenn er die Bean-Instanz nicht mehr benötigt und aus dem Pool nehmen will. Der Aufruf teilt der Bean-Instanz mit, dass der mit ihr assoziierte *EntityContext* ungültig wird.

```
void ejbActivate()
```

Der EJB-Container ruft diese Methode auf, um der Entity-Bean-Instanz mitzuteilen, dass sie eine bestimmte Entity-Bean-Identität bekommen hat. Im Entity-Kontext kann jetzt der Primärschlüssel abgefragt werden. Die Bean-Instanz wechselt in den Zustand *Ready-Async*.

```
void ejbPassivate()
```

Diese Methode bildet das Gegenstück zu *ejbActivate*. Sie wird vom EJB-Container aufgerufen, wenn die Bean-Instanz vom Zustand *Ready* zurück in den Zustand *Pooled* wechselt. Die Bean-Instanz besitzt danach keine Bean-Identität mehr.

```
void ejbLoad()
```

Durch den Aufruf dieser Methode informiert der EJB-Container die Bean-Instanz darüber, dass er die persistenten Attribute der Bean neu geladen hat. Dies ist erforderlich, falls die Bean-Instanz noch nicht initialisiert wurde oder falls der Datenbankinhalt sich durch einen parallelen Zugriff geändert hat. Der EJB-Container kann die Bean zu jedem Zeitpunkt im Zustand *Ready* synchronisieren. Häufig geschieht dies, bevor eine Methode der Anwendungslogik aufgerufen wird. Die Bean wechselt vom Zustand *Ready-Async* in den Zustand *Ready-Sync*.

Die Methode kann von der Entity-Bean z.B. dazu benutzt werden, um transiente Variablen aus den soeben mit dem Persistenzmedium synchronisierten persistenten Variablen neu zu berechnen.

```
void ejbStore()
```

Durch den Aufruf dieser Methode informiert der EJB-Container die Bean-Instanz darüber, dass er die persistenten Attribute der Bean in das Persistenzsystem schreiben wird. Die Bean-Instanz wechselt vom Zustand *Ready-Update* in den Zustand *Ready-Sync*.

5.7.4 Verwaltung der Identitäten

Die Methoden für die Identitätenverwaltung bilden die zweite Gruppe in Abbildung 5.18. Sie ermöglichen das Erzeugen, Löschen und Suchen von Entity-Bean-Identitäten. Diese Methoden finden sich im Home-Interface wieder und stehen dem Client zur Verfügung, schon bevor er eine Bean-Instanz kennt.

```
ejbFind<name>(<args>)
```

In dieser Gruppe gibt es eine oder mehrere Suchmethoden. Die Methoden sind im Home-Interface definiert. Die Treffermenge einer Suchmethode kann entweder eine einzelne Entity-Bean oder eine Menge von Entity-Beans sein. Die Methoden zum Suchen von Entity-Beans beginnen mit dem Präfix *find*. Jede Entity-Bean muss mindestens eine Methode *findByPrimaryKey* mit dem Primärschlüssel als einzigen Übergabeparameter definieren. Die restlichen Finder-Methoden können beliebige Signaturen haben. Die Finder-Methoden werden in der Bean-Klasse weder deklariert noch implementiert, sondern vom EJB-Container generiert (die diversen *ejbFind*-Methoden in der Bean-Klasse in Abbildung 5.18 sind deswegen auch kursiv markiert). Die *findByPrimaryKey*-Methode zu generieren ist für den EJB-Container nicht weiter schwierig. Für die

restlichen Finder-Methoden benötigt er Angaben in Form von Suchabfragen. Wie bereits erwähnt kennt EJB 1.1 kein EJB-QL. Die Formulierung solcher Suchanfragen ist daher abhängig von dem, was der jeweilige Applikationsserver anbietet. Dieser Umstand macht EJB 1.1 Entity-Beans nur schwer portierbar.

```
<primKeyClass> ejbCreate(<args>)
```

Zum Erzeugen von Beans gibt es eine oder mehrere Methoden, die alle *ejbCreate* heißen und als Rückgabewert die Primärschlüsselklasse der Bean definieren. Die Methoden unterscheiden sich durch ihre Parameter und können vom Entwickler sonst frei definiert werden. Die Methoden sind nicht Bestandteil des Entity-Bean-Interface, da die Parameter- und Rückgabetypen für jede Bean-Klasse unterschiedlich sind.

Die Methoden stehen dem Client zur Verfügung und werden deshalb auch im Home-Interface der Bean deklariert. Die Signaturen der *create*-Methoden im Home-Interface (*create*) und in der Bean-Klasse (*ejbCreate*) unterscheiden sich durch den Rückgabewert. Die Methoden im Home-Interface haben das Remote-Interface der Bean als Rückgabewert. Die Methoden an der Bean-Klasse verwenden den Primärschlüssel. Der EJB-Container leistet die erforderliche Konvertierung. Die Implementierung dieser Methode verwendet die Parameter, um die persistenten Attribute der Entity-Bean zu initialisieren. Als Ergebnis wird immer *null* zurückgegeben.

```
void ejbPostCreate(<args>)
```

Zu jeder *ejbCreate*-Methode muss der Bean-Entwickler eine *ejbPostCreate*-Methode mit den gleichen Parametertypen definieren. Die *ejbPostCreate*-Methode wird vom EJB-Container immer nach der entsprechenden *ejbCreate*-Methode mit dem gleichen Transaktionskontext ausgeführt (Details zu Transaktion siehe Kapitel 7).

In diesen Methoden können weitere Initialisierungsschritte ausgeführt werden. Im Gegensatz zu den *ejbCreate*-Methoden steht diesen Methoden die Bean-Identität zur Verfügung.

Mit der Ausführung von *ejbCreate* und *ejbPostCreate* wechselt die Bean-Instanz vom Zustand *Pooled* in den Zustand *Ready-Update*. In der Datenbank gesichert wird die neue Bean-Identität jedoch häufig erst mit der darauffolgenden Ausführung der Methode *ejbStore*.

```
void ejbRemove()
```

Zur Verwaltung von Bean-Identitäten gehören auch die Methoden zum Löschen von Beans. Der Client gibt den Auftrag zum Löschen einer Entity-Bean-Identität durch den Aufruf der Methode *remove* im Home- oder Remote-Interface. Die Methode *ejbRemove* an der Bean-Instanz wird jedoch vom EJB-Container aufgerufen, der den Aufruf des Clients entsprechend weiterleitet.

Die Bean-Instanz wird lediglich über den bevorstehenden Zustandswechsel informiert. Für die Löschung der Daten sorgt der EJB-Container. Eventuell müssen in dieser Methode zusätzliche Ressourcen freigegeben werden, bevor die Bean-Instanz in den Zustand *Pooled* wechselt.

Wie *ejbPassivate* wird diese Methode aufgerufen, wenn der Zustand einer Bean-Instanz von *Ready* auf *Pooled* wechselt. Hier wird jedoch die Bean-Identität gelöscht. Nach dem Aufruf hat die Bean-Instanz keine Bean-Identität mehr. Etwaige belegte Ressourcen müssen freigegeben werden, bevor die Bean-Instanz in den Zustand *Pooled* wechselt. Die Methode *ejbPassivate* wird bei diesem Zustandsübergang nicht aufgerufen.

5.7.5 Anwendungslogik

Die vierte Gruppe sind die Methoden für die Anwendungslogik, die sich auch im Remote-Interface wiederfinden. Diese Methoden werden vom Bean-Entwickler in der Bean-Klasse implementiert. Die Definition in der Bean-Klasse hat die gleiche Signatur wie die Definition im Remote-Interface. Bei diesen Methoden wird in der Regel lesend und schreibend auf die persistenten Attribute zugegriffen.

5.8 Bean-Managed-Persistence

Im Gegensatz zu den Entity-Beans mit Container-Managed-Persistence kümmern sich Entity-Beans mit Bean-Managed-Persistence selbst um die Kommunikation mit dem Persistenzmedium (z.B. eine Datenbank). Alle Zugriffe zum Lesen und Schreiben der Daten werden vom Bean-Provider in der Bean-Klasse programmiert. Der EJB-Container weiß nicht, welche Daten der Entity-Bean persistent sind oder in welchem Persistenzmedium sie gespeichert werden.

Entity-Beans mit Bean-Managed-Persistence bedeuten für den Bean-Entwickler zweifelsohne mehr Programmieraufwand. Unter Umständen sind sie nur eingeschränkt portierbar, denn der Code der Bean wird oft auf ein bestimmtes Speichermedium hin optimiert. Trotzdem gibt es Fälle, in denen man keine Container-Managed-Persistence verwenden will oder kann. Wenn eine Entity-Bean z.B. Bilder in Form von Gif-Dateien in der Datenbank speichern soll und dazu den Datentyp BLOB (Binary Large Objekt) verwenden muss. Gerade dieser Datentyp verlangt oft eine spezielle Handhabung, die der EJB-Container resp. der Persistence-Manager nicht unterstützt. Ein anderes Beispiel wäre, wenn die Entity-Bean die Daten nicht in einer Datenbank sondern in einem elektronischen Archiv speichert. Für solche Systeme gibt es von Seiten des EJB-Containers bzw. des Persistence-Managers keine Unterstützung. Muss so ein System als Speichermedium benutzt werden, kann die Persistenz über Entity-Beans mit Bean-Managed-Persistence realisiert werden.

5.8.1 Überblick

Abbildung 5.19 gibt einen vollständigen Überblick über die Klassen und Interfaces einer Entity-Bean mit Bean-Managed-Persistence, die den Remote-Client-View unterstützt. Rechts steht die Entity-Bean-Klasse, deren Instanzen zur Laufzeit vom EJB-Container verwaltet werden. Auf der linken Seite befinden sich die Schnittstellen, die durch das Home- und das Remote-Objekt des EJB-Containers implementiert werden (siehe auch Kapitel 3). Der Client bekommt über diese Interfaces Zugriff auf die Entity-Bean.

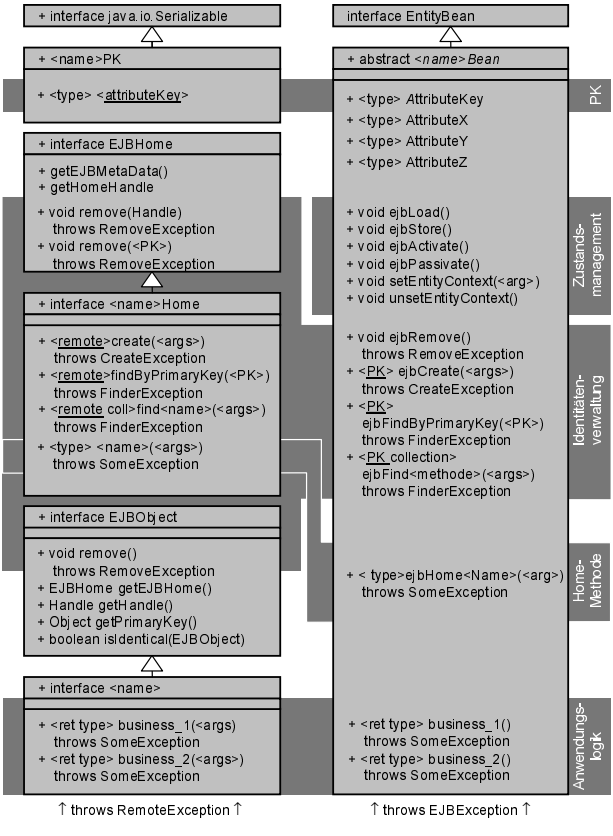

Abbildung 5.19: Remote-Interfaces und Klassen der BMP Entity-Bean

Jede Entity-Bean-Klasse implementiert das Interface *javax.ejb.EntityBean*. In dieser Entity-Bean-Klasse implementiert der Bean-Entwickler die eigentliche Funktionalität der Bean und auch ihren Persistenzmechanismus. Der Entwickler der Bean hat weitgehende Freiheit bei der Organisation der persistenten Daten. Er muss die persistenten Attribute nicht im Deployment-Deskriptor angeben.

Auch bei Bean-Managed-Persistence wird ein Primärschlüssel definiert, der jede Instanz der Bean-Klasse eindeutig identifiziert. Die Primärschlüsselklasse steht oben links in der Abbildung. Sie implementiert das Interface *java.io.Serializable*.

Abbildung 5.20 zeigt die Klassen und Interfaces einer Entity-Bean mit Bean-Managed-Persistence, die den Local-Client-View unterstützt. Rechts steht die Entity-Bean-Klasse, auf der linken Seite befinden sich die Schnittstellen, die vom lokalen Client benutzt werden und durch das Local-Home- bzw. Local-Objekt des EJB-Containers implementiert werden. Die Entity-Bean-Klasse des Local-Client-Views ist identisch mit der des Remote-Client-Views. Die Unterschiede zwischen den beiden Fällen werden vom EJB-Container durch die Implementierung der einschlägigen Interfaces gehandhabt. Die Bean-Klasse bzw. die Bean-Instanz bleibt davon unberührt. Die Unterschiede für den Client einer Entity-Bean zwischen Local- und Remote-Client-View sind die gleichen wie bei Session-Beans. Die Unterschiede wurden bereits in Kapitel 4 diskutiert.

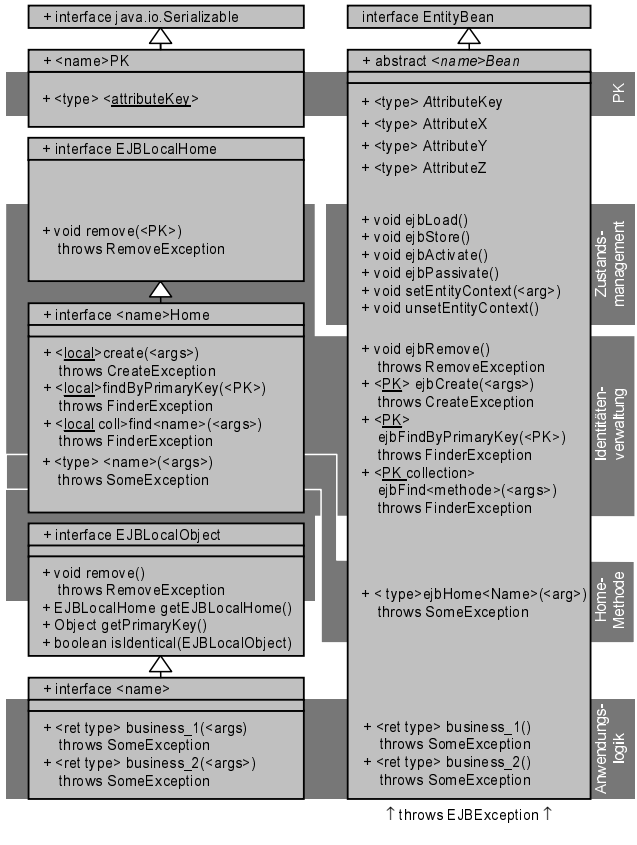

Abbildung 5.20: Local-Interfaces und Klassen der BMP Entity-Bean

Die Bestandteile der Entity-Bean mit Bean-Managed-Persistence lassen sich in vier Gruppen aufteilen:

- Attribute,
- Zustandsmanagement,
- Identitätenverwaltung,
- Anwendungslogik.

Das Beispiel am Ende dieses Abschnitts bietet einen praktischen Einstieg in die Programmierung von Entity-Beans mit Bean-Managed-Persistence. Alle Bestandteile der Entity-Bean werden dort im Detail diskutiert.

5.8.2 Attribute

Anders als bei Entity-Beans mit Container-Managed-Persistence ist der Bean-Provider in der Definition der Attribute völlig frei. Sichtbarkeit und Typ der Attribute können vom Bean-Provider völlig frei gewählt werden. Schließlich ist er selbst dafür verantwortlich, die Attribute mit dem Speichermedium zu synchronisieren.

5.8.3 Zustandsmanagement

An oberster Stelle in Abbildung 5.19 bzw. Abbildung 5.20 stehen die Methoden für das Zustandsmanagement. Der EJB-Container ruft diese Methoden auf, um Zustandsübergänge einzuleiten bzw. um die Bean über einen Zustandsübergang zu informieren (siehe Abbildung 5.6).

```
void setEntityContext(EntityContext ctx)
```

Die Methode *setEntityContext* wird vom EJB-Container aufgerufen, wenn die Bean-Instanz erzeugt wird und in den Zustand *Pooled* wechselt. Sie dient zur Initialisierung der Bean. Der EJB-Container übergibt der Entity-Bean ihren *EntityContext*. Diese Methode sollte nur den *EntityContext* speichern und darüber hinaus keine Funktionalität besitzen.

```
void unsetEntityContext()
```

Diese Methode wird vom EJB-Container aufgerufen, wenn er die Bean-Instanz nicht mehr benötigt und aus dem Pool nehmen will. Der Aufruf teilt der Bean-Instanz mit, dass der mit ihr assoziierte *EntityContext* ungültig wird.

```
void ejbActivate()
```

Der EJB-Container ruft diese Methode auf, um der Entity-Bean-Instanz mitzuteilen, dass sie eine bestimmte Entity-Bean-Identität bekommen hat. Die Bean-Instanz wechselt in den Zustand *Ready-Async*. In diesem Zustand kann im *EntityContext* schon der

Primärschlüssel mit *getPrimaryKey* und das Remote-Interface mit *getEJBObject* abgefragt werden. Die Bean-Instanz wurde jedoch noch nicht mit der Datenbank synchronisiert, und somit sind die Werte der persistenten Attribute noch nicht gesetzt.

Diese Methode kann genutzt werden, um Datenbankverbindungen zu öffnen oder andere Ressourcen zu reservieren. Zu beachten ist dabei, dass der Methode keine Transaktion zur Verfügung steht.

```
void ejbPassivate()
```

Diese Methode bildet das Gegenstück zu *ejbActivate*. Sie wird vom EJB-Container aufgerufen, wenn die Bean-Instanz vom Zustand *Ready* zurück in den Zustand *Pooled* wechselt. Die Bean-Instanz besitzt danach keine Bean-Identität mehr. In der Methode kann jedoch die Bean-Identität noch verwendet werden.

Diese Methode wird verwendet, um die Ressourcen wieder freizugeben, die mit *ejbActivate* oder später im Zustand *Ready* reserviert wurden.

```
void ejbLoad()
```

Die Methode *ejbLoad* wird vom EJB-Container aufgerufen, um den Zustand der Bean-Instanz mit dem Persistenzmedium zu synchronisieren. Dies ist erforderlich, falls die Bean-Instanz noch nicht initialisiert wurde oder falls sich z. B. der Datenbankinhalt durch einen parallelen Zugriff geändert hat. Der EJB-Container kann die Bean zu jedem Zeitpunkt im Zustand *Ready* synchronisieren. Häufig geschieht dies, bevor eine Methode der Anwendungslogik aufgerufen wird. Die Bean wechselt vom Zustand *Ready-Async* in den Zustand *Ready-Sync*.

Mit dieser Methode programmiert der Bean-Provider die Funktionalität zum Laden der Attributwerte aus dem Persistenzmedium. Zusätzlich kann es erforderlich sein, transiente Attribute neu zu berechnen.

Der Transaktionskontext für die Methode lässt sich nur schwer vorhersagen. Die Methode, die die Synchronisation verursacht, bestimmt auch den Transaktionskontext. Wenn beispielsweise ein EJB-Container vor jedem Methodenaufruf der Anwendungslogik die Bean synchronisiert, so bestimmt das Transaktionsattribut der folgenden Methode den Transaktionskontext (zu Transaktionen siehe Kapitel 7).

```
void ejbStore()
```

Wenn ein Client den Zustand einer Bean verändert, müssen die veränderten Attributwerte in das Persistenzmedium zurückgeschrieben werden. Dadurch wechselt die Bean-Instanz vom Zustand *Ready-Update* in den Zustand *Ready-Sync*. Mit dieser Methode wird das Speichern der persistenten Attribute implementiert.

Normalerweise wird die Methode vom EJB-Container aufgerufen, bevor dieser eine Transaktion beendet. Alle zwischengespeicherten Daten müssen in die Datenbank geschrieben werden, da sie sonst nicht mit dem Transaktionsabschluss persistent werden (zu Transaktionen siehe Kapitel 7).

5.8.4 Verwaltung der Identitäten

Die Methoden für die Identitätenverwaltung bilden die zweite Gruppe in Abbildung 5.19 bzw. Abbildung 5.20. Sie ermöglichen das Erzeugen, Löschen und Suchen von Entity-Bean-Identitäten. Diese Methoden finden sich im Home-Interface wieder und stehen dem Client zur Verfügung, schon bevor er eine Bean-Instanz kennt.

```
ejbFind<name>(<args>)
```

Es gibt eine oder mehrere Suchmethoden. Die Treffermenge einer Suchmethode kann entweder eine einzelne Entity-Bean oder eine Menge mehrerer Entity-Beans sein. Die Methoden sind im Home-Interface definiert und stehen dort dem Client zur Verfügung. Im Gegensatz zu Container-Managed-Persistence müssen die zugehörigen Suchmethoden vom Bean-Entwickler in der Bean-Klasse implementiert werden. Bei Bean-Managed-Persistence gibt es keine Möglichkeit, Finder-Methoden deklarativ mittels EJB-QL zu implementieren.

Wie bei Container-Managed-Persistence muss es mindestens eine Methode *findByPrimaryKey* im (Local-) Home-Interface der Bean geben, die in der Bean-Klasse als *ejbFindByPrimaryKey* implementiert wird. Der Bean-Provider kann je nach Bedarf weitere Finder-Methoden definieren, die nach dem gleichen Namensschema aufgebaut sind.

Zu beachten ist, dass die Suchmethoden der Entity-Bean ein oder mehrere Instanzen der Primärschlüsselklasse zurückgeben, während die Methoden im (Local-) Home-Interface eine oder mehrere Referenzen auf Beans zurückgeben werden. Die nötige Konvertierung leistet der EJB-Container. Er stellt sicher, dass eine Bean-Instanz mit der entsprechenden Bean-Identität aktiviert ist, und übergibt deren Remote- bzw. Local-Interface an den Client.

Suchen, die nur einen Treffer haben können, verwenden direkt die Klasse des Primärschlüssels respektive das Remote- bzw. Local-Interface als Datentyp für den Rückgabewert. Kann eine Suche mehrere Treffer haben, so wird *java.util.Enumeration* (Java 1.1.) oder *java.util.Collection* (Java 1.2) als Datentyp für den Rückgabewert verwendet.

Alle Suchmethoden können vom EJB-Container in einer Transaktion ausgeführt werden. Das Transaktionsattribut der entsprechenden Methoden im (Local-) Home-Interface bestimmt das Verhalten des EJB-Containers (zu Transaktionen siehe Kapitel 7).

Bei Bean-Managed-Persistence gibt es keine Select-Methoden.

```
<primKeyClass> ejbCreate(<args>)
```

Zum Erzeugen von Entity-Beans gibt es eine oder mehrere Methoden, die alle *ejbCreate* heißen und als Rückgabewert die Primärschlüsselklasse der Bean definieren. Die Methoden unterscheiden sich durch ihre Parameter und können vom Entwickler sonst frei definiert werden. Die Methoden sind nicht Bestandteil des Entity-Bean-Interface, da die Parameter- und Rückgabetypen für jede Bean-Klasse unterschiedlich sind.

Die Methoden stehen dem Client zur Verfügung und werden deshalb auch im (Local-) Home-Interface der Bean deklariert. Die Methoden heißen dort *create(...)* und unterscheiden sich sonst durch den Rückgabewert von ihren Pendants in der Bean-Klasse. Die Methoden im (Local-) Home-Interface haben das Remote-bzw. Local-Interface der Bean als Rückgabewert. Die entsprechenden *ejbCreate*-Methoden in der Bean-Klasse verwenden dafür den Primärschlüssel. Der EJB-Container leistet die erforderliche Konvertierung sowie das Mapping der *create*-Methoden des (Local-) Home-Interfaces auf die *ejbCreate*-Methoden der Bean-Klasse.

In den *ejbCreate*-Methoden der Entity-Bean programmiert der Bean-Provider das Anlegen eines entsprechenden Datensatzes im Persistenzsystem. Der Transaktionskontext der *ejbCreate*-Methoden ist von den Transaktionsattributen der Methoden im Deployment-Deskriptor abhängig.

```
void ejbPostCreate(<args>)
```

Zu jeder *ejbCreate*-Methode definiert der Bean-Entwickler eine *ejbPostCreate*-Methode mit den gleichen Parametertypen. Die *ejbPostCreate*-Methode wird vom EJB-Container immer nach der entsprechenden *ejbCreate*-Methode mit dem gleichen Transaktionskontext ausgeführt.

In diesen Methoden können weitere Initialisierungsschritte ausgeführt werden. Im Gegensatz zu den *ejbCreate*-Methoden steht diesen Methoden die Bean-Identität zur Verfügung. Mit der Ausführung von *ejbCreate* und *ejbPostCreate* wechselt die Bean-Instanz vom Zustand *Pooled* in den Zustand *Ready-Update*.

```
void ejbRemove()
```

Zur Verwaltung von Bean-Identitäten gehören auch die Methoden zum Löschen von Beans. Der Client gibt den Auftrag zum Löschen einer Entity-Bean-Identität durch den Aufruf der Methode *remove* im (Local-) Home-, Local- oder Remote-Interface. Der Aufruf wird vom EJB-Container an die Methode *ejbRemove* der Bean-Instanz delegiert.

In dieser Methode programmiert der Bean-Provider das Löschen des Datensatzes im Persistenzsystem. Nach der Ausführung der Methode darf die Bean-Identität im Persistenzsystem nicht mehr existieren. Zusätzlich belegte Ressourcen müssen in dieser Methode ebenfalls freigegeben werden. Die Bean-Instanz wechselt in den Zustand *Pooled* und kann für andere Bean-Identitäten verwendet werden. Die Methode *ejbPassivate* wird bei diesem Zustandsübergang nicht aufgerufen.

Auch die Methode *ejbRemove* kann vom EJB-Container mit einer Transaktion aufgerufen werden. Das zugehörige Transaktionsattribut im Deployment-Deskriptor definiert hier das Verhalten des EJB-Containers (zu Transaktionen siehe Kapitel 7).

5.8.5 Anwendungslogik und Home-Methoden

Die dritte Gruppe der Bean-Methoden sind die Methoden für die Anwendungslogik, die im Local- bzw. Remote-Interface definiert und in der Bean-Klasse implementiert werden. Die Definition in der Bean-Klasse hat die gleiche Signatur wie die Definition im Remote- bzw. Local-Interface. Die Implementierung ist äquivalent zu Entity-Beans mit Container-Managed-Persistence.

Entity-Beans mit Bean-Managed-Persistence können ebenfalls sogenannte Home-Methoden definieren. Hier gelten die gleichen Vorschriften und Verhaltensweisen wie für Entity-Beans mit Container-Managed-Persistence.

Auch bei einer Entity-Bean mit Bean-Managed-Persistence wird in den Anwendungsmethoden normalerweise nicht auf die Datenbank zugegriffen. Dies ist den Methoden für das Zustandsmanagement und für die Verwaltung der Identitäten vorbehalten.

5.8.6 Beispiel *Counter*

Am besten lassen sich die Unterschiede zwischen Bean-Managed- und Container-Managed-Persistence anhand eines Beispiels verdeutlichen. Dazu wollen wir das Beispiel *Counter* wieder aufgreifen und es diesmal als Entity-Bean mit Bean-Managed-Persistence implementieren. Die Counter-Bean wird (wie im Fall der Container-Managed-Persistence) den Remote-Cient-View unterstützen.

Listing 5.48 zeigt das Remote-Interface der Counter-Bean. Es ist identisch mit dem der Counter-Bean mit Container-Managed-Persistence (vgl. Listing 5.8). Auch die *CounterOverflowException* ist bereits bekannt, genauso wie das Datenbankschema (vgl. Listing 5.14).

```
package ejb.counterBmp;

import java.rmi.RemoteException;
import javax.ejb.EJBObject;

public interface Counter extends EJBObject {

    public void inc()
        throws RemoteException, CounterOverflowException;

    public void dec()
        throws RemoteException, CounterOverflowException;

    public int getValue()
        throws RemoteException;
}
```

Listing 5.48: Remote-Interface der Counter-Bean (BMP)

Listing 5.49 zeig das Home-Interface der Counter-Bean. Der einzige Unterschied zur Counter-Bean mit Container-Managed-Persistence ist, dass die Methode *getAllCounterIds* fehlt (vgl. Listing 5.10). Stattdessen gibt es jetzt die Methode *findAllCounters*. Die Home-Methode *getAllCounterIds* hat in der Implementierung eine Select-Methode benutzt, um alle Counter zu finden. Da es bei Bean-Managed-Persistence keine Select-Methoden gibt, wurde diese Funktionalität jetzt zur Veranschaulichung als Finder-Methode implementiert. Die Implementierung als Home-Methode wäre nach wie vor möglich gewesen. Die Home-Methode müsste den Datenbankzugriff implementieren, statt auf die Select-Methode zurückzugreifen.

```
package ejb.counterBmp;

import javax.ejb.CreateException;
import javax.ejb.EJBHome;
import javax.ejb.FinderException;
import java.rmi.RemoteException;

public interface CounterHome extends EJBHome {

    public Counter create(String counterId, int initCounterValue)
        throws CreateException, RemoteException;

    public Counter findByPrimaryKey(String primaryKey)
        throws FinderException, RemoteException;

    public java.util.Collection findAllCounters()
        throws FinderException, RemoteException;

}
```

Listing 5.49: Home-Interface der Counter-Bean (BMP)

Für den Client macht es im Übrigen keinen Unterschied, ob es sich um eine Entity-Bean mit Container-Managed- oder Bean-Managed-Persistence handelt. Das macht auch die Tatsache deutlich, dass das Home- und Remote-Interface im Wesentlichen unverändert geblieben ist.

Die Implementierung der Bean-Klasse hat sich dagegen entscheidend verändert. Listing 5.49 zeigt die Implementierung. Bereits bei den *import*-Statements wird deutlich, dass die Bean selbst auf die Datenbank zugreift (*java.sql.**).

In der Methode *ejbCreate* wird eine neue Counter-Bean in der Datenbank angelegt. Sie wird immer dann aufgerufen, wenn der Client einen neuen Counter erzeugen will und dazu im Home-Interface die Methode *create* aufruft. Neben der Initialisierung der persistenten Attribute (die diesmal *private* deklariert sind) muss die Bean dafür sorgen, dass der entsprechende Datensatz in der Datenbank angelegt wird. Das geschieht über die privaten Hilfsmethoden *initDataSource* und *create*. Eine weitere Besonderheit der

ejbCreate-Methode ist, dass nicht *null* zurückgegeben wird (wie bei Beans mit Container-Managed-Persistence), sondern die Primärschlüssel-Instanz der Bean. Die Implementierung der *ejbPostCreate*-Methode bleibt leer, da keine weiteren Initialisierungsschritte notwendig sind.

Als Nächstes folgt die Implementierung der beiden Finder-Methoden *findByPrimaryKey* (*ejbFindByPrimaryKey*) und *findAllCounters* (*ejbFindAllCounters*). Bei Container-Managed-Persistence werden die Finder-Methoden in der Bean-Klasse weder deklariert noch implementiert. Sie werden mit Hilfe der EJB-QL im Deployment-Deskriptor definiert. Die Methode *ejbFindByPrimaryKey* hat die Aufgabe zu verifizieren, ob ein Datensatz mit dem entsprechenden Primärschlüssel vorhanden ist. Wenn ja, wird der entsprechende Primärschlüssel zurückgegeben. Der EJB-Container initialisiert mit dieser Primärschlüssel-Instanz eine Counter-Bean im Zustand *Pooled* und gibt das Remote-Objekt dieser Bean an den Client als Rückgabewert für den *findByPrimaryKey*-Aufruf. Ist ein solcher Datensatz nicht vorhanden, wird eine Ausnahme ausgelöst. Gleiches gilt für die Methode *ejbFindAllCounters* mit dem Unterschied, dass eine leere Collection zurückgegeben wird, wenn keine Datensätze gefunden werden. Ein wichtiges Detail bei den Finder-Methoden ist der Aufruf der Hilfmethode *initDataSource* zu Beginn der jeweiligen Methode. Da die Finder-Methoden an der Bean-Instanz im Zustand *Pooled* aufgerufen werden, ist die Data-Source für Datenbankverbindungen womöglich nicht initialisiert.

```
package ejb.counterBmp;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.Statement;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.Collection;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

import javax.sql.DataSource;

import javax.ejb.CreateException;
import javax.ejb.EJBException;
import javax.ejb.EntityBean;
import javax.ejb.EntityContext;
import javax.ejb.FinderException;
import javax.ejb.ObjectNotFoundException;

public class CounterBean implements EntityBean {
```

```
public static final String dbRef =
    "java:comp/env/jdbc/CounterDB";

public final static int VALUE_MAX = 100;
public final static int VALUE_MIN = 0;

private EntityContext ctx;
private DataSource    dataSource;

private String counterId;
private int    counterValue;

 /*
  * Die Create Methode des Home-Interfaces
  */

public String ejbCreate(String counterId, int initCounterValue)
    throws CreateException
{
    if(counterId == null) {
        throw new CreateException("id must not be null!");
    }
    if(initCounterValue < VALUE_MIN ||
       initCounterValue > VALUE_MAX)
    {
        throw new CreateException("initValue out of range!");
    }

    this.initDataSource();

    this.counterId = counterId;
    this.counterValue = initCounterValue;

    try {
        this.create();
    } catch(SQLException ex) {
        throw new CreateException(ex.getMessage());
    }

    return this.counterId;
}

public void ejbPostCreate(String accountId,
                          int initCounterValue)
{}

 /*
  * Utility-Methoden
  */

private void initDataSource() {
```

```
        if(this.dataSource != null) {
            return;
        }
        try {
           Context c = new InitialContext();
           this.dataSource = (DataSource)c.lookup(dbRef);
        } catch(NamingException ex) {
           String msg = "Cannot get Resource-Factory:"
                      + ex.getMessage();
           throw new EJBException(msg);
        }
    }

    private void create()
        throws SQLException
    {
        final String query =
            "INSERT INTO COUNTER(ID, VALUE) VALUES(?, ?)";
        Connection        con = null;
        PreparedStatement st  = null;
        try {
            con = this.dataSource.getConnection();
            st  = con.prepareStatement(query);
            st.setString(1, this.counterId);
            st.setInt(2, this.counterValue);
            st.executeUpdate();
        } finally {
            try { st.close();  } catch(Exception ex) {}
            try { con.close(); } catch(Exception ex) {}
        }
    }

     /*
      * Die Finder-Methoden des Home-Interfaces
      */

    public String ejbFindByPrimaryKey(String pk)
        throws FinderException
    {
        this.initDataSource();

        final String query =
            "SELECT ID FROM COUNTER WHERE ID=?";
        Connection        con = null;
        PreparedStatement st  = null;
        ResultSet         rs  = null;
        try {
            con = this.dataSource.getConnection();
            st  = con.prepareStatement(query);
            st.setString(1, pk);
            rs = st.executeQuery();
```

```
            if(!rs.next()) {
                throw new ObjectNotFoundException(pk);
            }
        } catch(SQLException ex) {
            ex.printStackTrace();
            throw new FinderException(ex.getMessage());
        } finally {
            try { st.close();  } catch(Exception ex) {}
            try { rs.close();  } catch(Exception ex) {}
            try { con.close(); } catch(Exception ex) {}
        }
        return pk;
    }

    public Collection ejbFindAllCounters()
        throws FinderException
    {
        this.initDataSource();

        final String query =
            "SELECT ID FROM COUNTER";
        Connection con = null;
        Statement  st  = null;
        ResultSet  rs  = null;
        ArrayList  ret = new ArrayList();
        try {
            con = this.dataSource.getConnection();
            st  = con.createStatement();
            rs = st.executeQuery(query);
            while(rs.next()) {
                ret.add(rs.getString(1));
            }
        } catch(SQLException ex) {
            ex.printStackTrace();
            throw new FinderException(ex.getMessage());
        } finally {
            try { st.close();  } catch(Exception ex) {}
            try { rs.close();  } catch(Exception ex) {}
            try { con.close(); } catch(Exception ex) {}
        }
        return ret;
    }

     /*
      * Die Business-Methoden des Remote-Interfaces
      */

    public void inc() throws CounterOverflowException {
        if(this.counterValue < VALUE_MAX) {
            this.counterValue += 1;
        } else {
```

```
            String s = "Cannot increase above "+VALUE_MAX;
            throw new CounterOverflowException(s);
        }
    }

    public void dec() throws CounterOverflowException {
        if(this.counterValue > VALUE_MIN) {
            this.counterValue -= 1;
        } else {
            String s = "Cannot decrease below "+VALUE_MIN;
            throw new CounterOverflowException(s);
        }
    }

    public int getValue() {
        return this.counterValue;
    }

     /*
      * Die Methoden des Entity-Bean-Interfaces
      */

    public void ejbActivate() {
        this.initDataSource();
    }

    public void ejbPassivate() {
        this.dataSource = null;
    }

    public void setEntityContext(EntityContext ctx) {
        this.ctx = ctx;
    }

    public void unsetEntityContext() {
        this.ctx = null;
    }

    public void ejbLoad() {
        this.counterId = (String)this.ctx.getPrimaryKey();
        final String query =
            "SELECT VALUE FROM COUNTER WHERE ID=?";
        Connection        con = null;
        PreparedStatement st  = null;
        ResultSet         rs  = null;
        try {
            con = this.dataSource.getConnection();
            st  = con.prepareStatement(query);
            st.setString(1, this.counterId);
            rs = st.executeQuery();
            if(rs.next()) {
```

```
            this.counterValue = rs.getInt(1);
        } else {
            String s = this.counterId + " not found";
            throw new SQLException(s);
        }
    } catch(SQLException ex) {
        ex.printStackTrace();
        throw new EJBException(ex.getMessage());
    } finally {
        try { st.close();  } catch(Exception ex) {}
        try { rs.close();  } catch(Exception ex) {}
        try { con.close(); } catch(Exception ex) {}
    }
}

public void ejbStore() {
    final String query =
        "UPDATE COUNTER SET VALUE=? WHERE ID=?";
    Connection        con = null;
    PreparedStatement st  = null;
    try {
        con = this.dataSource.getConnection();
        st  = con.prepareStatement(query);
        st.setInt(1, this.counterValue);
        st.setString(2, this.counterId);
        st.executeUpdate();
    } catch(SQLException ex) {
        ex.printStackTrace();
        throw new EJBException(ex.getMessage());
    } finally {
        try { st.close();  } catch(Exception ex) {}
        try { con.close(); } catch(Exception ex) {}
    }
}

public void ejbRemove() {
    final String query =
        "DELETE FROM COUNTER WHERE ID=?";
    Connection        con = null;
    PreparedStatement st  = null;
    try {
        con = this.dataSource.getConnection();
        st  = con.prepareStatement(query);
        st.setString(1, this.counterId);
        st.executeUpdate();
    } catch(SQLException ex) {
        ex.printStackTrace();
        throw new EJBException(ex.getMessage());
    } finally {
        try { st.close();  } catch(Exception ex) {}
        try { con.close(); } catch(Exception ex) {}
```

```
            }
        }

    }
```

Listing 5.50: Bean-Klasse der Counter-Bean (BMP)

Die Implementierung der Business-Methoden *inc*, *dec* und *getValue* ist unverändert geblieben. Statt die abstrakten Persistenz-Methoden (die es ja im Fall der Bean-Managed-Persistence nicht gibt) aufzurufen, greifen die Methoden direkt auf die Member-Variablen der Bean-Klasse zurück.

Die Methode *ejbActivate* initialisiert die Bean-Instanz beim Zustandsübergang von *Pooled* nach *Ready* mit Hilfe der Hilfsmethode *initDataSource*. Die Methode *ejbPassivate* macht genau das Gegenteil.

Die Methode *ejbLoad* hat die Aufgabe, die Bean-Instanz mit den Daten aus der Datenbank zu synchronisieren. Der Primärschlüssel wird über den *EntityContext* initialisiert, die restlichen Attribute mit den Daten aus der Datenbank, die zu diesem Primärschlüssel passen. Es ist wichtig, das Primärschlüsselattribut aus dem *EntityContext* zu initialisieren, da die Entity-Bean-Instanz seit der letzten Synchronisation die Identität gewechselt haben könnte. Ein eventuell noch vorhandener Wert des Primärschlüsselattributs könnte ungültig sein.

Die Methode *ejbStore* hat die Aufgabe, die Daten der Datenbank mit den Daten aus der Bean-Instanz zu synchronisieren. Dazu werden die Daten in der Datenbank mit den Werten der persistenten Attribute überschrieben.

Die Methode *ejbRemove* hat die Aufgabe, den Datensatz, der durch die jeweilige Entity-Bean-Instanz repräsentiert wird, zu löschen. Die Entity-Bean-Instanz wechselt danach in den Zustand *Pooled* und muss darauf vorbereitet sein, mit einer anderen Identität wieder verwendet zu werden.

Listing 5.51 zeigt den Deployment-Deskriptor der Counter-Bean mit Bean-Managed-Persistence. Im Element *persistence-type* wird statt *Container* der Wert *Bean* angegeben. Eine Entity-Bean mit Bean-Managed-Persistence hat auch kein abstraktes Persistenzschema, das deklariert werden müsste. Stattdessen wird eine Resource-Factory-Referenz für den Zugriff auf die Datenbank definiert.

```
<?xml version="1.0" ?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0/
/EN" "http://java.sun.com/j2ee/dtds/ejb-jar_2_0.dtd">

<ejb-jar>

  <enterprise-beans>
    <entity>
```

```
        <ejb-name>CounterBmp</ejb-name>
        <home>ejb.counterBmp.CounterHome</home>
        <remote>ejb.counterBmp.Counter</remote>
        <ejb-class>ejb.counterBmp.CounterBean</ejb-class>
        <persistence-type>Bean</persistence-type>
        <prim-key-class>java.lang.String</prim-key-class>
        <reentrant>False</reentrant>
        <resource-ref>
          <description> Euro-Datenbank </description>
          <res-ref-name>jdbc/CounterDB</res-ref-name>
          <res-type>javax.sql.DataSource</res-type>
          <res-auth>Container</res-auth>
          <res-sharing-scope>Shareable</res-sharing-scope>
        </resource-ref>
      </entity>
    </enterprise-beans>

    <assembly-descriptor>
      <container-transaction>
        <method>
          <ejb-name>CounterBmp</ejb-name>
          <method-name>inc</method-name>
        </method>
        <method>
          <ejb-name>CounterBmp</ejb-name>
          <method-name>dec</method-name>
        </method>
        <trans-attribute>Required</trans-attribute>
      </container-transaction>
    </assembly-descriptor>

  </ejb-jar>
```

Listing 5.51: Deployment-Deskriptor der Counter-Bean (BMP)

Eine Beschreibung, wie die Abbildung der persistenten Attribute auf die Datenbank auszusehen hat, ist bei Bean-Managed-Persistence nicht erforderlich.

5.9 Zusammenfassung

Entity-Beans mit Container-Managed und Bean-Managed-Persistence haben ihre Vor- und Nachteile. Der große Vorteil der Container-Managed-Persistence ist sicher die Portabilität, da die wesentlichen Aspekte der Persistenz deklarativ in einer herstellerunabhängigen Art und Weise gehandhabt werden. Gerade diesbezüglich bietet die Version 2.0 gegenüber der Version 1.1 wesentliche Erweiterungen. Auch die persistenten Beziehungen, die vom EJB-Container gesteuert werden und bei der Abbildung komplexer Datenstrukturen eine sehr große Hilfe sind, verschaffen der Container-Managed-

Persistence weitere Vorteile. Zudem ist der Programmieraufwand wesentlich geringer. Entity-Beans mit Container-Managed-Persistence sind damit tendenziell weniger fehleranfällig.

Bean-Managed-Persistence bietet die größere Flexibilität. Der Bean-Entwickler hat freie Hand bei der Speicherung der Daten. Er entscheidet, wie und wo die Daten gespeichert werden und unterliegt keinen Einschränkungen von Seiten der Spezifikation. Das Optimierungspotential beim performanten Umgang mit dem Persistenzmedium ist bei Bean-Managed-Persistence wesentlich höher als bei Container-Managed-Persistence. Generik hat eben seinen Preis. Muss die Entity-Bean mit Datentypen umgehen, die der EJB-Container bzw. der Persistence-Mangager nicht oder nur mangelhaft unterstützen, führt ohnehin kein Weg an Bean-Managed-Persistence vorbei.

6 Message-Driven-Beans

Der Bean-Typ der Message-Driven-Bean (vgl. Abbildung 6.1) wurde mit Version 2.0 der Spezifikation der Enterprise JavaBeans eingeführt. Mit diesem Komponententyp wird der Java Message Service (JMS) untrennbar mit den Enterprise JavaBeans verknüpft.

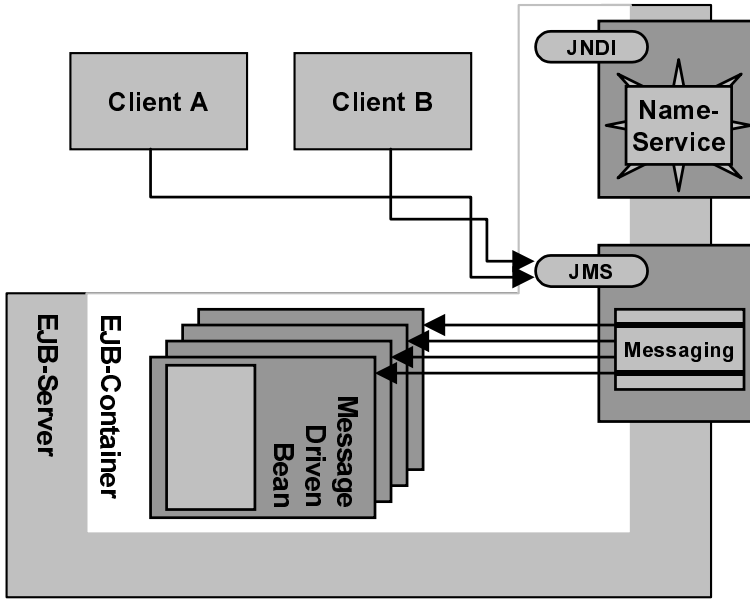

Abbildung 6.1: Überblick über Message-Driven-Beans

Der wesentliche Unterschied zu den anderen Bean-Typen besteht darin, dass ein Client eine Message-Driven-Bean nicht direkt ansprechen kann. Er kann sie nur indirekt ansprechen, indem er eine Nachricht über einen bestimmten Kanal des Messaging-Dienstes verschickt. Eine Message-Driven-Bean eines bestimmten Typs empfängt Nachrichten aus einem bestimmten Kanal des Messaging-Dienstes. Wesentlich ist auch, dass die Bean nicht weiß, welcher Client bzw. welcher Benutzer die Nachricht gesendet hat. Der Austausch von Nachrichten über einen Messaging-Dienst erfolgt

anonym. Die Applikation müsste selbst für eine Identifikationsmöglichkeit sorgen, sollte sie eine benötigen. Dazu könnte sie mit der Nachricht eine Kennung versenden, die die Identifikation des Senders ermöglicht.

Das Komponentenmodell der Enteprise JavaBeans gewinnt durch Message-Driven-Beans zwei neue Aspekte:

- parallele Verarbeitung,
- Asynchronität.

Ruft der Client z. B. eine Methode am Remote-Interface einer Entity-Bean auf, wird die Ausführung des Clients solange blockiert, bis die Verarbeitung der Methode der Entity-Bean am Server beendet wird. Es handelt sich um einen *synchronen* Aufruf. Das Versenden einer Nachricht über einen Messaging-Dienst ist *asynchron*. Das bedeutet, dass der Client die Verarbeitung fortsetzen kann, sobald er die Nachricht abgesendet hat. Die Verarbeitung des Clients wird nicht solange blockiert, bis die Nachricht zugestellt oder vom Empfänger verarbeitet wird. Message-Driven-Beans arbeiten demnach asynchron, was sie in diesem Punkt wesentlich von Session- oder Entity-Beans unterscheidet. Durch asynchrone Verarbeitung können die Antwortzeiten des Servers stark verkürzt werden. Der Client muss nicht auf die Beendigung arbeitsintensiver Aktionen warten. Sie können im Hintergrund ablaufen. Andererseits weiß der Client nicht automatisch, wann die Verarbeitung des asynchronen Vorgangs abgeschlossen ist, bzw. ob sie erfolgreich abgeschlossen werden konnte. Die JMS-Spezifikation trifft viele Vorkehrungen, um die Zustellung einer Nachricht so sicher wie möglich zu machen (in Bezug auf Datenverlust). Es gibt aber keinen Mechanismus, der die Benachrichtigung des Clients über die erfolgreiche Verarbeitung einer Nachricht vorsieht. Solche Mechanismen können jedoch mit den Mitteln von JMS selbst implementiert werden.

Um asynchrone Ereignisse zu verarbeiten, oder um Vorgänge in einem Programm zu beschleunigen, versucht man in der Regel, bestimmte Verarbeitungsschritte unter Zuhilfenahme mehrerer Threads zu parallelisieren. Die Spezifikation verbietet einer Bean ausdrücklich einen neuen Thread zu starten oder Threads zu steuern (vgl. auch Abschnitt 3.8). Man könnte versuchen, Vorgänge am Client zu parallelisieren, um mehrere Aufrufe an Enterprise-Beans gleichzeitig durchzuführen. Dies ist außer bei zustandslosen Session-Beans nur mit Einschränkungen möglich. Zustandsbehaftete Session-Beans erlauben es dem Client z. B. nicht, mehrere Aufrufe an einer Enterprise-Bean-Identität gleichzeitig durchzuführen. Bei einer Entity-Bean-Identität werden gleichzeitig eingehende Aufrufe (in unterschiedlichen Transaktionen) vom EJB-Container unter Umständen serialisiert, also hintereinander statt parallel abgearbeitet (abhängig von der Implementierung des EJB-Containers). Zudem würde die Komplexität des Clients stark zunehmen. Ein Grund für den Einsatz von Enterprise JavaBeans besteht nicht zuletzt in der Verlagerung der Komplexität auf den Applikationsserver und die Vereinfachung und Verschlankung des Clientprogramms. Der Applikations-

server ist der Ort, an dem man parallele Verarbeitung zur Beschleunigung der Anwendungslogik nutzen will. Message-Driven-Beans ermöglichen mit Hilfe des Messaging-Dienstes parallele Verarbeitung.

Damit Message-Driven-Beans richtig eingesetzt werden können, ist ein grundlegendes Verständnis des Java Message Service unbedingte Voraussetzung. Der nächste Abschnitt wird daher die Grundlagen des Java Message Service vorstellen. Wir beschränken uns auf die Teile, die für das Verständnis von Message-Driven-Beans von Bedeutung sind. Eine vollständige Beschreibung von JMS ist in [JMS, 1999] zu finden.

6.1 Java Message Service (JMS)

Messaging-Systeme sind auch unter dem Namen Message-Oriented-Middleware (MOM) bekannt. Sie unterscheiden sich von klassischen Client-Server-Architekturen in mehrerlei Hinsicht:

- Statt über hierarchische Strukturen sind die Clients eines Messaging-Dienstes als gleichberechtigte Teilnehmer organisiert. Clients können Anwendungsprogramme, Teile von Anwendungsprogrammen, Applikationsservern oder sonstige Prozesse sein.
- Die Clients sind lose über den Messaging-Dienst gekoppelt, d.h. sie können sich in beliebiger Anzahl zu beliebiger Zeit beim Messaging-Dienst an- bzw. abmelden.
- Der Nachrichtenaustausch erfolgt asynchron, d.h. sobald ein Client eine Nachricht an den Messaging-Dienst überstellt hat, kann er mit der Verarbeitung fortfahren und muss nicht warten, bis die Nachricht zugestellt oder verarbeitet ist.
- Der Nachrichtenaustausch erfolgt anonym, d.h. der Client, der eine Nachricht empfängt, weiß nicht, welcher Client diese Nachricht geschickt hat. Er weiß nur, über welchen Kanal des Messaging-Dienstes er sie empfangen hat.

JMS ist ein API (Application Programming Interface) der Firma Sun Microsystems. Sie ist eine herstellerunabhängige Fassade für den Zugriff aus Java-Programmen auf Messaging-Systeme. Verwendet ein Java-Programm JMS für den Zugriff auf ein Messaging-System, kann das verwendete Messaging-System ausgetauscht werden, ohne das Java-Programm wesentlich ändern zu müssen. Ist das Verhalten der Messaging-Systeme zur Laufzeit gleich (was durch die API allein nicht sichergestellt ist) und bieten sie den gleichen Funktionsumfang, muss das Clientprogramm in der Regel gar nicht geändert werden. Viele Messaging-Systeme sind außerdem neben JMS über eine proprietäre API des Herstellers zugänglich. Abbildung 6.2 verdeutlicht diese Zusammenhänge.

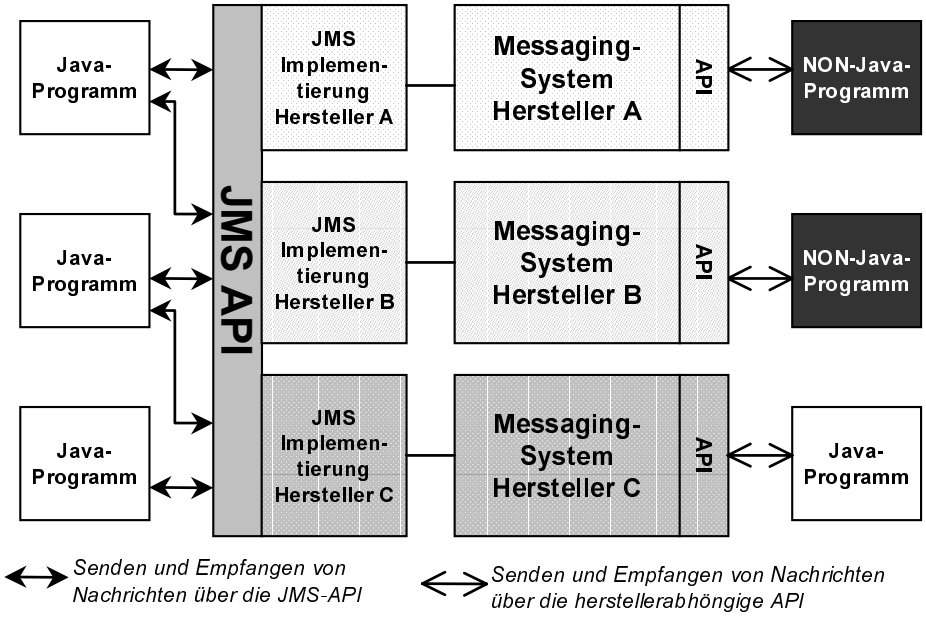

Abbildung 6.2: JMS und JMS-Provider

Sun Microsystems stellt keine Referenzimplementierung für JMS zur Verfügung. Sie bieten lediglich eine kommerzielle Implementierung des JMS an (Java Message Queue). Die meisten Hersteller von Messaging-Systemen bieten mittlerweile eine JMS-Implementierung an. Den Hersteller eines JMS-kompatiblen Messaging-Dienstes bezeichnet man als *JMS-Provider*.

6.1.1 Messaging-Konzepte

Eine Nachricht, die über einen Messaging-Dienst versendet wird, ist nicht direkt für einen Benutzer gedacht, wie es beispielsweise bei einer E-Mail der Fall ist. Nachrichten eines Messaging-Dienstes werden in der Regel in einem für Menschen unlesbaren Format übertragen. Sie dienen hauptsächlich zur Kommunikation zwischen Teilen einer bestimmten Anwendung oder zwischen verschiedenen Anwendungen. Messaging-Dienste werden häufig dazu benutzt, über eingehende Nachrichten bestimmte Aktionen beim Empfänger auszulösen. Des Weiteren benutzt man sie, um langdauernde Prozesse zu entkoppeln. Man verlagert zeitaufwändige Operationen in den Hintergrund, um das Antwortverhalten eines Systems zu verbessern. Messaging-Dienste eignen sich auch gut dazu, Aktionen gesammelt auszuführen. Eine Nachricht (Message) kann beispielsweise ein Auftrag für das Löschen eines Datensatzes in der Datenbank sein. Statt beim Eingang jeder Nachricht diesen Typs eine Datenbankverbindung zu

öffnen, können diese Aufträge gesammelt werden. Ist eine ausreichende Anzahl an Aufträgen aufgelaufen, wird die Verbindung zur Datenbank geöffnet und die Aktionen en bloc ausgeführt.

Ein Messaging-Dienst ist auf *asynchrone* Kommunikation zwischen *verschiedenen Prozessen* ausgerichtet. Events z. B. (wie sie im Abstract Windowing Toolkit oder beim JavaBeans-Komponentenmodell verwendet werden) dienen typischerweise zur prozessinternen Kommunikation. Ein Messaging-Dienst wird deswegen in erster Linie bei verteilten Anwendungen eingesetzt, bei denen Teile der Anwendung über Prozessgrenzen hinweg miteinander kommunizieren müssen. Messaging-Dienste werden auch für die Kommunikation zwischen verschiedenartigen Anwendungen eingesetzt.

Bei Messaging-Systemen unterscheidet man im Wesentlichen zwischen zwei Arten der Kommunikation:

- Point-to-Point,
- Publish-and-Subscribe.

Beim Point-to-Point-Modell wird eine Nachricht von einem Sender an genau *einen* Empfänger gesandt. Beim Publish-and-Subscribe-Modell wird eine Nachricht von einem Sender an *viele* Empfänger gesandt. Den Kanal des Messaging-Dienstes, über den eine Nachricht im Point-to-Point-Verfahren gesendet wird, bezeichnet man als *Queue*. Beim Publish-and-Subscribe-Verfahren bezeichnet man den Kanal als *Topic*. Umgekehrt kann man sagen: Eine Queue ist ein Nachrichtenkanal eines Messaging-Dienstes, bei dem eine gesendete Nachricht an genau einen Empfänger zugestellt wird. Ein Topic ist ein Nachrichtenkanal eines Messaging-Dienstes, bei dem eine gesendete Nachricht an mehrere Empfänger zugestellt wird (siehe Abbildung 6.3).

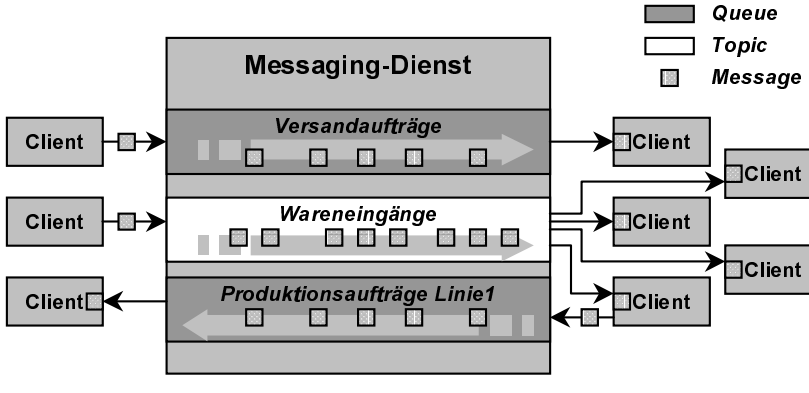

Abbildung 6.3: Messaging-Konzepte (Queue, Topic)

Durch Konfiguration können innerhalb eines Messaging-Dienstes beliebig viele Queues und Topics angelegt werden. Unterschieden werden sie in der Regel durch einen Namen.

Queues und Topics können persistent sein. Die Nachrichten, die sich gerade in einer Queue oder einem Topic befinden, werden beispielsweise in einer Datenbank oder im Dateisystem gespeichert. Sobald sie erfolgreich an den Empfänger-Client zugestellt sind, werden sie aus dem jeweiligen Speichermedium wieder gelöscht. Im Falle eines Serverabsturzes gehen bei persistenten Queues oder Topics die noch nicht zugestellten Nachrichten nicht verloren. Nach dem Neustart des Systems wird der Messaging-Dienst versuchen, die Nachrichten, die sich noch in einer persistenten Queue oder einem persistenten Topic befinden, zuzustellen.

Ob eine Queue oder ein Topic persistent ist, wird bei deren Konfiguration festgelegt. Handelt es sich um Queues oder Topics, über die wichtige Nachrichten versendet werden, so wird man diese in der Regel persistent machen. Bei weniger wichtigen Nachrichten kann darauf verzichtet werden. Die Speicherung jeder Nachricht wirkt sich negativ auf die Performanz des Systems aus.

Nicht alle Messaging-Systeme unterstützen beide Formen der Nachrichtenzustellung (Publish-and-Subscribe, Point-to-Point). Selbst wenn ein Messaging-Dienst nur eine der beiden Formen zur Verfügung stellt, ist eine Benutzung über JMS möglich. JMS modelliert beide Konzepte in getrennten Interfaces (vergleiche dazu den nächsten Abschnitt).

6.1.2 JMS-Interfaces

Dieser Abschnitt erläutert, wie JMS die Konzepte (d.h. die Bestandteile) eines Messaging-Systems, wie sie im vorangegangenen Abschnitt erläutert wurden, auf Interfaces abbildet. Die Verwendung dieser Interfaces wird im weiteren Verlauf dieses Kapitels anhand von Beispielen detailliert dargestellt.

javax.jms.ConnectionFactory

Eine Connection-Factory ist ein Objekt, mit dessen Hilfe ein JMS-Client eine Verbindung zum Messaging-Dienst aufbaut. Er bekommt ein solches Objekt über einen Namensdienst (JNDI-Lookup). Je nachdem ob eine Queue oder ein Topic benutzt werden soll, ist dieses Interface weiter zu *javax.jms.QueueConnectionFactory* und *javax.jms.TopicConnectionFactory* spezialisiert. Laut JMS-Spezifikation ist es die Aufgabe des Administrators, Connection-Factories zu konfigurieren. Man nennt sie deshalb auch *Administered Objects*. Die JMS-Implementierung stellt diese dann beim Start des Messaging-Dienstes im Namensdienst für die Clients zur Verfügung.

javax.jms.Connection

Ein Connection-Objekt repräsentiert eine Verbindung zu einem Messaging-Dienst. Sie wird über ein *ConnectionFactory*-Objekt erzeugt. Je nachdem ob eine Queue oder Topic benutzt wird, handelt es sich konkret um Objekte vom Typ *javax.jms.QueueConnection* oder *javax.jms.TopicConnection*.

javax.jms.Session

Eine Session ist ein Objekt, das über eine Verbindung zum JMS an eine bestimmte Queue oder an einen bestimmten Topic gebunden ist. Eine Session wird über ein Connection-Objekt erzeugt. Über eine Session werden Nachrichten gesendet und empfangen. Je nachdem ob die Session an eine Queue oder einen Topic gebunden ist, handelt es sich konkret um eine *javax.jms.QueueSession* oder eine *javax.jms.TopicSession*.

javax.jms.Destination

Destination ist der Oberbegriff für Topic oder Queue. Dieses Interface ist demnach weiter spezialisiert in *javax.jms.Queue* und *javax.jms.Topic*. Ein Objekt vom Typ Queue oder Topic kann über einen JNDI-Lookup bezogen werden. Auch hier sieht die JMS-Spezifikation vor, dass sie vom Administrator per Konfiguration angelegt werden. Auch eine Queue oder ein Topic wird deswegen *Administered Object* genannt. Die JMS-Implementierung stellt sie den JMS-Clients beim Start des Messaging-Dienstes über JNDI zur Verfügung. Über eine Connection-Factory kann dann eine Verbindung bzw. eine Session zu einer Queue oder einem Topic geöffnet werden, um Nachrichten zu senden oder zu empfangen.

javax.jms.Message

Dieses Interface repräsentiert die Nachricht, die über eine Queue oder einen Topic geschickt wird. Eine Message hat folgende Bestandteile:

- *Header*: Im Header einer Message werden vom JMS-Client als auch vom Provider Informationen gespeichert, die zur Identifikation und zur Zustellung einer Nachricht an den Empfänger dienen. Nachrichten-Filter (siehe unten) können auf Header-Felder angewendet werden.
- *Properties*: Sie geben einer Anwendung, die einen Messaging-Dienst benutzt, die Möglichkeit, anwendungsbezogene, zusätzliche Informationen in der Nachricht zu speichern. Solche Informationen geben dem Empfänger der Nachricht in der Regel Hinweise oder Anweisungen, die die Verarbeitung der Nachricht betreffen. Beispielsweise könnten Informationen über den Absender der Nachricht in einem Pro-

perty-Feld übertragen werden, so dass der Empfänger weiß, von wem die Nachricht gesendet wurde. Nachrichtenfilter (siehe unten) können auf Property-Felder angewendet werden.

- *Body*: Der Body beinhaltet die eigentlichen Informationen der Nachricht.

JMS definiert fünf konkrete Formen einer Message. Sie sind von *javax.jms.Message* abgeleitet und repräsentieren unterschiedliche technische Darstellungen der Nutzdaten einer Nachricht (Message-Body).

- *javax.jms.TextMessage*: Nachricht, die einen String als Message-Body beinhaltet. Über diesen Nachrichtentyp können beispielsweise auch Nachrichten im XML-Format ausgetauscht werden. Manche Messaging-Systeme bieten umfangreiche Unterstützung für das XML-Format an.
- *javax.jms.ObjectMessage*: Nachricht, die ein beliebiges Objekt als Message-Body beinhaltet. Die Klasse des Objekts muss das Interface *java.io.Serializable* implementieren.
- *javax.jms.BytesMessage*: Nachricht, die den Message-Body in Form eines Byte-Streams beinhaltet. Die einzelnen Bytes werden nicht interpretiert. Es bleibt dem Empfänger der Nachricht überlassen, den Inhalt der Nachricht entsprechend zu interpretieren.
- *javax.jms.StreamMessage*: Nachricht, mit der ein Stream von primitiven Java-Datentypen und serialisierbaren Objekten versendet werden kann. Die Primitiven und die Objekte werden sequenziell in die Nachricht geschrieben und müssen beim Empfänger in derselben Reihenfolge wieder gelesen werden. Der Empfänger muss demnach die Reihenfolge und den Typ der übermittelten Daten kennen. Das Senden einer Stream-Message entspricht einer angepassten Serialisierung/Deserialisierung.
- *javax.jms.MapMessage*: Nachricht, mit der Name-Wert-Paare als Message-Body versendet werden können. Die Namen werden in Form von Strings, die Werte als primitive Datentypen übertragen.

Jede Anwendung sollte in der Lage sein, einen für ihre Zwecke geeigneten Nachrichtentyp zu finden. Gerade so allgemeine Typen wie *javax.jms.ObjectMessage* oder *java.jms.StreamMessage* bieten dem Entwickler viele Freiräume.

javax.jms.MessageProducer

Ein Message-Producer ist ein Objekt, mit dessen Hilfe Nachrichten gesendet werden können. Je nach Destination-Typ unterscheidet man hier die konkreten Ausprägungen *javax.jms.QueueSender* bzw. *javax.jms.TopicPublisher*. Ein Message-Producer wird über ein Session-Objekt erzeugt.

javax.jms.MessageConsumer

Ein Message-Consumer ist ein Objekt, mit dessen Hilfe Nachrichten empfangen werden können. Je nach Destination-Typ unterscheidet man hier die konkreten Ausprägungen *javax.jms.QueueReciever* bzw. *java.jms.TopicSubscriber*. Ein Message-Consumer wird über ein Session-Objekt erzeugt.

Abbildung 6.4 stellt die Zusammenhänge zwischen den zentralen Interfaces der JMS-API nochmals grafisch dar.

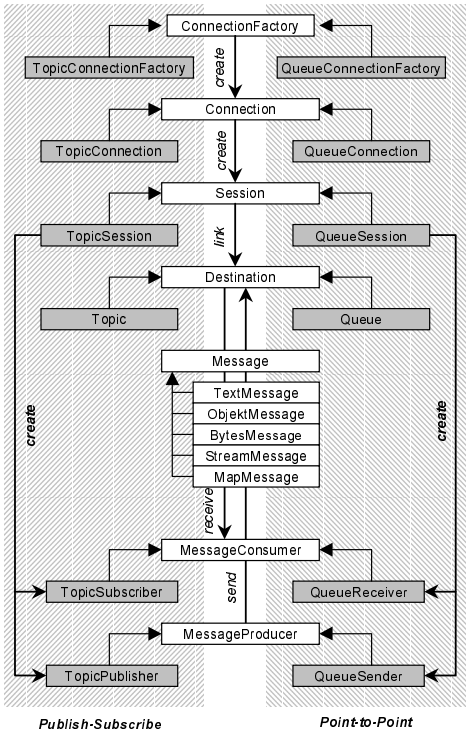

Abbildung 6.4: JMS-Interfaces im Überblick

6.1.3 JMS-Clients

Ein JMS-Client benutzt die Interfaces des Java Message Service zum Senden *und* Empfangen von Nachrichten über einen Messaging-Dienst. Ein JMS-Client kann Nachrichten senden oder empfangen oder beides. Die Implementierung der JMS-Interfaces stellt der Provider des Messaging-Dienstes zur Verfügung. Neben den Interfaces des JMS benutzt der Client auch die Interfaces des JNDI (Java Naming and Directory Service). Die Verwendung von JNDI wurde bereits ausführlich in Kapitel 4, Session-Beans, dargestellt. Über JNDI gelangt der JMS-Client an JMS-Ressourcen wie Queues,

Topics oder Connection-Factories (Administered Objects). Diese JMS-Ressourcen werden (wie oben bereits erwähnt) vom Administrator des Messaging-Dienstes konfiguriert. Das Messaging-System stellt diese Ressourcen den JMS-Clients zur Laufzeit über einen Naming- und Directory-Service zur Verfügung.

Die folgenden Abschnitte zeigen die Implementierung der einzelnen Aspekte eines JMS-Clients. Alle Beispiele verwenden eine Hilfsklasse namens *Lookup*, die die Methode *get* hat. In dieser Hilfsklasse und der Methode *get* ist der Zugriff auf den Namensdienst gekapselt, um den Beispielcode auf das Wesentliche zu beschränken. Die Implementierung dieser Hilfsklasse ist Bestandteil des Beispiel-Quellcodes zu diesem Buch, der von angegebener Stelle bezogen werden kann.

6.1.4 Senden einer Nachricht

Single-Threaded

Listing 6.1 zeigt das Senden einer Textnachricht. Das Beispiel zeigt die Benutzung einer Queue. Die Verwendung eines Topics ist in der Abfolge der Schritte identisch. Statt der queue-spezifischen Interfaces kommen die topic-bezogenen Interfaces zum Einsatz (z.B *TopicConnection* statt *QueueConnection*, *TopicSession* statt *QueueSession* etc.).

```
…
    static final String FACTORY =
                    "SampleConnectionFactory";
    static final String QUEUE = "SampleQueue";
    QueueConnection qc    = null;
    QueueSession    qs    = null;
    QueueSender     qsend = null;
    try {
        QueueConnectionFactory qcf = null;
        qcf = (QueueConnectionFactory)
                  Lookup.get(FACTORY);
        qc = qcf.createQueueConnection();
        qs = qc.createQueueSession(false,
                                   Session.AUTO_ACKNOWLEDGE);
        Queue queue = (Queue)Lookup.get(QUEUE);
        qsend = qs.createSender(queue);
        Message m = null;
        m = qs.createTextMessage("a text");
        qsend.send(m);
    } finally {
        try  { qsend.close(); } catch(Exception ex) {}
        try  { qs.close(); }    catch(Exception ex) {}
        try  { qc.close(); }    catch(Exception ex) {}
    }
…
```

Listing 6.1: Senden einer Textnachricht

Im ersten Schritt holt sich der JMS-Client mit Hilfe der Klasse *Lookup* ein *QueueConnectionFactory*-Objekt aus dem JNDI. Der Administrator des Messaging-Dienstes stellt diese durch Konfiguration zur Verfügung. Über die Queue-Connection-Factory erzeugt der JMS-Client im nächsten Schritt eine *QueueConnection*, mit deren Hilfe er dann eine *QueueSession* erzeugt. Die *Queue*, über die er die Nachricht schicken will, holt er (wie die Queue-Connection-Factory) aus dem JNDI. Auch die Queue wurde vom Administrator des Messaging-Dienstes durch Konfiguration eingerichtet. Mit Hilfe des *QueueSession*-Objekts erzeugt der JMS-Client für diese Queue einen *QueueSender*. Das *Message*-Objekt wird ebenfalls über die Session erzeugt. In unserem Fall handelt es sich um ein Objekt vom Typ *TextMessage*. Im letzten Schritt sendet der JMS-Client die Nachricht über den Queue-Sender. Unmittelbar nach dem Einstellen der Nachricht in die Queue kehrt die Methode *send* zurück und der JMS-Client kann seine Verarbeitung fortsetzen.

Eine JMS-Connection und eine JMS-Session beanspruchen in der Regel Ressourcen wie z.B. eine Netzwerkverbindung oder einen Thread. Es empfielt sich daher, diese Ressourcen durch Aufruf der diversen *close*-Methoden wieder freizugeben. Im Beispiel wird die Session und die Verbindung sofort wieder geschlossen. Versendet der Client regelmäßig Nachrichten, bietet es sich an, die Verbindung und die Session offen zu halten und erst dann zu schließen, wenn das Clientprogramm beendet wird.

Beim Senden einer Nachricht kann der JMS-Client der Nachricht eine Priorität geben (über die Methode *Message.setJMSPriority)*. Nachrichten mit hoher Priorität werden vor den Nachrichten mit niedriger Priorität zugestellt. Die Priorität kann bei der *send*-Methode mit übergeben werden (die *send*-Methode besitzt verschiedene Signaturen). Wird die Priorität nicht explizit angegeben, wird der Nachricht eine Standardpriorität zugeteilt.

Multi-Threaded

Eine JMS-Session ist nicht thread-sicher. Das heißt, eine Session kann nicht von mehreren Threads gleichzeitig benutzt werden. Die JMS-Spezifikation weist ausdrücklich auf diesen Umstand hin (vgl. [JMS, 1999]). Einer der Hauptgründe dafür ist, dass die Session diejenige Einheit im Messaging-System ist, die Transaktionen unterstützt (sofern das Messaging-System an sich Transaktionen unterstützt). Es ist sehr schwer, Transaktionen zu implementieren, die multithreaded sind. Wird eine Session dennoch von mehreren Threads gleichzeitig benutzt, ist das Verhalten des JMS-Providers ungewiss und es ist mehr als wahrscheinlich, dass schwere Fehler auftreten.

Die Tatsache, dass eine JMS-Session nicht thread-sicher ist, ist insbesondere bei den Anwendungen kritisch, die viele Threads benutzen. Ein typisches Beispiel dafür sind Internetanwendungen. Sie werden von vielen Benutzern gleichzeitig benutzt. Innerhalb des Web-Container-Prozesses ist damit mindestens ein Thread pro parallelem

Zugriff aktiv. Die Art und Weise des Versendens einer Nachricht, wie sie Listing 6.1 zeigt, ist insbesondere im Fall einer Webanwendung wenig effektiv (ausgehend davon, dass regelmäßig Nachrichten versendet werden). Sind viele Benutzer und damit viele Threads gleichzeitig aktiv, werden viele JMS-Verbindungen und JMS-Sessions permanent geöffnet und wieder geschlossen. Die JMS-Session offen zu halten und erst bei Beendigung des Programms zu schließen, würde dazu führen, dass eine JMS-Session von mehreren Threads gleichzeitig benutzt wird. Solche Probleme werden in der Regel durch die Verwendung von thread-sicheren Pools umgangen. Listing 6.2 zeigt an einem einfachen Beispiel, wie ein solcher Ressourcenpool implementiert werden kann.

```
package jms.client;

import java.util.LinkedList;
import javax.jms.Session;
import javax.jms.JMSException;

public class SessionPool {

    private LinkedList sessions;
    private int        size;

    public SessionPool(int size) {
        sessions  = new LinkedList();
        this.size = size;
    }

    public synchronized void addSession(Session s) {
        if(!sessions.contains(s)) {
            sessions.addLast(s);
        } else {
            throw new IllegalArgumentException
                              ("session already in use!");
        }
    }

    public synchronized Session getSession() {
        Session ret = null;
        while(sessions.isEmpty()) {
            try {
                this.wait();
            } catch(InterruptedException ex) {}
        }
        ret = (Session)sessions.removeFirst();
        return ret;
    }

    public int getAvailable() {
        return sessions.size();
    }
```

```
    public synchronized void releaseSession(Session s) {
        if(sessions.size() >= size) {
            throw new IllegalStateException
                        ("pool is exceeding initial size");
        }
        if(sessions.contains(s)) {
            throw new IllegalArgumentException
                        ("session is available");
        }
        sessions.addLast(s);
        this.notify();
    }

    public synchronized void destroy() {
        for(int i = 0; i < sessions.size(); i++) {
            try {
                ((Session)sessions.get(i)).close();
            } catch(JMSException jmsex) {
                jmsex.printStackTrace();
            }
        }
        sessions.clear();
        sessions = null;
    }
}
```

Listing 6.2: Beispiel eines Session-Pools

Durch die Verwendung eines Session-Pools wird das permanente Erzeugen und Schließen von Sessions vermieden. Außerdem sind durch den Pool weniger JMS-Sessions als aktive Threads notwendig. Zudem verhindert der Session-Pool, dass eine JMS-Session von mehreren Threads gleichzeitig verwendet wird. Eine verfügbare Session wird über die Methode *getSession* angefordert. Falls alle verfügbaren Sessions von anderen Threads belegt sind, blockiert die Methode durch den Auruf der Methode *wait*. Wenn ein Session-Objekt nicht mehr benötigt wird, muss es über die Methode *releaseSession* wieder freigegeben werden. Durch den Aufruf der Methode *notify* beim Freigeben einer JMS-Session wird einer der an der Methode *wait* blockierten Threads wieder freigegeben. Er kann seine Verarbeitung fortsetzen und ist nun im Besitz eines JMS-Session-Objekts. Wenn permanent wesentlich mehr Threads aktiv sind, als JMS-Sessions zur Verfügung stehen, kann sich der Pool zum Engpass entwickeln, da ständig Threads blockiert werden. Das Problem kann behoben werden, indem die Anzahl der verfügbaren JMS-Sessions im Pool erhöht wird. Listing 6.3 zeigt die Verwendung des Session-Pools in einer beliebigen Klasse.

```
…
private QueueConnection queueCon = null;
private SessionPool     thePool  = null;
…
```

```
public void init() {
  …
  //Erzeugen des SessionPools, z.B. zum Zeitpunkt der
  //Initialisierung
  final int SIZE = 10;
  QueueConnectionFactory qcf;
  qcf = (QueueConnectionFactory)
            Lookup.get("ANY_FACTORY");
  Queue q = (Queue)Lookup.get("ANY_QUEUE");
  QueueConnection qc = qcf.createQueueConnection();
  SessionPool sp  = new SessionPool(SIZE);
  for(int i = 0; i < SIZE; i++) {
      sp.addSession(qc.createQueueSession
                    (false, Session.AUTO_ACKNOWLEDGE));
  }
  queueCon = qc;
  thePool  = sp;
  …
}

public void send(String message)
    throws NamingException, JMSException
{
  QueueSession qs    = null;
  QueueSender  qsend = null;
  try {
    qs = (QueueSession)thePool.getSession();
    qsend = qs.createSender(theQueue);
    TextMessage tm = qs.createTextMessage();
    tm.setText(message);
    qsend.send(tm);
  } finally {
      try  { qsend.close(); } catch(Exception ex) {}
      thePool.releaseSession(qs);
  }
}
…
public void shutdown()
    throws JMSException
{
  …
  //Zerstören des Session-Pools und Schliessen
  //der Sessions, wenn der Prozess beendet wird
  thePool.destroy();
  queueCon.close();
  …
}
…
```

Listing 6.3: Verwendung des Session-Pools

Verwendet eine Anwendung den in diesem Abschnitt vorgestellten Session-Pool, muss sie pro Queue oder Topic einen solchen Pool zur Verfügung stellen. Damit nicht zu viele verschiedene Pools benutzt werden müssen, bietet es sich an, »generische« Queues oder Topics zu benutzen. Man sendet verschiedene Nachrichtentypen über eine Queue oder einen Topic, die dann auf der Empfängerseite verschiedene Aktionen auslösen, statt für jeden Typ einer Aktion eine eigene Queue oder einen eigenen Topic einzurichten. Es ist in jedem Fall wichtig, eine Session umgehend nach der Benutzung wieder freizugeben. Je länger ein Thread ein Session-Objekt blockiert, desto größer ist die Wahrscheinlichkeit, dass deswegen ein anderer Thread blockiert wird und auf eine verfügbare Session warten muss. Im Beispiel geschieht das Anfordern einer Session in einem *try*-Block und das Freigeben der Session in einem angeschlossenen *finally*-Block. Ohne die Verwendung des Try-Finally-Mechanismus hätte das Auftreten einer unerwarteten Ausnahme zur Folge, dass die Session nicht mehr freigegeben wird. Damit würde sich der Pool von Session-Instanzen ungewollt verkleinern. Das Session-Objekt ginge verloren und die von ihm belegten Ressourcen könnten nicht mehr ordnungsgemäß freigegeben werden.

6.1.5 Empfangen einer Nachricht

Beim Empfangen von Nachrichten hat der JMS-Client mehrere Möglichkeiten, was die Art des Empfangs angeht. Er kann sich Nachrichten vom Messaging-Dienst abholen oder sich die Nachrichten vom Messaging-Dienst quasi bringen lassen. Für die Beispiele in diesem Abschnitt wird erneut das Point-to-Point-Modell zur Veranschaulichung verwendet. Die Verwendung des Publish-Subscribe-Modells ist in der Abfolge der Schritte identisch. Statt der queue-spezifischen Interfaces kommen die topic-bezogenen Interfaces zum Einsatz. Wir werden an den entsprechenden Stellen auf die Unterschiede im Verhalten des Messaging-Dienstes hinweisen.

Die Methode receive

Die Methode *receive()* am Interface *QueueReceiver* kann dazu benutzt werden, Nachrichten beim Messaging-Dienst abzuholen. Listing 6.4 zeigt, wie die Methode angewendet wird.

```
…
    static final String FACTORY =
                    "SampleConnectionFactory";
    static final String QUEUE = "SampleQueue";
    QueueConnectionFactory qcf = null;
    QueueConnection        qc  = null;
    QueueSession           qs  = null;
    QueueReceiver          qr  = null;
    try {
        qcf = (QueueConnectionFactory)
```

```
                Lookup.get(FACTORY);
        qc = qcf.createQueueConnection();
        qs = qc.createQueueSession(false,
                                   Session.AUTO_ACKNOWLEDGE);
        Queue queue = (Queue)Lookup.get(QUEUE);
        qr = qs.createReceiver(queue);
        qc.start();
        while(true) {
            Message m = qr.receive();
            //…
            //Verarbeiten der Nachrichten
            //…
            m.acknowledge();
        }
    } finally {
        try { qr.close(); } catch(Exception ex) {}
        try { qs.close(); } catch(Exception ex) {}
        try { qc.close(); } catch(Exception ex) {}
    }
…
```

Listing 6.4: Abholen einer Nachricht

Die ersten Schritte beim Empfangen einer Nachricht sind identisch mit denen beim Senden einer Nachricht. Statt eines Queue-Senders erzeugt der JMS-Client nun ein *QueueReciever*-Objekt. Mit dem Aufruf der *start*-Methode am *QueueConnection*-Objekt veranlasst der JMS-Client den Messaging-Dienst, ab sofort mit der Zustellung von Nachrichten zu beginnen. Der Aufruf der *receive*-Methode liefert die nächste anstehende Nachricht zurück. Ist momentan keine Nachricht verfügbar, dann blockiert diese Methode den Aufrufer solange, bis die nächste Nachricht verfügbar ist. Auf die *acknowledge*-Methode kommen wir im Detail noch weiter unten zu sprechen.

Die Methode *receive()* steht in zwei weiteren Ausprägungen zur Verfügung. Sie kann einen Parameter *timeout* (Zeitintervall in Millisekunden) entgegennehmen. Steht keine Nachricht für den JMS-Client zur Verfügung, blockiert auch diese Methode solange, bis eine Nachricht verfügbar ist. Ist innerhalb des angegebenen Zeitintervalls *Timeout* keine Nachricht verfügbar, wird die Blockade aufgehoben. Die Methode kehrt zurück und liefert statt einer Nachricht *null* zurück. Die Methode *receiveNoWait* kehrt in jedem Fall sofort zum Aufrufer zurück. War eine Nachricht verfügbar, wird diese zurückgegeben. War keine Nachricht verfügbar, wird *null* zurückgegeben.

Auch im Fall des Empfangens von Nachrichten gilt, dass eine JMS-Connection und eine JMS-Session einige Ressourcen beanspruchen. Gibt der (empfangende) JMS-Client diese Resourcen durch Aufruf der jeweiligen close-Methoden wieder frei, endet der Empfang von Nachrichten.

Das Interface MessageListener

Anders als bei der *receive*-Methode, bei der der JMS-Client von sich aus die Nachrichten beim Messaging-Dienst abholt, kann er sich die Nachrichten auch zustellen lassen, sobald sie verfügbar sind. Sie werden ihm asynchron über den Thread der Session zugestellt. Der JMS-Client muss dazu ein Objekt, welches das *javax.jms.MessageListener*-Interface implementiert, bei einem *QueueReceiver*-Objekt registrieren. Listing 6.5 zeigt die Anwendung dieses Mechanismus.

```
…
    static final String FACTORY =
                   "SampleConnectionFactory";
    static final String QUEUE = "SampleQueue";
    QueueConnectionFactory qcf = null;
    QueueConnection        qc  = null;
    QueueSession           qs  = null;
    QueueReceiver          qr  = null;
    qcf = (QueueConnectionFactory)
              Lookup.get(FACTORY);
    qc = qcf.createQueueConnection();
    qs = qc.createQueueSession(false,
                               Session.AUTO_ACKNOWLEDGE);
    Queue queue = (Queue)Lookup.get(QUEUE);
    qr = qs.createReceiver(queue);
    qr.setMessageListener(this);
    qc.start();
…
```

Listing 6.5: Empfangen einer Nachricht

Um den Empfang von Nachrichten in Gang zu setzen, wird auch in diesem Fall die Methode *start()* am *QueueConnection*-Objekt aufgerufen. Der Aufruf der *setMessageListener*-Methode kehrt sofort zurück und der JMS-Client setzt die Verarbeitung fort. Über diese Methode registriert der JMS-Client das Objekt, welches das *javax.jms.MessageListener*-Interface implementiert (im Beispiel *this*). Dieses Interface beinhaltet nur die Methode *onMessage(Message)* (siehe Listing 6.6). Wenn dem JMS-Client eine Nachricht zugestellt werden soll, wird diese Methode aufgerufen und die Nachricht in Form eines *Message*-Objekts übergeben.

```
…
  public void onMessage(javax.jms.Message m) {
      // ...
      // Verarbeiten der Nachricht
      // ...
      try {
          m.acknowledge();
      } catch(javax.jms.JMSException jmsex) {
```

```
            jmsex.printStackTrace();
        }
    }
...
```

Listing 6.6: Implementierung MessageListener.onMessage()

Innerhalb einer Session wird das Zustellen von Nachrichten an einen Empfänger serialisiert. Eine Session benutzt immer nur einen Thread. Erst wenn die Zustellung einer Nachricht an den bzw. die Empfänger abgeschlossen ist, wird die nächste Nachricht zugestellt. Um die Verarbeitung von Nachrichten zu parallelisieren, kann der JMS-Client entweder selbst innerhalb der *onMessage*-Methode mit mehreren Threads arbeiten oder er kann mehrere Sessions verwenden. So überlässt er den Umgang mit Threads der JMS-Implementierung des Messaging-Dienstes.

Beim asynchronen Empfangen von Nachrichten ist das Verhalten des Messaging-Dienstes unterschiedlich, je nachdem ob eine Queue oder ein Topic verwendet wird. Wenn ein JMS-Client Nachrichten von einer Queue empfängt und dafür mehrere Sessions benutzt, dann ist es von der Implementierung des JMS-Providers abhängig, welcher Client die nächste Nachricht erhält. Sicher ist jedoch, dass nur ein Client die Nachricht bekommt. Es gibt JMS-Provider, bei denen nur eine Session und ein Empfänger je Queue erlaubt sind. Beim Publish-Subscribe-Modell bekommt jeder Empfänger jeder Session jede Nachricht zugestellt. Abbildung 6.5 verdeutlicht diesen Sachverhalt.

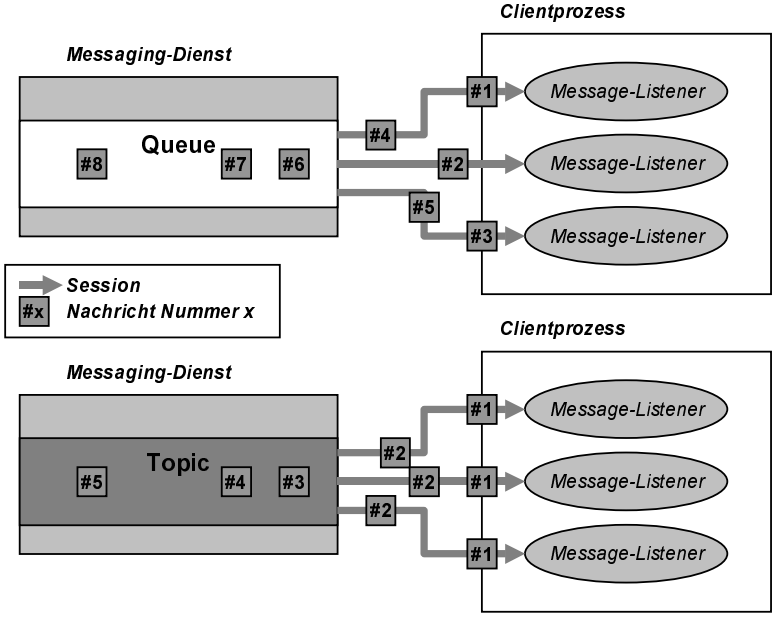

Abbildung 6.5: Empfangen von Nachrichten über mehrere Sessions

Wenn der JMS-Client mehrere Sessions benutzt, um Nachrichten parallel zu empfangen, und bei jeder Session das gleiche *MessageListener*-Objekt registriert, dann muss die Implementierung der *onMessage*-Methode thread-sicher sein. Da jede Session einen Thread für die Zustellung verwendet, kann die *onMessage*-Methode deswegen von mehreren Threads gleichzeitig durchlaufen werden. Registriert der JMS-Client bei jeder Session ein anderes *MessageListener*-Objekt, so muss die Implementierung nicht thread-sicher sein. Der JMS-Client muss sich aber in diesem Fall unter Umständen um die Synchronisation der parallel eingehenden Nachrichten kümmern.

Der Server-Session-Pool

Eine andere Art des Nachrichtenempfangs wird von der JMS-Spezifikation in einem Abschnitt behandelt, der sich ausdrücklich weiterführenden Themen widmet, die nicht zum Standardfunktionsumfang eines Messaging-Dienstes gehören. Der Abschnitt *JMS Application Server Facilities* (vgl. [JMS, 1999]) behandelt Sachverhalte, die an die Anbieter von Applikationsservern gerichtet sind. Dazu gehört der *Server-Session-Pool*, der das parallele Empfangen mehrerer Nachrichten ermöglicht.

Statt selber mehrere Sessions zu öffnen und zu verwalten, kann der JMS-Client diese Aufgabe dem Applikationsserver überlassen, indem er einen Server-Session-Pool benutzt. Unter einem Applikationsserver versteht man im Java-Umfeld typischerweise ein Produkt, das konform zur Spezifikation von Java 2, Enterprise Edition, ist (d.h. ein Server, der neben einem Web- und EJB-Container auch einen Messaging-Dienst und eine JMS-Implementierung anbietet). Es ist auch möglich, dass der JMS-Provider selbst eine *Server-Session-Pool*-Implementierung zu Verfügung stellt. Im weiteren Verlauf dieses Abschnitts werden wir den Begriff Applikationsserver synonym für beide Fälle verwenden.

Eine Server-Session ist ein Objekt, das der Applikationsserver zur Verfügung stellt. Eine Server-Session verbindet eine JMS-Session mit einem Thread des Applikationsservers. Der Messaging-Dienst ist verantwortlich dafür, die Nachrichten über die Session zuzustellen. Der Applikationsserver verwaltet die Session und stellt für die JMS-Session einen Thread zur Verfügung. Der JMS-Client stellt lediglich die Implementierung des *MessageListener*-Interface in einer eigenen Klasse zur Verfügung, welche für die Verarbeitung der Nachrichten zuständig ist. Da jedem *MessageListener*-Objekt ein eigener Thread zur Verfügung gestellt wird, muss die Implementierung nicht thread-sicher sein.

Die Verarbeitung der eingegangenen Nachrichten findet in diesem Fall nicht wie bei den beiden anderen Fällen im Prozess des JMS-Clients statt, sondern im Prozess des Applikationsservers. Will oder muss der JMS-Client aus Performancegründen Nachrichten parallel verarbeiten, ist die Verwendung eines Server-Session-Pools eine große Erleichterung. Er muss sich weder um die Verwaltung mehrerer JMS-Sessions und

mehrerer Threads kümmern noch wird zur Laufzeit der Client-Prozess durch die Belegung diverser Ressourcen wie Netzwerkverbindungen und Threads belastet. Allerdings werden die Ressourcen des Applikationsservers zusätzlich durch einen Server-Session-Pool belastet. Vor der Benutzung von Server-Session-Pools sollte auf jeden Fall in Erwägung gezogen werden, ob der Applikationsserver die zusätzliche Belastung verkraften kann.

Listing 6.7 zeigt an einem Beispiel die Verwendung eines Server-Session-Pools, Listing 6.8 zeigt die Implementierung des *MessageListener*-Interface, die im Server-Session-Pool für die Verarbeitung der Nachrichten zuständig ist. Ein Beispiel für einen Applikationsserver, der einen Messaging-Dienst und eine Implementierung des Server-Session-Pools anbietet, ist der WebLogic Applikationsserver von BEA Systems (vgl. [WEBLOGIC]). Das in Listing 6.7 gezeigte Beispiel wurde mit diesem entwickelt.

```
…
    static final String FACTORY =
                   "SampleConnectionFactory";
    static final String QUEUE = "SampleQueue";
    static final String SSP_FACTORY =
                   "SampleSessionPoolFactory";
    static final String LISTENER = "jms.client.MQListener";
    QueueConnectionFactory qcf = null;
    QueueConnection        qc  = null;
    ConnectionConsumer     cc  = null;
    qcf = (QueueConnectionFactory)
              Lookup.get(FACTORY);
    qc = qcf.createQueueConnection();
    qc.start();
    Queue queue = (Queue)Lookup.get(QUEUE);
    ServerSessionPoolFactory factory =
        (ServerSessionPoolFactory)
            Lookup.get(SSP_FACTORY);
    ServerSessionPool sessionPool;
    sessionPool = factory.getServerSessionPool
                  (qc, 5, false,
                   Session.AUTO_ACKNOWLEDGE, LISTENER);
    cc = qc.createConnectionConsumer
             (queue, "TRUE", sessionPool, 10);
…
```

Listing 6.7: Erzeugen eines Server-Session-Pools

Um ein *ServerSessionPool*-Objekt zu erzeugen, benötigt der JMS-Client eine *ServerSessionPoolFactory*. Diese bekommt er, wie z.B. auch die Connection-Factory, aus dem JNDI. Auch sie wird vom Administrator über die Konfiguration zur Verfügung gestellt. Beim Erzeugen des Server-Session-Pools wird ein *Connection*-Objekt, die Größe des Pools (im Beispiel der Wert *5*), *true* oder *false* (je nachdem ob Transaktionen verwendet wer-

den), die Art des Acknowledgements sowie der Name der Klasse angegeben, die das *MessageListener*-Interface implementiert und für die Verarbeitung der Nachrichten zuständig ist (siehe Listing 6.8). Über die Methode *createConnectionConsumer()* am *QueueConnection*-Objekt wird der Session-Pool aktiviert. Ab sofort wird damit begonnen, anstehende Nachrichten zu verarbeiten. Der Session-Pool öffnet die notwendige Anzahl an Server-Sessions, assoziiert sie mit den Threads und benutzt Instanzen der übergebenen Klasse dazu, die Nachrichten zu verarbeiten. Die *MessageListener*-Klasse muss dem Applikationsserver zur Verfügung stehen, da er für das Erzeugen der Instanzen verantwortlich ist.

```
package jms.client;

import javax.jms.*;

public class MQListener implements MessageListener {

    public MQListener() {}

    public void onMessage(Message m) {
        // ...
        // process the message
        // ...
        try {
            m.acknowledge();
        } catch(JMSException jmsex) {
            jmsex.printStackTrace();
        }
    }
}
```

Listing 6.8: Empfängerklasse des Server-Session-Pools

An dieser Stelle sei noch einmal darauf hingewiesen, dass die Instanzen der MessageListener-Klasse im Prozess des Applikationsservers erzeugt werden und nicht im Prozess des JMS-Clients. Damit findet auch die Verarbeitung der Nachrichten im Prozess des Applikationsservers statt und nicht im Prozess des JMS-Clients.

Empfangsverhalten

Eine Queue stellt eine Nachricht immer nur einem Empfänger zu. Sind mehrere Empfänger bei einer Queue registriert, hängt es von der Implementierung des JMS-Providers ab, welchem der Empfänger die Nachricht zugestellt wird. Queues haben in der Regel nur einen Empfänger. Die Fähigkeit einer JMS-Implementierung, mehrere Empfänger einer Queue zuzulassen, wird oft zu Skalierungszwecken genutzt. Mehrere Sessions und mehrere Empfänger für eine Queue zu benutzen, ist für den JMS-Client eine bequeme Möglichkeit, eingehende Nachrichten parallel zu verarbeiten.

Topics haben typischerweise mehrere Empfänger. Eine Nachricht eines Topics wird nur den gerade aktiven Subscribern zugestellt. Inaktive Subscriber bekommen die Nachricht nicht. Will ein JMS-Client sicherstellen, alle Nachrichten eines Topics zu erhalten, kann er sich als sogenannter *durable Topic-Subscriber* registrieren. Die Nachrichten eines Topics werden solange gespeichtert, bis sie an alle als *durable* registrierten Empfänger zugestellt wurden.

Anders verhält es sich bei der Nachrichtenverteilung einer Queue. Ist bei einer Queue gerade kein Empfänger registriert, bleibt die Nachricht solange in der Queue, bis sich wieder ein Empfänger registriert. Damit ist garantiert, dass jede Nachricht genau einen Client erreicht.

Im Fall von einem Serverabsturz können die oben genannten Garantien nur übernommen werden, wenn die Queue oder das Topic persistent sind. Die Nachrichten werden dann vom Messaging-Dienst persistent gespeichert und bleiben damit auch über einen Serverabsturz oder einen Serverneustart hinaus erhalten. Der Messaging-Dienst beginnt unmittelbar nach dem Neustart mit der Zustellung der verbliebenen Nachrichten.

Die Nachrichten werden normalerweise in der Reihenfolge zugestellt, in der sie in die Queue bzw. in den Topic eingefügt werden. In welcher Reihenfolge sie beim Empfänger ankommen, ist von mehreren Bedingungen abhängig. Vergibt der Sender für verschiedene Nachrichten unterschiedliche Prioritäten, beeinflusst er damit die Reihenfolge der Zustellung an den Empfänger. Nachrichten mit hoher Priorität werden vor Nachrichten mit niedriger Priorität zugestellt. Benutzt der Empfänger einen Nachrichtenfilter (siehe unten), wird die Reihenfolge der Zustellung ebenfalls beeinflusst.

6.1.6 Transaktionen und Acknowledgement

Lokale Transaktionen

Um das Zustellen und das Versenden mehrerer Nachrichten zu einer atomaren Aktion zusammenzufassen, kann der JMS-Client eine transaktionale Session verwenden (vorausgesetzt der JMS-Provider unterstützt diese Funktionalität). Listing 6.9 zeigt, wie eine transaktionale Session erzeugt wird.

```
…
QueueConnectionFactory qcf = null;
qcf = (QueueConnectionFactory)Lookup.get(FACTORY);
QueueConnection qc = qcf.createQueueConnection();
qc.createQueueSession(true, Session.AUTO_ACKNOWLEDGE);
…
```

Listing 6.9: Erzeugen einer transaktionalen Session

Beim Aufruf der Methode *createQueueSession()* bewirkt der Parameter *true*, dass die Session transaktional ist. Gleiches gilt für die Methode *createTopicSession*. Gesendete Nachrichten werden erst dann an eine Queue oder einen Topic überstellt, wenn *Session.commit()* aufgerufen wird. Empfangene Nachrichten gelten erst dann als erfolgreich empfangen, wenn der Empfänger *Session.commit()* aufruft. Die Transaktionsklammer umfasst nur das Senden oder das Empfangen einer bestimmten Nachricht. Das Senden *und* das Empfangen einer bestimmten Nachricht kann nicht in einer Transaktion geklammert werden, da die empfangende Session eine andere ist als die sendende Session. Nur das Empfangen und Senden verschiedener Nachrichten innerhalb einer Session ist transaktional.

Wenn der sendende JMS-Client eine transaktionale Session benutzt, muss er *commit* aufrufen, um die Nachricht endgültig in die Queue oder den Topic zu übertragen. Gleiches gilt für den empfangenden JMS-Client, wenn er eine transaktionale Session verwendet. Eine Nachricht, die über eine transaktionale Session gesendet wurde, kann über eine nicht-transaktionale Session empfangen werden und umgekehrt.

Der Aufruf von *commit* schließt nicht nur die aktuelle Transaktion, sondern öffnet gleichzeitig eine neue Transaktion. Transaktionen können demnach nicht geschachtelt werden. Wird *Session.rollback()* aufgerufen, werden alle seit dem letzten Aufruf der Methode *commit* gesendeten Nachrichten verworfen. Keine der Nachrichten wird an die Queue oder das Topic überstellt. Gleiches gilt für das Empfangen von Nachrichten. Die seit dem letzten Aufruf von *commit* zugestellten Nachrichten gelten erst dann als erfolgreich zugestellt, wenn die Methode *commit* ein weiteres Mal aufgerufen wird. Erst dann werden die zugestellten Nachrichten endgültig aus der Queue oder dem Topic entfernt. Wird *Session.rollback()* aufgerufen, beginnt der Messaging-Dienst mit dem erneuten Zustellen der seit dem letzten Aufruf von *commit* in der Queue bzw. im Topic verbliebenen Nachrichten an die registrierten Empfänger. Ob eine Session transaktional ist oder nicht, ob also ein JMS-Client die Methode *commit* zur Sende- oder Empfangsbestätigung aufrufen muss oder nicht, kann er über die Methode *Session.getTransacted()* herausfinden. Ob eine Nachricht wegen eines Aufrufs der Methode *rollback* erneut zugestellt wurde, kann der JMS-Client über die Methode *Message.getJMSRedelivered()* herausfinden.

Verteilte Transaktionen

Der JMS-Provider kann optional auch verteilte Transaktionen unterstützen (Details zu verteilten Transaktionen siehe Kapitel 7). Anders als bei lokalen Transaktionen einer Session, können bei einer verteilten Transaktion Aktionen verschiedener Dienste zu einer atomaren Aktion zusammengefasst werden. Diese Dienste können auch auf unterschiedlichen Servern zum Einsatz kommen. Eine verteilte Transaktion kann beispielsweise einen Methodenaufruf an einer Enterprise-Bean und das Einstellen einer Nachricht in eine Queue umfassen. Schlägt eine der beiden Aktionen fehl, werden

beide Aktionen rückgängig gemacht. Der Kontext einer verteilten Transaktion endet beim Senden einer Nachricht. Er wird nicht bis zum Empfänger der Nachricht weitergereicht. Damit kann der Empfang und die Verarbeitung der Nachricht nicht Bestandteil der verteilten Transaktion sein, in der die Nachricht gesendet wurde. Hier gilt das gleiche Prinzip wie bei einfachen Transaktionen.

Der JMS-Provider unterstützt verteilte Transaktionen über die JTA XAResource API (Details hierzu werden in Kapitel 7 behandelt). Implementiert wird die Unterstützung für verteilte Transaktionen vom JMS-Provider in der JMS-Session. Die Spezifikation definiert hierfür ein Interface *javax.jms.XASession*, welches von *javax.jms.Session* abgeleitet ist. *XASession* wird noch für die jeweiligen Kommunikationsarten (Publish-Subscribe, Point-to-Point) spezialisiert. Über die XA-Interfaces wird die Transaktion vom Transaktions-Monitor gesteuert. Der JMS-Client wird mit diesen Interfaces nicht direkt in Berührung kommen. Bewegt er sich innerhalb einer verteilten Transaktion, wird er zwar ein *XASession*-Objekt bekommen, welches ihm gegenüber aber als *Session*-Objekt auftritt. Versucht der JMS-Client innerhalb einer verteilten Transaktion die Methoden *commit* oder *rollback* am Session-Objekt aufzurufen, löst diese eine Ausnahme vom Typ *TransactionInProgressException* aus. Die Unterstützung verteilter Transaktionen in JMS ist in erster Linie für die Integrationsfähigkeit von JMS in Applikationsservern gedacht.

Acknowledgement

Wenn weder lokale noch verteilte Transaktionen verwendet werden, greift der Acknowledgement-Mechanismus. Er steuert, wann zugestellte Nachrichten endgültig aus der Queue oder dem Topic entfernt werden. Der Acknowledgement-Mechanismus ist unerheblich für das Senden von Nachrichten. Welches Acknowledge-Verhalten bei einer JMS-Session gewünscht wird, kann der JMS-Client beim Erzeugen einer Session angeben. Benutzt wird der Mechanismus durch den JMS-Client über die Methoden *Message.acknowledge()* bzw. *Session.recover()*. Die Methode *acknowledge* ist mit der *commit*-, die Methode *recover* mit der *rollback*-Methode einer Transaktion vergleichbar. Der JMS-Client kann beim Erzeugen der JMS-Session unterschiedliches Acknowledgement-Verhalten durch folgende Werte erwirken:

- *javax.jms.Session.AUTO_ACKNOWLEDGE*: Die JMS-Session bestätigt eine Nachricht automatisch, nachdem die *receive*-Methode erfolgreich aufgerufen wurde oder nachdem der Aufruf der *onMessage*-Methode an einem Message-Listener zurückgekehrt ist. Die soeben zugestellte Nachricht wird nicht noch einmal zugestellt und wird umgehend aus der Queue oder dem Topic entfernt. Bei diesem Modus entspricht das Zustellen *einer* Nachricht einer atomaren Aktion. Aufrufe der *acknowledge*-Methode des JMS-Clients werden ignoriert. Auch der Aufruf der *recover*-Methode durch den JMS-Client ändert nichts am Verhalten der Session.

- *javax.jms.Session.CLIENT_ACKNOWLEDGE*: Der JMS-Client muss die *acknowledge*-Methode explizit aufrufen, um den Empfang einer oder mehrerer Nachrichten als erfolgreich zu quittieren. Solange die *acknowledge*-Methode nicht aufgerufen wird, werden bereits zugestellte Nachrichten nicht aus der Queue oder dem Topic entfernt. Ruft der JMS-Client nach jedem Empfang einer Nachricht die *acknowledge*-Methode auf, ist das Verhalten identisch mit dem Modus AUTO_ACKNOWLEDGE. Ruft der JMS-Client die Methode *recover* auf, stoppt die Session sofort die Zustellung von Nachrichten. Die Session geht zu der Nachricht zurück, die unmittelbar nach dem letzten Aufruf von *acknowledge* eingestellt wurde und beginnt erneut mit der Zustellung. Nachrichten, die wegen eines Aufrufs von *recover* erneut zugestellt wurden, erkennt der JMS-Client durch den Aufruf der Methode *Message.getJMSRedelivered()*.
- *javax.jms.Session.DUPS_OK_ACKNOWLEDGE*: In diesem Modus bestätigt die JMS-Session den Empfang einer Nachricht wie im Modus AUTO_ACKNOWLEDGE automatisch. Allerdings bleibt es ihr überlassen, wann sie die Bestätigung vornimmt und die Nachrichten endgültig aus der Queue bzw. dem Topic entfernt. Das bedeutet, dass in diesem Modus Nachrichten mehrfach zugestellt werden können (der Präfix DUPS steht für *Duplicates*). Anders als beim Aufruf der *recover*-Methode im Modus CLIENT_ACKNOWLEDGE ist es der Nachricht nicht anzusehen, ob sie zum wiederholten Mal zugestellt wurde. Der Modus kann also nur bei JMS-Clients verwendet werden, die mit doppelt zugestellten Nachrichten umgehen können. Aufrufe der Methoden *acknowledge* und *recover* sind ohne Wirkung. Der Vorteil dieses Modus gegenüber dem Modus AUTO_ACKNOWLEDGE ist, dass die Session effizienter arbeiten kann. Das Entfernen der zugestellten Nachrichten kann beispielsweise in einem eigenen Thread passieren, der mit niedriger Priorität im Hintergrund läuft.

AUTO_ACKNOWLEDGE dürfte derjenige Modus sein, der für die meisten JMS-Clients geeignet ist. Der Mehraufwand der JMS-Session zur Laufzeit gegenüber dem DUPS_OK_ACKNOWLEDGE-Modus amortisiert sich oftmals dadurch, dass der JMS-Client einfacher zu implementieren und dadurch weniger fehleranfällig ist. Der Modus CLIENT_ACKNOWLEDGE wird bei JMS-Clients eingesetzt, die eingehende Nachrichten en bloc verarbeiten, um eine höhere Laufzeiteffizienz zu erzielen. Dies ist beispielsweise dann sinnvoll, wenn zum Verarbeiten einer Nachricht ein Datenbankzugriff erforderlich ist. Statt bei jeder eingehenden Nachricht einen Datenbankzugriff zu machen, kann für mehrere eingegangene Nachrichten eine Sammelanfrage an die Datenbank gestellt werden. So wird der Aufwand des mehrfachen Verbindungsaufbaus zur Datenbank gespart und die Datenbank wird mit weniger Anfragen belastet. War die Aktion erfolgreich, kann mit dem Aufruf der Methode *acknowledge* an der zuletzt empfangenen Nachricht der Empfang der anderen Nachrichten des Verarbeitungsblocks quittiert werden.

6.1.7 Filtern von Nachrichten

Ein JMS-Client hat die Möglichkeit, beim Empfangen von Nachrichten einen Filter anzugeben. Dieser Filter bewirkt, dass dem Client nur die Nachrichten zugestellt werden, die den im Filter angegebenen Kriterien entsprechen. Die JMS-Spezifikation nennt diesen Filtermechanismus den *Message-Selector*. Der Filter kann beim Erzeugen eines Empfängers am Interface der Session in Form eines String-Objekts angegeben werden:

- *QueueSession.createReceiver(Queue queue, java.lang.String* **messageSelector***)*
- *TopicSession.createSubscriber(Topic topic, java.jang.String* **messageSelector***, boolean noLocal)*

Über das Attribut *noLocal*, das nur bei einem Topic angegeben werden kann, besteht die Möglichkeit, die Zustellung von Nachrichten, die über dieselbe Verbindung gesendet wurden, auszuschließen. Dieses Attribut ist sinnvoll, wenn ein JMS-Client über einen bestimmten Topic Nachrichten sendet und empfängt (Chat-Anwendungen benutzen diese Vorgehensweise beispielsweise). Über *noLocal* kann er steuern, ob er Nachrichten, die er selbst sendet, empfangen will oder nicht.

Die Syntax des Message-Selectors ist angelehnt an die WHERE-Klausel der Abfragesprache SQL92. Für eine Beschreibung der Syntax siehe [JMS, 1999]. Der Message-Selector wird nur auf die Header-Felder und die Properties einer Nachricht angewendet. Er kann nicht auf den Message-Body angewendet werden.

Beispiel 1:

```
…
QueueReceiver qr;
qr = queueSession.createReceiver(queue, "JMSPriority >= 5");
…
```

Dem in Beispiel 1 gezeigten Empfänger werden vom Messaging-Dienst nur die Nachrichten aus der Queue zugestellt, deren Priorität größer ist als fünf. *JMSPriority* ist ein von der JMS-Spezifikation definiertes Header-Feld und ist bei jeder Nachricht gesetzt. Wenn die Priorität vom Sender nicht ausdrücklich angegeben wird, ist das Feld *JMSPriority* mit einem Standardwert belegt. *JMSPriority* kann Werte von 0 bis 9 annehmen, wobei 0 die niedrigste und 9 die höchste Priorität ist. Die Standardpriorität hat den Wert 4.

Beispiel 2:

```
…
QueueReceiver qr;
String selector = "AppProp in ('aProp', 'bProp')";
qr = queueSession.createReceiver(queue, selector);
…
```

Dem in Beispiel 2 gezeigten Empfänger werden vom Messaging-Dienst nur die Nachrichten aus der Queue zugestellt, bei denen das von der Applikation definierte Property *AppProp* den Wert *aProp* oder *bProp* hat. Der Sender einer Nachricht setzt das Property *AppProp* über die Methode *setStringProperty* am Interface *javax.jms.Message*.

Die Verwendung eines Message-Selektors bewirkt bei einem bestimmten Empfänger, dass ihm nur die Nachrichten, die den Kriterien des Selektors entsprechen, zugestellt werden. Bei einer Queue hat das zur Folge, dass die Nachrichten, die den Kriterien nicht entsprechen, solange in der Queue verbleiben, bis sich ein Empfänger registriert,

- der keinen Message-Selektor verwendet,
- der einen Message-Selektor verwendet, dem die Header und Properties dieser Nachricht entsprechen.

Bei einem Topic werden Nachrichten, die aufgrund von Selektor-Kriterien keinem Empfänger zugestellt werden können, gar nicht zugestellt. Für den oder die Empfänger ist es so, als wäre die Nachricht nie gesendet worden.

6.1.8 Request-Reply

Messaging-Systeme sind von Haus aus auf asynchrone Kommunikation ausgelegt. Ein JMS-Client schickt eine Nachricht und stößt damit auf der Empfängerseite bestimmte Vorgänge an. Er weiß nicht genau, wann der Anstoß stattfindet (wann also die Nachricht empfangen wird), auch nicht wer die Verarbeitung vornimmt (wer also die Nachricht empfängt). Bei einem klassischen RPC-Aufruf (Remote Proceduce Call) beispielsweise muss der Client warten, bis die Verarbeitung der Prozedur abgeschlossen ist. Nachdem der RPC-Aufruf zurückgekehrt ist, nimmt der Client das Ergebnis der Verarbeitung in Form des Rückgabewerts in Empfang. Dieser Rückgabewert kann für den RPC-Client verschiedene Bedeutungen haben. Er kann ihm signalisieren, ob die Verarbeitung erfolgreich war oder ob sie fehlgeschlagen ist. Der Rückgabewert kann außerdem die weitere Vorgehensweise des Clients beeinflussen.

Wenn ein Messaging-Dienst für die Kommunikation zwischen den Komponenten einer verteilten Anwendung eingesetzt wird, ist das eben geschilderte Verhalten ofmals erwünscht. Durch den Request-Reply-Mechanismus kann über einen Messaging-Dienst ein Verhalten simuliert werden, dass einem klassischen RPC-Aufruf entspricht. Das bedeutet, dass auch über einen Messaging-Dienst synchrone Kommunikation stattfinden kann. Der einzige Unterschied zum klassischen RPC besteht darin, dass der Aufrufer nicht entscheiden kann, von wem der Request entgegengenommen wird.

Die JMS-Spezifikation stellt Hilfsklassen zur Verfügung, mit denen ein Request-Reply-Verhalten erreicht werden kann. Diese Klassen sind *javax.jms.QueueRequestor* für das Point-to-Point-Kommunikationsmodell und *javax.jms.TopicRequestor* für das Publish-

Subscribe-Kommunikationsmodell. Listing 6.10 zeigt an einem Beispiel, wie die Klasse *QueueRequestor* verwendet wird (die Verwendung der Klasse *TopicRequestor* ist entsprechend).

```
…
    QueueConnection qc    = null;
    QueueSession    qs    = null;
    QueueSender     qsend = null;
    QueueRequestor  qr    = null;
    try {
        QueueConnectionFactory qcf = null;
        qcf = (QueueConnectionFactory)
                   Lookup.get(FACTORY);
        qc = qcf.createQueueConnection();
        qc.start();
        qs = qc.createQueueSession(false,
                                   Session.AUTO_ACKNOWLEDGE);
        Queue queue = (Queue)Lookup.get(qn);
        qsend = qs.createSender(queue);
        m = qs.createTextMessage("a message");
        qr = new QueueRequestor(qs, queue);
        Message ret = qr.request(m);
        //...
        //Verarbeiten der Antwort
        //...
    } finally {
        try  { qr.close(); }    catch(Exception ex) {}
        try  { qsend.close(); } catch(Exception ex) {}
        try  { qs.close(); }    catch(Exception ex) {}
        try  { qc.close(); }    catch(Exception ex) {}
    }
…
```

Listing 6.10: Request-Reply mit der Klasse QueueRequestor

Der Code ist weitgehend identisch mit dem Versenden einer Nachricht, wie es in Abschnitt 6.2.5 gezeigt wurde. Der wesentliche Unterschied besteht in der Verwendung der Klasse *QueueRequestor*. Statt einen Queue-Sender über die Queue-Session zu erzeugen, wird eine Instanz der Klasse *javax.jms.QueueRequestor* erzeugt. Ihr wird die Queue-Session und die Queue als Parameter übergeben. Zum Senden der Nachricht ruft der JMS-Client die Methode *request()* auf und übergibt die zu sendende Nachricht. Die Nachricht wird über die Queue versandt, die dem Queue-Requestor im Konstruktor übergeben wurde. Gleichzeitig legt der Queue-Requestor eine temporäre Queue an, über die er die Antwort erwartet. Temporäre Queues (Gleiches gilt für temporäre Topics) sind nur gültig, solange die Verbindung, über die sie erzeugt wurden, aktiv sind. Nur die Sessions, die zu der Verbindung gehören, über die die temporäre Queue oder das temporäre Topic erzeugt wurde, dürfen einen *Empfänger* für diese temporäre

Queue bzw. Topic erzeugen. Der Aufruf der *request()*-Methode blockiert den Aufrufer solange, bis über die temporäre Queue eine Antwort eingeht. Listing 6.11 zeigt einen Empfänger, der auf die gesendete Nachricht aus Listing 6.10 antwortet.

```
…
    public void onMessage(Message m) {
        // ...
        // process Message
        // ...
        QueueSender qs = null;
        try {
            TextMessage tm =
                queueSession.createTextMessage();
            tm.setText("Reply:" +
                        ((TextMessage)m).getText());
            Queue reply = (Queue)m.getJMSReplyTo();
            qs = queueSession.createSender(reply);
            qs.send(tm);
            m.acknowledge();
        } catch(JMSException jmsex) {
            jmsex.printStackTrace();
        } finally {
            try { qs.close(); } catch(Exception ex) {}
        }
    }
…
```

Listing 6.11: Antwort eines JMS-Clients auf eine Nachricht

Um auf die Nachricht zu antworten, wertet der Empfänger das Header-Feld *JMSReplyTo* aus. Es enthält die (temporäre) Queue bzw. den (temporären) Topic, über die der Sender der Nachricht die Antwort erwartet. Unter anderen Umständen hätte er gar keinen Zugriff auf die temporäre Queue. Wegen der Eigenschaften temporärer Queues und Topics (wie sie oben beschrieben wurden) ist sichergestellt, dass nur der Empfänger einer Nachricht eine Antwort senden kann. Das Feld *JMSReplyTo* wird vom QueueRequestor über die Methode *setJMSReplyTo()* am Interface *javax.jms.Message* gesetzt. Der Empfänger erzeugt einen Queue-Sender für die temporäre Queue und schickt über ihn die Antwort an den Sender der ursprünglichen Nachricht. Ist die Antwort beim Sender eingegangen, kehrt der Aufruf der *request()*-Methode zurück und der Sender kann die Antwort auswerten.

Da beim Request-Reply temporäre Queues bzw. Topics eingesetzt werden, sollten in der Antwort keine kritischen Informationen enthalten sein. Der Sender der Nachricht muss damit rechnen, dass die Antwort z.B. wegen eines Serverabsturzes oder wegen eines Verbindungsabbruchs verloren geht. Eine temporäre Queue oder Topic existiert (wie bereits erwähnt) nur im Kontext einer bestimmten JMS-Connection. Bricht diese Verbindung ab, ist die temporäre Queue bzw. das temporäre Topic samt der enthaltenen Nachrichten verloren.

6.2 Konzepte

6.2.1 Die Sicht des Clients auf die Message-Driven-Bean

Eine Message-Driven-Bean ist dem Client gegenüber ein Nachrichtenempfänger (Message-Consumer) im Sinne des Java Message Service. Der Client kann eine Message-Driven-Bean nicht direkt ansprechen. Sie hat kein Home-, kein Remote-, kein Local-Home- und kein Local-Interface. Der Client kann eine Message-Driven-Bean nur indirekt ansprechen, indem er eine Nachricht über einen bestimmten Kanal (Destination) des JMS sendet, von dem er weiß, dass der Empfänger eine bestimmte Message-Driven-Bean ist. Der Client hat keinen Einfluss auf den Lebenszyklus einer Message-Driven-Bean. Dieser wird ausschließlich vom EJB-Container gesteuert. Der Client hat auch keinerlei Einfluss darauf, wie viele Instanzen einer Message-Driven-Bean für die Verarbeitung von Nachrichten eines bestimmten Kanals benutzt werden. Auch das wird ausschliesslich vom Container gesteuert. Die Instanzen einer Message-Driven-Bean stehen einem Client nicht exklusiv zur Verfügung und sie können für ihn keinen Zustand speichern. Sie sind zustandslos wie zustandslose (stateless) Session-Beans.

Der Client kann über den JNDI einen bestimmten Kanal des Messaging-Dienstes lokalisieren (vgl. Abschnitt 6.2.5). Ein Client, der (indirekt) Message-Driven-Beans benutzt, unterscheidet sich nicht von einem normalen JMS-Client. Benutzt der Client eine Session- oder eine Entity-Bean, ist er fest an diese gekoppelt. Benutzt der Client eine Message-Driven-Bean über den Kanal eines Messaging-Dienstes, ist er lose mit dieser Enterprise-Bean gekoppelt. Würde die Message-Driven-Bean zur Laufzeit gegen einen anderen Empfänger (z. B. eine andere Message-Driven-Bean oder eine herkömmliche Java-Klasse) ausgetauscht werden, hätte das keine Auswirkungen auf den Client. Die API des Java Message Services liegt als Indirektionsebene dazwischen.

6.2.2 Lebenszyklus einer Message-Driven-Bean

Der Lebenszyklus einer Message-Driven-Bean wird ausschließlich vom EJB-Container gesteuert. Die Instanz einer Message-Driven-Bean kann folgende Zustände annehmen:

- *Nicht existent*: Die Instanz existiert nicht.
- *Pooled ready*: Die Instanz existiert und steht für Methodenaufrufe zur Verfügung.

Abbildung 6.6 stellt den Lebenszyklus einer Message-Driven-Bean grafisch dar. Aus der Abbildung wird deutlich, dass lediglich der Container den Zustandsübergang veranlassen kann.

Instanzen einer Message-Driven-Bean werden in der Regel beim Hochfahren des EJB-Containers erzeugt oder wenn der EJB-Container die Poolgröße erhöht, um eine angemessen schnelle Verarbeitung der eingehenden Nachrichten zu gewährleisten. Der

Lebenszyklus einer Message-Driven-Bean beginnt mit dem Aufruf des Konstruktors. Danach ruft der Conainer die Methode *setMessageDrivenContext* auf und übergibt damit der Instanz ihren Kontext. Über den Kontext kann eine Message-Driven-Bean mit dem EJB-Container kommunizieren. Anschließend ruft der Container die Methode *ejbCreate* auf, was die Instanz der Message-Driven-Bean in den Zustand *Pooled Ready* versetzt. Ab diesem Zeitpunkt kann sie vom Container für die Verarbeitung eingehender Nachrichten verwendet werden, die von einem Client über einen bestimmten Kanal des Messaging-Dienstes gesendet wurden. Der EJB-Container verwendet für die Verarbeitung einer eingehenden Nachricht eine beliebige Instanz eines bestimmten Typs einer Message-Driven-Bean. Er unterhält einen Pool von Instanzen für jeden Typ einer Message-Driven-Bean.

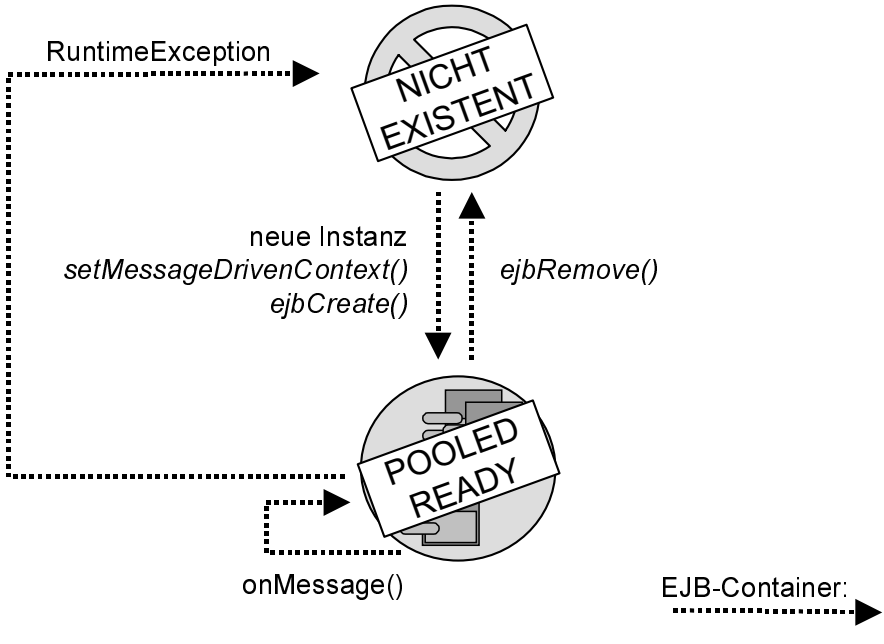

Abbildung 6.6: Lebenszyklus einer Message-Driven-Bean

Wenn der EJB-Container eine Instanz einer Message-Driven-Bean nicht mehr benötigt, ruft er die *ejbRemove*-Methode auf. Mit dem Aufruf dieser Methode endet der Lebenszyklus einer Message-Driven-Bean. Das ist dann der Fall, wenn der EJB-Container heruntergefahren wird oder der EJB-Container den Pool von Instanzen eines bestimmten Bean-Typs verkleinern will. Der Lebenszyklus einer Message-Driven-Bean endet ebenfalls, wenn die Message-Driven-Bean-Instanz während der Verarbeitung einer Nachricht eine Ausnahme vom Typ *RuntimeException* wirft.

6.2.3 Parallele Verarbeitung

Da der EJB-Container in der Regel mehrere Instanzen einer Message-Driven-Bean benutzt, kann er die Verarbeitung unmittelbar nacheinander eingehender Nachrichten auf mehrere Instanzen verteilen. Nachrichten können so von den Instanzen eines Typs einer Message-Driven-Bean parallel verarbeitet werden (siehe Abbildung 6.7). Der Container sorgt dabei für das Thread-Management und stellt der Message-Driven-Bean die von den anderen Bean-Typen bekannten Dienste zur Verfügung.

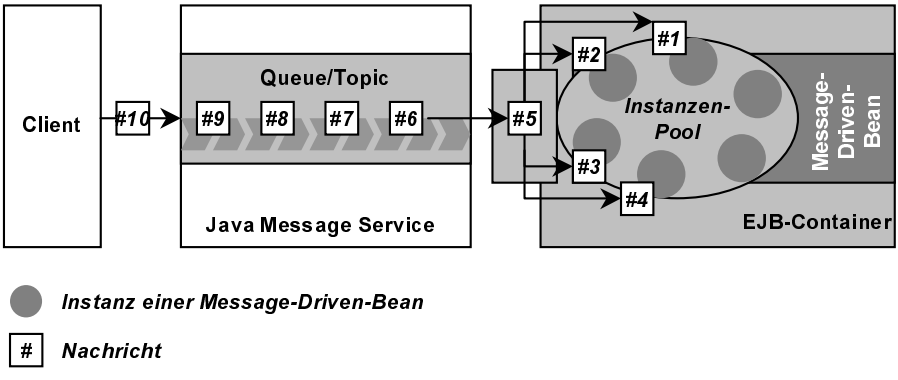

Abbildung 6.7: Parallele Verarbeitung bei Message-Driven-Beans

Das Konzept, das der EJB-Container zur parallelen Verarbeitung von Nachrichten einsetzt, ähnelt stark dem Konzept des Server-Session-Pools, der in Abschnitt 6.2.6.3 vorgestellt wurde. Vereinfachend könnte man folgende Formel aufstellen:

Server-Session-Pool
+ *zustandslose (stateless) Session-Bean*
- *Home-/Remote- bzw. Local-Home-/Local-Interface*
= Message-Driven-Bean

Parallele Verarbeitung bedeutet, dass *eine bestimmte* Instanz einer Message-Driven-Bean die Nachrichten nicht in der Reihenfolge verarbeitet, wie sie vom Client geschickt werden.

Der Container muss dafür sorgen, dass die Aufrufe an den Bean-Instanzen zur Verarbeitung der Nachrichten serialisiert werden. Damit muss die Implementierung der Bean-Klasse nicht thread-sicher sein, was die Entwicklung von Message-Driven-Beans vereinfacht.

6.3 Programmierung

Abbildung 6.8 gibt einen vollständigen Überblick über die Klassen und Interfaces von Message-Driven-Beans. Der auffälligste Unterschied zu Session- und Entity-Beans ist das Fehlen von Home- und Remote- bzw. Local-Home- und Local-Interfaces.

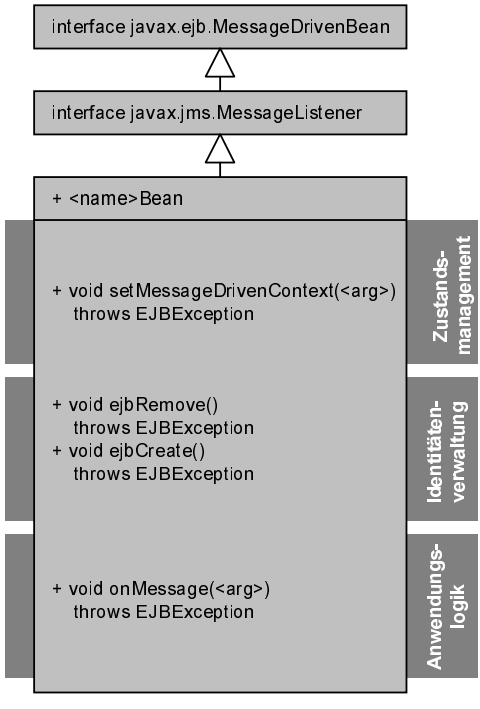

Abbildung 6.8: Interfaces der Message-Driven-Bean

Die Message-Driven-Bean implementiert das Interface *javax.ejb.MessageDrivenBean* und das Interface *java.jms.MessageListener* (vgl. auch Abschnitt 6.2.6.2) direkt oder indirekt. Die Klasse muss *public* deklariert sein. Sie darf nicht *final* und nicht *abstract* deklariert sein. Die Klasse muss einen parameterlosen Konstruktor haben, der *public* deklariert ist. Dieser wird vom EJB-Container benutzt, um Instanzen einer Message-Driven-Bean zu erzeugen. Die Klasse darf nicht die Methode *finalize* implementieren.

Die Methoden einer Message-Driven-Bean lassen sich in drei Gruppen einteilen:

- Zustandsmanagement,
- Identitätenverwaltung,
- Anwendungslogik.

Die folgenden Abschnitte beschreiben die einzelnen Methoden in diesen Gruppen. Einen schnellen Einstieg in die Programmierung dieser Methoden können Sie sich auch durch das Studium des Programmierbeispiels erarbeiten.

6.3.1 Zustandsmanagement

Bei den Message-Driven-Beans gibt es nur eine Methode, die eine Bean-Instanz über einen bevorstehenden Zustandswechsel informiert, nämlich die Methode *setMessageDrivenContext*. Im Fall von Message-Driven-Beans wird der Zustandswechsel ausschließlich vom Container eingeleitet. Die Implementierung der Methode *setMessageDrivenContext* speichert den übergebenen Kontext. Er kann später benutzt werden, um mit dem Container zu kommunizieren (in Abschnitt 6.4.5 werden wir näher auf den Kontext eingehen). Die Spezifikation gestattet in dieser Methode nur Zugriffe auf das Environment der Message-Driven-Bean über den JNDI. Wie eine Enterprise-Bean auf ihr Environment zugreift, ist ausführlich in Kapitel 4, *Session-Beans*, beschrieben. Unmittelbar nach dem Aufruf der Methode *setMessageDrivenContext* folgt der Übergang in den Zustand *Pooled Ready*.

6.3.2 Verwaltung der Identität

Die Methoden der Identitätenverwaltung bilden die zweite Gruppe von Methoden einer Message-Driven-Bean. Sie ermöglichen das Erzeugen und das Löschen von Message-Driven-Bean-Identitäten. Anders als bei Session- und Entity-Beans werden diese Methoden ausschließlich vom EJB-Container benutzt. Der Client hat keinerlei Möglichkeit, auf das Erzeugen und das Löschen von Message-Driven-Bean-Identitäten Einfluss zu nehmen. Eine Instanz einer Message-Driven-Bean hat gegenüber dem Client keine Identität. Der EJB-Container benutzt sie lediglich intern für die Verwaltung der Instanzen eines bestimmten Typs einer Message-Driven-Bean.

ejbCreate

Durch den Aufruf von *ejbCreate* versetzt der EJB-Container eine Instanz einer Message-Driven-Bean in den Zustand *Pooled Ready* und verwendet sie ab sofort für die Verarbeitung eingehender Nachrichten. Eine Message-Driven-Bean muss genau eine Methode *ejbCreate* ohne Parameter definieren. Der Message-Driven-Bean ist innerhalb dieser Methode nur der Zugriff auf ihr Environment über JNDI und gegebenenfalls auf ein User-Transaction-Objekt über den Message-Driven-Kontext gestattet. Der Zugriff auf das Environment über JNDI ist in Kapitel 4, *Session-Beans*, beschrieben. Der Umgang mit Transaktionen und die Verwendung des Interfaces *javax.transaction.UserTransaction* wird in Kapitel 7, *Transaktionen*, beschrieben.

ejbRemove

Wenn der EJB-Container eine Bean-Instanz nicht mehr benötigt, löscht er diese durch den Aufruf der *ejbRemove*-Methode. Damit zerstört er ihre Identität. Sie kann nicht mehr für die Verarbeitung von Nachrichten verwendet werden. Falls die Bean Ressourcen belegt, muss sie diese beim Aufruf dieser Methode freigeben. Wie bei der Methode *ejbCreate* ist der Bean innerhalb dieser Methode nur der Zugriff auf ihr Environment über JNDI und gegebenenfalls auf ein *UserTransaction*-Objekt über den Message-Driven-Kontext gestattet.

6.3.3 Anwendungslogik

Die Anwendungslogik einer Message-Driven-Bean kann nur in einer Methode implementiert werden, nämlich *onMessage*. Sie kann von keinem Client aufgerufen werden. Sie wird immer dann vom EJB-Container aufgerufen, wenn eine Nachricht über den der Message-Driven-Bean zugeordneten Kanal des Messaging-Dienstes eingeht. Der EJB-Container muss dafür sorgen, dass die Aufrufe an dieser Methode serialisiert werden. Die Implementierung dieser Methode muss also nicht thread-sicher sein. Mit welchem Kanal (Destination) des Messaging-Dienstes die Message-Driven-Bean assoziiert wird, wird im Deployment-Deskriptor festgelegt.

In der Methode *onMessage* ist der Zugriff auf das Environment der Bean über den JNDI auf die Dienste, die der Container anbietet (Datenbank, Java Message Service etc.), sowie der Zugriff auf andere Enterprise-Beans gestattet. Wie eine Enterprise-Bean auf ihr Environment über JNDI zugreift, ist in Kapitel 4, *Session-Beans*, beschrieben. Dort wird auch erklärt, wie eine Enterprise-Bean auf die Dienste und Resourcen des EJB-Containers zugreifen kann. Benutzt eine Message-Driven-Bean eine andere Enterprise-Bean, ist sie ihr gegenüber ein ganz normaler Enterprise-Bean-Client. Die Sicht eines Clients auf eine Session- oder Entity-Bean ist in den einschlägigen Kapiteln (Kapitel 4, *Session-Beans*, Kapitel 5, *Entity-Beans*) beschrieben.

Bei Transaktionen, die vom Container gesteuert werden, kann die Bean die Methoden *getRollbackOnly* und *setRollbackOnly* des Interface *MessageDrivenContext* verwenden. Bei Transaktionen, die von der Bean selbst gesteuert werden, kann sie ein *UserTransaction*-Objekt verwenden, auf welches sie über den Message-Driven-Kontext Zugriff hat. Transaktionen und der Umgang mit einem Objekt vom Typ *javax.transaction.UserTransaction* sind in Kapitel 7, *Transaktionen*, beschrieben. Die Spezifikation schreibt vor, dass eine Message-Driven-Bean den Acknowledgement-Mechanismus (vgl. Abschnitt 6.2.7) des JMS *nicht* benutzen darf. Falls die Message-Driven-Bean Container-gesteuerte Transaktionen verwendet, wird der Empfang einer Nachricht mit dem *commit* der Transaktion vom EJB-Container automatisch quittiert. Steuert die Message-Driven-Bean Transaktionen selbst, ist der Empfang der Nachricht nicht Bestandteil der Transaktion. In diesem Fall empfiehlt die Spezifikation den Acknowledge-Modus auf AUTO_

ACKNOWLEDGE einzustellen. Diese Einstellung wird im Deployment-Deskriptor vorgenommen. Fehlt die Einstellung des Acknowledgement-Mechanismus im Deployment-Deskriptor, wird vom EJB-Container die Einstellung AUTO_ACKNOWLEDGE automatisch vorgenommen.

Die Methode *onMessage* deklariert keine Exceptions und sie sollte auch keine Exceptions werfen (das gilt auch für *javax.ejb.EJBException*). Ein normaler JMS-Client, der Nachrichten empfängt, hat ebenso wenig die Möglichkeit, in dieser Methode eine Exception zu werfen. Wirft die Message-Driven-Bean eine Exception vom Typ *RuntimeException*, wird sie vom EJB-Container in den Zustand *Nicht Existent* versetzt. Er geht davon aus, dass diese Instanz nicht mehr in der Lage ist, Nachrichten korrekt zu verarbeiten.

6.3.4 Der Message-Driven-Kontext

Über den *MessageDrivenContext* hat die Instanz einer Message-Driven-Bean die Möglichkeit, mit dem EJB-Container zu kommunizieren. Das ist bei Message-Driven-Beans v.a. für die Steuerung von Transaktionen notwendig. Der Kontext einer Message-Driven-Bean kann vom EJB-Container während der Lebensdauer der Enterprise-Bean verändert werden. Übergeben wird der Kontext über die Methode *setMessageDrivenKontext* des Interface *javax.ejb.MessageDrivenBean*. Die Methoden des Message-Driven-Kontexts werden nachfolgend kurz vorgestellt:

setRollbackOnly

Diese Methode dient der Transaktionssteuerung (siehe auch Kapitel 7, *Transaktionen*). Sie wird von der Bean-Instanz verwendet, um im Falle eines Fehlers alle Aktionen einer Transaktion rückgängig zu machen. Diese Methode kann nur von Message-Driven-Beans verwendet werden, die vom Container gesteuerte Transaktionen verwenden.

getRollbackOnly

Die Methode *getRollbackOnly* ist das lesende Gegenstück zur Methode *setRollbackOnly*. Mit dieser Methode kann überprüft werden, ob der momentane Methodenaufruf noch erfolgreich durchgeführt werden kann. Auch diese Methode kann nur von Message-Driven-Beans verwendet werden, die vom Container gesteuerte Transaktionen verwenden.

getUserTransaction

Diese Methode kann nur von Message-Driven-Beans aufgerufen werden, die Transaktionen selbst steuern. Der Rückgabewert dieser Methode ist ein Objekt vom Typ *javax.transaction.UserTransaction* aus der Java Transaction API (siehe auch Kapitel 7, *Transaktionen*).

getCallerPrincipal

Diese Methode erbt der Message-Driven-Kontext vom Interface *EJBContext*. Eine Message-Driven-Bean darf diese Methode nicht aufrufen. Normalerweise macht es diese Methode möglich, den Benutzer zu bestimmen, der diesen Methodenaufruf vornimmt. Da eine Message-Driven-Bean von einem Client nicht direkt angesprochen werden kann, ist der Aufruf dieser Methode ohnehin nicht sinnvoll.

isCallerInRole

Für diese Methode gilt das Gleiche wie für die Methode *getCallerPrincipal*. Diese Methode würde erlauben, die Rolle des Benutzers, der den momentanen Methodenaufruf tätigt, zu bestimmen.

getEJBHome

Auch der Aufruf dieser Methode ist einer Message-Driven-Bean untersagt. Auch sie wird aus dem Interface *EJBContext* geerbt. Die Methode erlaubt Zugriff auf eine Instanz der Implementierung des (Local-)Home-Interfaces der Bean. Da eine Message-Driven-Bean kein (Local-)Home-Interface benötigt, ist ein Aufruf dieser Methode auch wenig sinnvoll.

Der Versuch einer Message-Driven-Bean, eine verbotene Methode am Message-Driven-Kontext aufzurufen, wird mit einer *java.lang.IllegalStateException* quittiert.

6.3.5 Client

Die Programmierung des Clients entspricht exakt der Programmierung eines normalen JMS-Clients. Die Programmierung von JMS-Clients wurde ausführlich in Abschnitt 6.2 dargestellt.

6.4 Beispiel

Ein Anwendungssystem will beispielsweise Lagerbuchungen protokollieren. Da die Protokolldaten zu Revisionszwecken benötigt werden, sollen sie sicher in der Datenbank aufbewahrt werden. Um die Durchführung einer Bestandsbuchung nicht unnötig zu verlängern, entscheidet man sich dafür, die Protokollierung asynchron durchzuführen. Da die Daten nicht verloren gehen dürfen, wird der Java Message Service benutzt. Durch die Unterstützung für verteilte Transaktionen und persistente Queues und Topics bietet er ausreichend Datensicherheit für diesen Zweck. Die Message-Driven-Bean *LoggerBean* ist dafür zuständig, eingehende Daten in einer Datenbanktabelle zu sichern (siehe Listing 6.12).

```
package ejb.logger;

import javax.ejb.*;
import javax.jms.MessageListener;
import javax.jms.Message;
import javax.jms.TextMessage;
import javax.jms.JMSException;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.sql.DataSource;
import java.sql.SQLException;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.Date;

public class LoggerBean
    implements MessageDrivenBean, MessageListener {

    private static final String DS =
            "java:/comp/env/ds-name";
    private static final String ST =
            "insert into logs(tst, msg) values(?, ?)";

    private MessageDrivenContext mCtx       = null;

    private transient DataSource dataSource = null;

    public LoggerBean() {}

    public void setMessageDrivenContext
                    (MessageDrivenContext ctx)
    {
        mCtx = ctx;
    }

    public void ejbCreate() { }

    public void ejbRemove() {
        dataSource = null;
    }

    public void onMessage(Message message) {
        try {
            initDataSource();
        } catch(NamingException nex) {
            //Fehler-Reporting
            mCtx.setRollbackOnly();
            return;
        }
        String msg = null;
```

```
            try {
                msg = ((TextMessage)message).getText();
            } catch(ClassCastException ccex) {
                //Fehler-Reporting
                mCtx.setRollbackOnly();
                return;
            } catch(JMSException jmsex) {
                //Fehler-Reporting
                mCtx.setRollbackOnly();
                return;
            }
            try {
                logMessage(msg);
            } catch(SQLException sqlex) {
                //Fehler-Reporting
                mCtx.setRollbackOnly();
                return;
            }
        }

        private void logMessage(String msg)
            throws SQLException
        {
            Connection        con = dataSource.getConnection();
            PreparedStatement pst = null;
            String            mm  = (msg == null) ? "" : msg;
            try {
                pst = con.prepareStatement(ST);
                Date dd = new Date(System.currentTimeMillis());
                pst.setDate(1, dd);
                pst.setString(2, mm);
                pst.execute();
            } finally {
                try { pst.close(); } catch(Exception ex) {}
                try { con.close(); } catch(Exception ex) {}
            }
        }

        private void initDataSource()
            throws NamingException
        {
            if(dataSource != null) {
                return;
            }
            Context ctx = new InitialContext();
            String dsname = (String)ctx.lookup(DS);
            dataSource = (DataSource)ctx.lookup(dsname);
        }

    }
```

Listing 6.12: Beispiel einer Message-Driven-Bean

Die *LoggerBean* besorgt sich über ihr Environment den Namen der Data-Source, die sie für den Datenbankzugriff benutzt. Über den JNDI verschafft sie sich Zugriff auf diese Data-Source. Beim Eingang einer Nachricht ruft der EJB-Container die *onMessage*-Methode auf und übergibt der Enterprise-Bean die Nachricht. Die *LoggerBean* erwartet eine Nachricht vom Typ *javax.jms.TextMessage*. Sie entnimmt die Textnachricht und speichert sie zusammen mit dem Datum und der Uhrzeit des Empfangs in einer Datenbanktabelle mit dem Name *logs*. Tritt ein Fehler auf, den die LoggerBean nicht beheben kann, benutzt sie die Methode *setRollbackOnly* des Message-Driven-Kontexts. Damit zeigt sie dem EJB-Container an, dass die Verarbeitung fehlgeschlagen ist. Die *LoggerBean* benutzt vom EJB-Container gesteuerte Transaktionen (siehe das Beispiel Deployment-Deskriptor in Listing 6.13). Wird die Transaktion nicht commited, wird dem Messaging-Dienst die nicht erfolgreich verarbeitete Nachricht erneut zugestellt.

Listing 6.13 zeigt den zur *LoggerBean* gehörenden Deployment-Deskriptor. Für eine vollständige Beschreibung der Elemente des Deployment-Deskriptors für eine Message-Driven-Bean vergleiche [Sun Microsystems, 2001].

```
<?xml version="1.0" ?>
<!DOCTYPE ejb-jar PUBLIC
 "-//Sun Microsystems, Inc.//DTD EnterpriseJavaBeans 2.0//EN"
 "http://java.sun.com/dtd/ejb-jar_2_0.dtd">
<ejb-jar>
 <enterprise-beans>
   <message-driven>
     <!-- Name der Enterprise-Bean -->
     <ejb-name>loggerBean</ejb-name>
     <!-- Klasse der Enterprise-Bean -->
     <ejb-class>ejb.logger.LoggerBean</ejb-class>
     <!-- Gibt an, wer Transaktionen steuern soll -->
     <transaction-type>Container</transaction-type>
     <!-- Gibt an, für welche Queue oder Topic die
      Enterprise-Bean als Empfänger registriert wird. -->
     <message-driven-destination>
       <jms-destination-type>javax.jms.Queue
       </jms-destination-type>
     </message-driven-destination>
     <!-- Environment-Einträge -->
     <env-entry>
       <env-entry-name>ds-name</env-entry-name>
       <env-entry-type>java.lang.String</env-entry-type>
       <env-entry-value>wombatDS</env-entry-value>
     </env-entry>
     <!-- Gibt an, mit welchen Befugnissen die
       Enterprise-Bean ausgestattet wird. -->
     <security-identity>
       <run-as-specified-identity>
         <role-name>guest</role-name>
       </run-as-specified-identity>
```

```
      </security-identity>
    </message-driven>
  </enterprise-beans>
</ejb-jar>
```

Listing 6.13: Deployment-Deskriptor einer Message-Driven-Bean

Damit ein Vorgang protokolliert werden kann, muss eine Nachricht über den Messaging-Dienst verschickt werden. Dazu könnte von jedem Bestandteil des Systems, der eine Buchung durchführt, eine Hilfsklasse benutzt werden, wie sie Listing 6.14 zeigt.

```
package ejb.logger;

import javax.jms.JMSException;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueConnection;
import javax.jms.QueueSession;
import javax.jms.QueueSender;
import javax.jms.Queue;
import javax.jms.Session;
import javax.jms.Message;
import javax.jms.TextMessage;
import javax.naming.NamingException;
import ejb.util.*;

public class Logger {

    public static final String FACTORY    =
        "java:/env/jms/DefaultConnectionFactory";
    public static final String QUEUE_NAME =
        "java:/env/jms/LoggerQueue";

    private static QueueConnectionFactory queueCF  = null;
    private static Queue                  theQueue = null;

    static {
        try {
            queueCF  = (QueueConnectionFactory)
                        Lookup.get(FACTORY);
            theQueue = (Queue)
                        Lookup.get(QUEUE_NAME);
        } catch(NamingException nex) {
            nex.printStackTrace();
            throw new
                IllegalStateException(nex.getMessage());
        }
    }

    public static void log(String logMessage)
        throws NamingException, JMSException
```

```
    {
        QueueConnection qc    = null;
        QueueSession    qs    = null;
        QueueSender     qsend = null;
        try {
            qc = queueCF.createQueueConnection();
            qs = qc.createQueueSession
                        (false, Session.AUTO_ACKNOWLEDGE);
            qsend = qs.createSender(theQueue);
            TextMessage tm = qs.createTextMessage();
            tm.setText(logMessage);
            qsend.send(tm);
        } finally {
            try  { qsend.close(); } catch(Exception ex) {}
            try  { qs.close(); }    catch(Exception ex) {}
            try  { qc.close(); }    catch(Exception ex) {}
        }
    }

}
```

Listing 6.14: Beispiel eines Clients einer Message-Driven-Bean

Das Beispiel verwendet eine Hilfsklasse namens *Lookup*, die die Methode *get* hat. In dieser Hilfsklasse und der Methode *get* ist der Zugriff auf den Namensdienst gekapselt, um den Beispielcode auf das Wesentliche zu beschränken. Die Implementierung dieser Hilfsklasse ist Bestandteil des Beispiel-Quellcodes zu diesem Buch, der von angegebener Stelle bezogen werden kann.

Über die statische Methode *log* kann eine Textnachricht über die *LoggerQueue* gesendet werden. Diese Nachricht wird von einer Instanz der *LoggerBean* empfangen und in der Datenbank gespeichert. Der in Listing 6.14 gezeigte »Client-Code« für die Benutzung einer Message-Driven-Bean unterscheidet sich nicht von dem Code eines normalen JMS-Clients. Der gezeigte Code könnte auf zwei verschiedene Arten optimiert werden. Wenn Nachrichten nur von einem Thread versendet werden, könnte die JMS-Verbindung und die JMS-Session offen gehalten werden, um das permanente Öffnen und Schließen zu vermeiden (vgl. Abschnitt 6.2.5.1). Wird die Klasse Logger in einer Anwendung verwendet, bei der mehrere Threads Nachrichten schicken, könnte ein Session-Pool eingesetzt werden, wie er in Abschnitt 6.2.5.2 vorgestellt wurde.

6.5 Zusammenfassung

Message-Driven-Beans sind zweifelsohne eine Bereicherung für das Komponentenmodell der Enterprise JavaBeans. Durch Message-Driven-Beans wird sowohl asynchrone als auch parallele Verarbeitung von Anwendungslogik möglich. Diese beiden Funktionalitäten sind wichtige Bausteine bei der Entwicklung von Anwendungen, die bei

Unternehmen eingesetzt werden. Sie tragen in erster Linie zur Verbesserung der Performanz und zur Verbesserung der Antwortzeiten des Systems bei. Asynchronität und parallele Verarbeitung sind sehr komplexe und schwierige Themenbereiche der Informatik. Sie bringen viele Vorteile mit sich und sind für die Brauchbarkeit vieler Systeme unbedingte Voraussetzung, allerdings sind sie schwer zu implementieren. Durch Message-Driven-Beans wird dem Bean-Provider die Nutzung von Asynchronität und paralleler Verarbeitung bei der Implementierung von Geschäftslogik stark erleichtert.

Ein herkömmlicher JMS-Client kann auch relativ leicht und ohne viel Programmieraufwand in den Genuss von Asynchronität und paralleler Verarbeitung kommen, denn ein Messaging-Dienst verhält sich von Haus aus asynchron; und parallele Verarbeitung kann der JMS-Client durch die Benutzung mehrerer JMS-Sessions oder durch die Verwendung eines Server-Session-Pools erreichen. Ein JMS-Client ist im Vergleich zu Message-Driven-Beans jedoch nicht unbedingt eine Komponente und in den wenigsten Fällen wiederverwendbar. Eine Message-Driven-Bean kommt zudem in den Genuss von Diensten, die ihr der EJB-Container anbietet. Dazu gehören insbesondere verteilte Transaktionen.

Eine Message-Driven-Bean darf den Acknowledge-Mechanismus nicht benutzen (siehe Abschnitt 6.2.7). Dieser ist in vielen Fällen von hohem Nutzen. Auch die *LoggerBean* in Kapitel 6.5 könnte von dem Acknowledge-Mechanismus profitieren. Sie müsste dann nicht jede eingehende Nachricht protokollieren, sondern könnte die Protokollierung blockweise vornehmen, ohne dabei das Risiko einzugehen, Informationen zu verlieren. Eine blockweise Protokollierung der Daten würde die Belastung der Datenbank reduzieren. Statt des Acknowledge-Mechanismus könnte die LoggerBean beim Empfang von Nachrichten auch einfache Transaktionen der JMS-Session benutzen, um die Protokollierung auf eine sichere Art und Weise blockweise vorzunehmen. Auch das ist bei Message-Driven-Beans nicht möglich. Dazu müsste die Message-Driven-Bean Zugriff auf die JMS-Session haben, über die ihr die Nachrichten vom JMS-Provider zugestellt werden. Diese JMS-Session ist jedoch unter der Kontrolle des EJB-Containers und für die Message-Driven-Bean nicht zugänglich. Die Benutzung des Acknowledge-Mechanismus und der einfachen Transaktionen der JMS-Session ist nicht möglich, weil es keine Möglichkeit gibt, die Instanzen des Instanzenpools eines bestimmten Message-Driven-Bean-Typs zu synchronisieren. Eine bestimmte Instanz ist daher nicht in der Lage, zu entscheiden, wann ein *acknowledge* oder *commit* sinnvoll ist, da sie nicht weiß, welche bzw. wie viele Nachrichten bereits von den anderen Instanzen des Pools verarbeitet wurden.

Der EJB-Container könnte der Message-Driven-Bean z.B. über den Message-Driven-Kontext eine Möglichkeit anbieten, sich mit den anderen Instanzen des Pools zu synchronisieren und somit einfache Transaktionen des JMS-Providers oder den Acknowledge-Mechanismus zu nutzen. Der Message-Driven-Kontext könnte dazu eine Methode *acknowledge* und eine Methode *recover* enthalten, die die gleiche Wirkung

haben wie *javax.jms.Message.acknowledge()* bzw. *javax.jms.Session.recover()*. Allerdings hätte der EJB-Container durch diese Indirektion die Möglichkeit, das Acknowledgement bzw. den Commit zu überwachen und gegebenenfalls einzuschreiten.

Andererseits wäre im Fall der LoggerBean parallele Verarbeitung nicht unbedingt notwendig. *Eine* Instanz dieser Bean würde ausreichen, um die Nachrichten zu verarbeiten (moderates Nachrichtenaufkommen vorausgesetzt). Die Queue würde ohnehin bereits als sicherer Puffer fungieren. Es besteht nicht die Möglichkeit dem EJB-Container mitzuteilen, dass eine Instanz der Message-Driven-Bean ausreichend ist, um den angestrebten Zweck zu erfüllen. Die Ressourcen des EJB-Containers wären zur Laufzeit weniger belastet, da er keinen Pool von LoggerBean-Instanzen verwalten muss. Außerdem wäre die Verwendung des Acknowledge-Mechanismus oder lokaler Transaktionen der JMS-Session in diesem Fall problemlos möglich, da es keine anderen Instanzen dieser Message-Driven-Bean gibt, unter denen eine Synchronisation notwendig wäre. Die Bean-Instanz wäre in diesem Fall sehr wohl in der Lage zu entscheiden, wann ein *acknowledge* bzw. ein *commit* sinnvoll ist.

Die Spezifikation sagt nichts darüber aus, was mit Nachrichten passiert, die zwar für eine bestimmte Message-Driven-Bean bestimmt sind, von dieser aber nicht verarbeitet werden können. Die Message-Driven-Bean darf in der Methode *onMessage* weder selbstdefinierte noch System- oder Laufzeit-Exceptions werfen. Im Beispiel der *LoggerBean* in Kapitel 6.5 würde z.B. ein Fehlschlag des Datenbankzugriffs dazu führen, dass die Transaktion rückgängig gemacht wird. Der Messaging-Dienst wird in so einem Fall die Nachricht erneut zustellen. Ist das Problem, welches die korrekte Verarbeitung der Nachricht verhindert, nur von vorübergehender Dauer, kann die Verarbeitung nach einer erneuten Zustellung unter Umständen erfolgreich fortgesetzt werden. Schwierig wird es, wenn das Problem grundsätzlicher Natur oder von langfristiger Dauer ist. Beispielsweise erwartet die *LoggerBean* eine Nachricht vom Typ einer *javax.jms.TextMessage*. Würde ein Client eine Nachricht vom Typ *javax.jms.ObjectMessage* in die *LoggerQueue* einstellen (z.B wegen eines Programmierfehlers), würde das in der Methode *onMessage* der *LoggerBean* zu einer Ausnahme vom Typ *ClassCastException* führen. Die *LoggerBean* würde die Transaktion abbrechen und die Verarbeitung beenden. Das Abbrechen der Transaktion führt dazu, dass der Messaging-Dienst die Nachricht erneut zustellt. Die *LoggerBean* hat keine Möglichkeit, dem EJB-Container oder dem Messaging-Dienst mittzuteilen, dass sie wegen grundlegender Probleme die Nachricht nicht verarbeiten kann. Und auch keine der anderen Instanzen der *LoggerBean*-Klasse wird nach einer erneuten Zustellung die Nachricht korrekt verarbeiten können. In gewisser Weise steckt die Verarbeitung der Nachricht in einer Endlosschleife. Die *LoggerQueue* wäre durch diese eine fehlerhafte Nachricht blockiert und würde stetig anwachsen. Die Behandlung eines solchen Problems obliegt allein der Implementierung der Message-Driven-Bean. Zur Lösung des eben beschriebenen Problems könnte man folgende Vorgehensweise vorschlagen: Es könnte neben der

LoggerQueue eine weitere Queue, die *LoggerErrorQueue*, eingerichtet werden. Die *LoggerBean* könnte alle Nachrichten, bei denen grundlegende Probleme auftreten und eine korrekte Verarbeitung unmöglich machen, zuerst in die *LoggerErrorQueue* einstellen (quasi umleiten) und dann so tun, als sei die Nachricht korrekt verarbeitet worden. Abbildung 6.9 verdeutlicht diesen Sachverhalt.

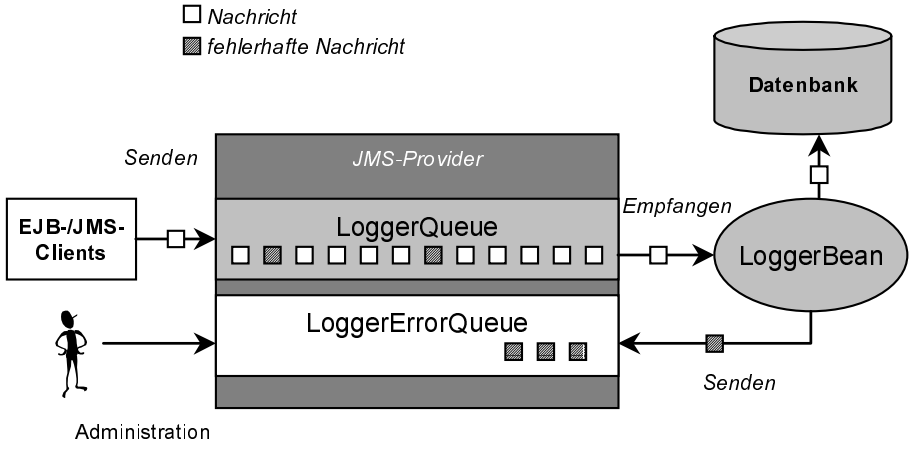

Abbildung 6.9: Fehlerbehandlung bei Message-Driven-Beans

Durch diesen Mechanismus kann die *LoggerQueue* von einer fehlerhaften Nachricht nicht blockiert werden. Die *LoggerErrorQueue* könnte als Empfänger z.B. einen normalen JMS-Client haben, der sich die Nachrichten zustellen läßt und den Administrator (beispielsweise via E-Mail) benachrichtigt. Der Administrator könnte über ein Tool den Inhalt der *LoggerErrorQueue* einsehen (z.B. mit Hilfe eines Queue-Browsers, vergleiche dazu [JMS, 1999]) und die Ursache analysieren. Nach evtl. Behebung der Ursache könnte er die Verarbeitung der Nachricht manuell anstossen und die Nachricht aus der *LoggerErrorQueue* löschen.

Aus Sicht des Bean-Providers wäre es natürlich wünschenswert, dass ein solcher oder ähnlicher Mechanismus vom EJB-Container zur Verfügung gestellt wird. Die Message-Driven-Bean könnte über das Werfen einer Ausnahme eines bestimmten Typs (z.B. *javax.ejb.EJBException*) anzeigen, dass die Nachricht wegen grundlegender Probleme nicht korrekt verarbeitet werden kann und dass die Message-Driven-Bean nicht in der Lage ist, diese Probleme zu beheben. Der EJB-Container könnte dafür sorgen, dass die Nachricht aus der Queue oder dem Topic entfernt wird, um eine sich endlos wiederholende Zustellung der Nachricht durch den Messaging-Dienst zu vermeiden. Der EJB-Container könnte die Nachricht in einen anderen persistenten Zustand überführen, z.B. in eine »Error«-Queue oder Topic umleiten oder in eine Datenbank oder in das

Dateisystem schreiben. Der EJB-Container-Provider könnte Tools zur Verfügung stellen, die den Administrator bei der Analyse und Reparatur der fehlerhaften Nachrichten unterstützen.

Vermutlich wird die nächste Version der Spezifikation der Enterprise JavaBeans im Bereich der Fehlerbehandlung und des Acknowledgements bei Message-Driven-Beans noch einige Verbesserungen bringen. Immerhin ist der Typ der Message-Driven-Bean noch sehr neu und lässt viel Spielraum für Optimierungen. Aus der praktischen Erfahrung im Umgang mit Message-Driven-Beans werden sicherlich noch weitere Verbesserungsansätze gewonnen werden.

7 Transaktionen

7.1 Grundlagen

Transaktionen sind ein vielseitig wertvoller Mechanismus und eine Stärke des EJB-Konzepts. Erst sie ermöglichen die Abwicklung wichtiger Geschäftsprozesse im Mehrbenutzerbetrieb.

Transaktionen fassen mehrere Arbeitsschritte zu einer Arbeitseinheit zusammen. Eine Transaktion ist der Weg von einem konsistenten Zustand in einem Datenbestand zu einem anderen konsistenten Zustand. Eine erfolgreich durchgeführte Transaktion führt zu einer dauerhaften Änderung der Daten. Eine typische Anwendung ändert den Inhalt einer Datenbank mit einer Transaktion.

Der Weg zum nächsten konsistenten Zustand führt oft über mehrere Arbeitsschritte und inkonsistente Zwischenzustände. Deshalb hat eine Transaktion die Aufgabe, einzelne Arbeitsschritte zu einer Einheit zusammenzufassen. Tritt in einer Transaktion eine Bedingung ein, die die vollständige Durchführung verhindert, so werden alle bis zu diesem Zeitpunkt ausgeführten Aktionen rückgängig gemacht. Das heißt, eine Transaktion wird entweder vollständig durchgeführt, oder sie hinterlässt keinen Effekt im Datenbestand. Deshalb wird eine Transaktion *atomar* oder *unteilbar* genannt.

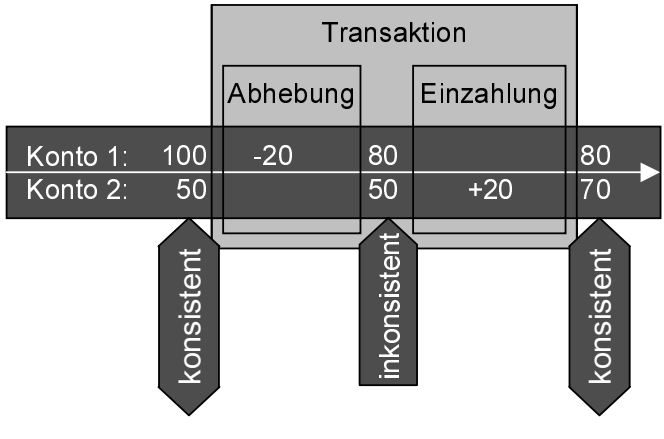

Abbildung 7.1: Buchung als Beispiel für eine Transaktion

Ein typisches Beispiel für eine Transaktion ist eine Buchung (vgl. Abbildung 7.1). Dabei wird ein bestimmter Betrag von einem Konto auf ein anderes übertragen. Tritt bei der Abhebung oder bei der Einzahlung ein Fehler auf, so muss der ursprüngliche Zustand als letzter konsistenter Stand wiederhergestellt werden.

Andere Benutzer der gleichen Daten sollen möglichst immer konsistente Zustände sehen. Deshalb kann der Transaktionsmechanismus dafür sorgen, dass ihnen inkonsistente Zwischenstände verborgen bleiben. Diese Eigenschaft wird als *Sichtenkonsistenz* oder *Isolation von Transaktionen* bezeichnet.

Beim parallelen Zugriff mehrerer Clients auf die gleichen Daten gibt es drei Kategorien von Problemen. Es kann somit drei unterschiedliche Gründe geben, warum ein Client einen inkonsistenten Zwischenzustand sieht.

1. *Inkonsistenz (Dirty Reads)*: Es besteht zu einem bestimmten Zeitpunkt ein inkonsistenter Zustand im Datenbestand (siehe Abbildung 7.1).

2. *Nichtwiederholbares Lesen (Nonrepeatable Reads)*: Ein Client liest die Daten zeitlich verteilt, den ersten Teil zum Zeitpunkt A und den zweiten Teil zum Zeitpunkt B. Eine parallel ablaufende Transaktion eines anderen Clients modifiziert die gleichen Daten in der Zeitspanne zwischen diesen beiden Lesezugriffen. Auch wenn zu den Zeitpunkten A und B jeweils ein konsistenter Zustand herrschte, so kann der lesende Client doch eine fehlerhafte Sicht erhalten.

   In dem in Abbildung 7.1 gegebenen Beispiel könnte die inkonsistente Sicht wie folgt entstehen: Ein anderer Client liest vor der dargestellten Transaktion das Konto 1 und nach der Transaktion das Konto 2. Der Client erhält die fehlerhafte Sicht Konto 1 = 100 und Konto 2 = 70.

3. *Phantom-Problem (Phantom Reads)*: Dieses Problem entsteht durch neu eingefügte Datenelemente. Ein Client liest zum Zeitpunkt A alle Datenelemente, die bestimmte Kriterien erfüllen, und führt zum Zeitpunkt B eine schreibende Operation aus, die auf dem Ergebnis des Zeitpunkts A basiert. Wenn zwischen den Zeitpunkten A und B ein neues Datenelement hinzukommt, das die gleichen Kriterien erfüllt, so hätte die schreibende Operation zum Zeitpunkt B dieses berücksichtigen müssen.

   Dieses Problem kann beispielsweise bei einer Lagerverwaltung auftreten. Ein Artikel mit einem gegebenen Gewicht soll gelagert werden. Dazu wird zuerst ermittelt, ob das Lager noch die benötigte Aufnahmekapazität hat. Die enthaltenen Artikel werden bestimmt und die Gewichte summiert. Wenn die Kapazität ausreicht, wird der Artikel in die Liste der enthaltenen Artikel aufgenommen. Wenn zwischen dem Lesen der enthaltenen Artikel und dem Aufnehmen des Artikels in das Lager ein anderer Artikel in das Lager aufgenommen wird, kann die Gesamtkapazität überschritten werden.

Heutige Datenbanksysteme können diese Probleme vermeiden. Durch die Zwischenspeicherung von Daten für eine laufende Transaktion (engl. *caching*) und durch das exklusive Reservieren von Datenelementen (engl. *locking*) für eine Transaktion kann jeder Transaktion stets eine konsistente Sicht geboten werden. Beide Maßnahmen kosten jedoch Ressourcen und verringern deshalb die Performanz der Datenbank.

Das exklusive Reservieren beeinträchtigt die Parallelität der Transaktionen und kann im Extremfall zur Serialisierung aller Transaktionen führen. Eine vollständige Sichtenkonsistenz ist somit teuer, d.h. sie hat negative Auswirkungen auf die Performanz und Parallelität der Transaktionen. Aus diesem Grunde werden teilweise Einschränkungen akzeptiert.

Man unterscheidet vier Konsistenzebenen (vergleiche [Lang 1995], siehe auch Abschnitt *Transaktionsisolation* in diesem Kapitel). Die Qualität der Konsistenz steigert sich von Ebene 1 zu Ebene 4:

1. Die Probleme Inkonsistenz und nichtwiederholbares Lesen werden zugunsten der Performanz akzeptiert.
2. Nur das Problem des nichtwiederholbaren Lesens wird akzeptiert. Im Gegensatz zu Ebene 1 darf hier keine Inkonsistenz mehr auftreten.
3. Das konsistente Ergebnis einer Transaktion ist durch das Phantom-Problem gefährdet.
4. Vollständige Sichtenkonsistenz ist immer gewährleistet.

Die grundlegenden Eigenschaften von Transaktionen werden häufig unter dem englischen Akronym ACID zusammengefasst. ACID steht für *Atomic* (atomar), *Consistent* (konsistent), *Isolated* (isoliert) und *Durable* (langlebig).

Wir unterscheiden zwischen der Steuerung und der Regelung von Transaktionen. Die Transaktionssteuerung (*Transaction Demarcation* oder *Transaction Management*) bestimmt, wann eine Transaktion beginnt und wann sie abgeschlossen bzw. abgebrochen wird. Die Transaktionsregelung ist die technische Umsetzung des Transaktionsmechanismus. Sie besteht aus dem Kommunikationsmechanismus für verteilte Transaktionen und dem Transaktionsmonitor.

Im Vergleich zu dem theoretischen Wissen, das für die Anwendung von Transaktionen notwendig ist, ist die Programmierung der Transaktionssteuerung sehr einfach. Drei Befehle bilden die Basis für die Transaktionssteuerung:

- *begin* – startet eine Transaktion.
- *commit* – veranlasst den Abschluss einer Transaktion.
- *rollback* – veranlasst den fehlerbedingten Abbruch einer Transaktion.

7.2 Konzepte

7.2.1 Überblick

Die EJB-Architektur realisiert Transaktionen mit Hilfe eines Transaktionsservice (siehe Abschnitt 7.2.2). Dieser arbeitet als zentrale Instanz, die die Koordination aller Transaktionsteilnehmer übernimmt.

Die Transaktionsteilnehmer sind ein Client, ein oder mehrere transaktionale Systeme, eine oder mehrere Beans und der EJB-Container. Der Client ist die Schnittstelle zum Benutzer und steuert meist die Ausführung der Geschäftsprozesse. Die transaktionalen Systeme speichern die Daten, die in der Transaktion verändert werden. Das am häufigsten verwendete transaktionale System ist eine Datenbank. Die Beans basieren normalerweise auf den Daten in den transaktionalen Systemen. Die Konsistenz ihrer Zustände wird durch die Transaktion gewährleistet. Der EJB-Container arbeitet eng mit dem Transaktionsservice zusammen. Er stellt den Beans eine Umgebung zur Verfügung, die den einfachen Umgang mit Transaktionen besonders unterstützt.

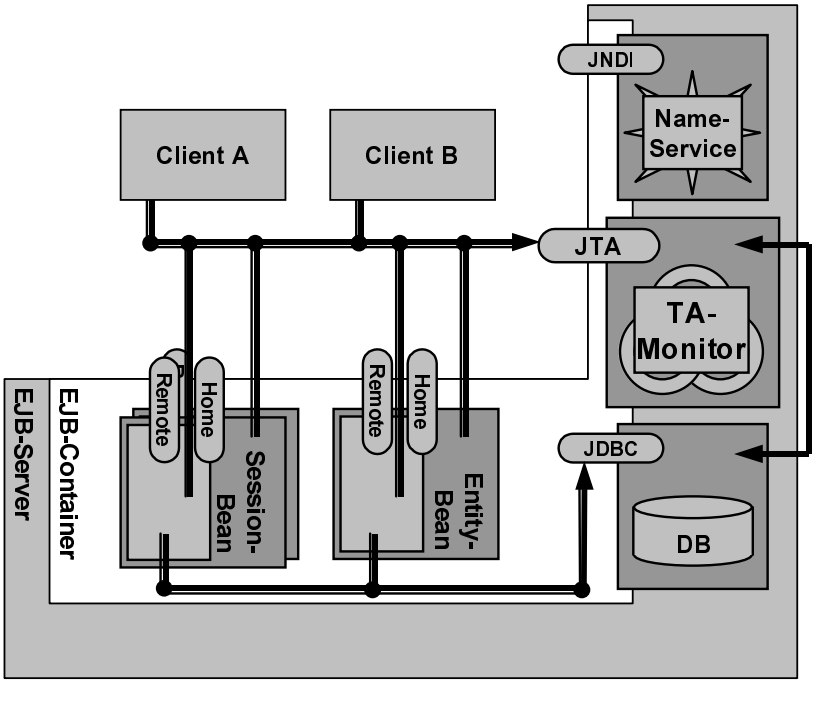

Abbildung 7.2: Kommunikationswege bei EJB-Transaktionen

Die Transaktionssteuerung kann der Client, die Bean oder auch der EJB-Container übernehmen. In letzterem Fall wird von deklarativen Transaktionen gesprochen, da die Transaktionssteuerung nicht in der Anwendungslogik programmiert, sondern erst beim Deployment deklariert wird. Dadurch wird u.a. die Wiederverwendung von Beans gefördert, da für eine Bean unterschiedliche transaktionale Verhaltensweisen deklariert werden können. Somit lassen sich Beans leichter in unterschiedlichen Applikationen einsetzen.

Für die Transaktionssteuerung ist es unerheblich, ob die Enterprise-Bean den Remote- oder den Local-Client-View oder beide unterstützt. Die in diesem Kapitel dargestellten Sachverhalte gelten uneingeschränkt für beide Fälle. Deswegen wird in diesem Kapitel keine Unterscheidung zwischen Remote- und Local-Client-View getroffen. Die Ausführungen beziehen sich auf den Standardfall, den Remote-Client-View.

Die Rollenverteilung des EJB-Konzepts bestimmt die Prinzipien für Steuerung und Regelung der Transaktionen. Die Transaktionsregelung wird vollständig vom Hersteller des EJB-Containers behandelt. Der Bean-Entwickler ist davon gänzlich entbunden. Diese Strategie hat sich schon bei den klassischen Datenbanksystemen bewährt. Bei der Transaktionssteuerung geht das EJB-Konzept aber einen Schritt weiter. Mit den deklarativen Transaktionen kann der Anwendungsentwickler von den Problemen der Transaktionssteuerung entlastet werden. Diese Aufgabe wird auf die Experten verlagert, die den Überblick für das Zusammenstellen von Applikationen haben und das erforderliche transaktionale Verhalten besser bestimmen können.

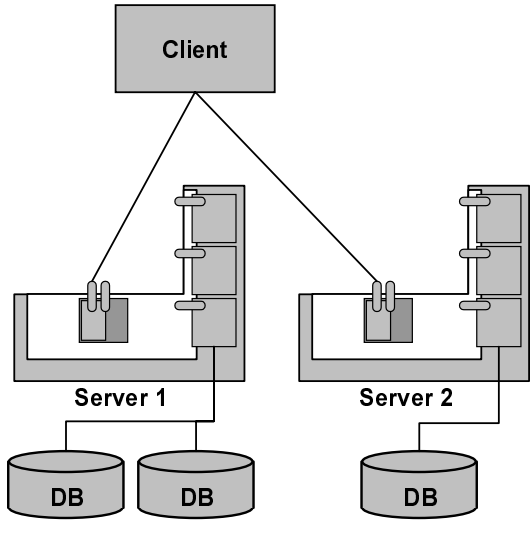

Abbildung 7.3: Beispiel für an einer verteilten Transaktion beteiligten Systeme

EJB unterstützt verteilte Transaktionen. Von einer verteilten Transaktion spricht man auch dann, wenn mehrere transaktionale Objekte verschiedener Dienste (z.B. JMS, Datenbank, EJB-Container) an einer Transaktion beteiligt sind. Die verschiedenen, an der Transaktion beteiligten Dienste können auf dem selben Server laufen oder auf verschiedenen Servern. Mit einer verteilten Transaktion können z.B. Daten aus unterschiedlichen Datenbanken in einer Transaktion verändert werden. Eine Transaktion kann sowohl unterschiedliche Datenbanken eines EJB-Containers als auch unterschiedliche EJB-Container betreffen.

Die Sichtenkonsistenz (Isolation von Transaktionen) fällt nur teilweise in den Zuständigkeitsbereich des EJB-Containers. Bei der Konfiguration des Applikations-Servers für die Datenbankverbindung kann bei den meisten Produkten die Konsistenzebene angegeben werden. Der Applikations-Server könnte außerdem einen nicht standardisierten Dienst zur Verfügung stellen, um das Setzen der Konsistenzebene in der Anwendungslogik zu ermöglichen.

7.2.2 JTA und JTS

Java Transaction API (JTA) und *Java Transaction Service* (JTS) sind Spezifikationen von Sun Microsystems. JTA definiert die Schnittstellen, die die beteiligten Instanzen einer verteilten Transaktion zur Kommunikation benötigen. Die Schnittstellen befinden sich auf einem genügend hohen Abstraktionsniveau, so dass sie für Anwendungsprogrammierer gut geeignet sind.

JTS ist für die Anbieter von Transaktionsmonitoren (engl. *Transaction Service*) gedacht. Ein Transaktionsmonitor regelt die Transaktionen. Er koordiniert die Abwicklung von Transaktionen in einem Verbund unterschiedlicher transaktionaler Systeme. JTS beinhaltet die nötigen Definitionen, um einen Transaktionsmonitor zu entwickeln, der JTA unterstützt.

JTS ist die Verbindung zwischen Java und dem *Object Transaction Service 1.1* (OTS). Hierbei handelt es sich um einen Standard der *OMG* (*Object Management Group*), der die Transaktionssteuerung und Regelung mit CORBA-Objekten definiert. JTS ist eine kompatible Untermenge zu OTS.

Die Programmierung der folgenden Beispiele zu Transaktionen basiert ausschließlich auf einem Teil von JTA. Der komplexe Mechanismus der Transaktionsregelung ist für den Bean-Entwickler unsichtbar. Auch wenn dieses sehr einfache Interface die Komplexität verbirgt, sollten Sie die genauen Vorgänge kennen, um effiziente Programme zu schreiben.

Im Folgenden soll ein Überblick über JTS gegeben werden. Die hier benutzten Begriffe bilden die Basis für die folgende Darstellung der Transaktionen von EJBs. Danach gilt unser besonderes Augenmerk den Kommunikationsstrecken bei der Verwendung von Transaktionen.

Die vollständigen Spezifikationen von JTA und JTS können auf den Web-Seiten von Sun gefunden werden:

- *http://java.sun.com/products/jta* [Sun Microsystems, JTA, 1999]
- *http://java.sun.com/products/jts* [Sun Microsystems, JTS, 1999]

Begriffe und Konzepte

JTS und OTS unterscheiden Objekttypen nach ihrer Rolle in Transaktionen. In den unterschiedlichen Objekttypen spiegelt sich auch das EJB-Konzept wieder.

- *Transaktionale Clients* (*transactional clients*) sind Programme, die auf Objekte auf dem Server zugreifen. Sie können die Transaktionssteuerung übernehmen. Selbst speichern sie keine Daten, die transaktionsgesichert verarbeitet werden. Nicht alle Clients, die mit EJBs arbeiten, sind transaktionale Clients. Mit deklarativen Transaktionen ist es möglich, den Client vollständig von der Transaktionssteuerung zu entbinden.
- *Transaktionale Objekte* (*transactional objects*) sind Objekte auf dem Server, die an Transaktionen teilnehmen, jedoch keine eigenen Daten transaktionsgesichert speichern. Sie übernehmen die Transaktionssteuerung nicht im eigentlichen Sinn, können jedoch einen Abbruch der Transaktion erzwingen. Alle Enterprise JavaBeans sind transaktionale Objekte.
- *Wiederherstellbare Objekte* (*recoverable objects*) sind die Objekte, die die eigentlichen Daten speichern. Der Zugriff erfolgt immer transaktionsgesichert. Im EJB-Modell verwaltet die Datenbank die meisten wiederherstellbaren Objekte. Ein wiederherstellbares Objekt kann dann eine Zeile in einer Datenbanktabelle sein. Auch andere Services, wie der Java Messaging Service (JMS) können mit wiederherstellbaren Objekten umgehen.

Außerdem werden zwei grundlegende Transaktionsarten unterschieden:

- *Lokale Transaktionen* behandeln nur Daten aus einer Datenbank oder einem anderen transaktionalen System. Die Koordination kann das wiederherstellbare Objekt ohne einen eigenständigen Transaktionsservice leisten.
- *Globale Transaktionen* können Daten von unterschiedlichen Datenbanken bzw. anderen transaktionalen Systemen behandeln. Die Koordination übernimmt ein eigenständiger Transaktionsservice, der die beteiligten Systeme synchronisiert.

Eine Methode, die in einer globalen Transaktion ausgeführt wird, hat einen *Transaktionskontext*. Der Transaktionskontext beschreibt die eindeutige Identität der Transaktion, ihren Zustand und eventuell weitere Attribute.

Der *Transaktionsservice* koordiniert alle globalen Transaktionen. Er stellt sicher, dass alle Operationen auf wiederherstellbaren Objekten transaktionsgesichert sind. Um dies zu ermöglichen, muss jedes an einer Transaktion beteiligte Objekt seinen Transaktionskontext kennen. Der Transaktionsservice assoziiert deshalb den Transaktionskontext mit dem Thread. Dadurch steht jedem Objekt im gleichen Thread der Transaktionskontext zur Verfügung.

Bei einem Aufruf einer Methode über eine Netzwerk- oder Prozessgrenze hinweg ist jedoch ein zusätzliches Konzept nötig. Die *Vererbung des Transaktionskontextes* stellt sicher, dass auch bei diesen Aufrufen der Transaktionskontext weitergegeben wird. Die Realisierung obliegt dem Applikations-Server und basiert meist auf einer Erweiterung des Netzwerkprotokolls (RMI / IIOP).

Arbeitsweise von JTS

Abbildung 7.4 stellt die Interaktion der an einer Transaktion beteiligten Objekte schematisch dar.

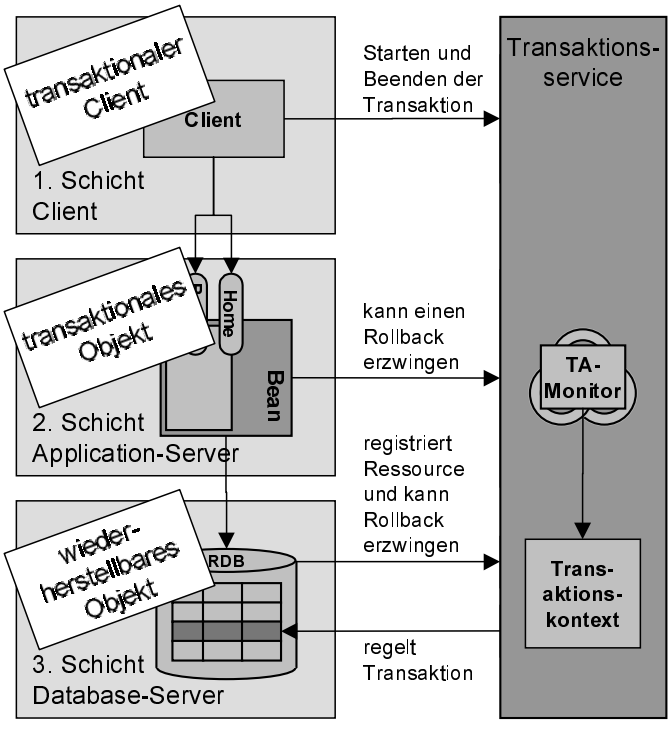

Abbildung 7.4: JTS und EJB

- Ein zentrales System, der Transaktionsservice, koordiniert alle Transaktionen. Er kommuniziert direkt mit allen Objekten, die an einer Transaktion beteiligt sind (transaktionale Clients, transaktionale Objekte und wiederherstellbare Objekte).
- Zu Beginn einer Transaktion wird der Transaktionskontext von dem transaktionalen Client erzeugt. Dieser enthält Informationen über die Transaktion und den Benutzer. Der Transaktionskontext wird in Java mit dem laufenden Thread assoziiert. Einerseits bedeutet das, dass alle Aktionen dieses Threads in der Transaktion ausgeführt werden, und andererseits, dass es immer nur eine Transaktion je Thread geben kann.
- Jede neue Transaktion wird beim Transaktionsservice angemeldet. Der Transaktionsservice kennt den Transaktionskontext und den Zustand jeder laufenden Transaktion.
- Der transaktionale Client ruft während einer Transaktion ein- oder mehrmals Methoden von transaktionalen Objekten auf dem Server auf. Die Methoden der transaktionalen Objekte rufen ihrerseits weitere Methoden auf, entweder von weiteren transaktionalen Objekten oder von wiederherstellbaren Objekten.
- Jedes Objekt, das durch eine beliebige Aufrufkette an einer Transaktion beteiligt ist, muss den Transaktionskontext erfahren. Solange die Aufrufe auf einem System bleiben, ist dies kein Problem, da der Transaktionskontext mit dem Thread assoziiert ist und EJB das Starten neuer Threads verbietet. Muss jedoch eine Netzwerkstrecke überwunden werden, so ist eine entsprechende Protokollerweiterung nötig, die zusätzlich zu den Daten des Funktionsaufrufs auch den Transaktionskontext überträgt. Dies wird *Vererbung des Transaktionskontexts* genannt (*transactional context propagation*).
- Jedes wiederherstellbare Objekt, das durch eine beliebige Aufrufkette an einer Transaktion beteiligt ist, benutzt den Transaktionskontext, um beim Transaktionsservice seine Daten zu registrieren. Es macht dem Transaktionsservice die lokale Transaktion bekannt, die es für den Zugriff verwendet. Damit kennt der Transaktionsservice alle Daten, die an einer laufenden Transaktion beteiligt sind.
- Am Ende der Transaktion signalisiert der transaktionale Client einen erfolgreichen Abschluss (*commit*) oder einen Abbruch (*rollback*) der Transaktion. Soll die Transaktion erfolgreich abgeschlossen werden, so müssen alle beteiligten Objekte ihr Einverständnis geben. Die Transaktion kann nicht erfolgreich abgeschlossen werden, wenn ein wiederherstellbares Objekt einen Fehler beim Zugriff auf seine Daten hatte oder ein transaktionales System den Status der Transaktion zuvor auf »*rollback only*« gesetzt hat.
- Der Transaktionsservice koordiniert den Abschluss einer Transaktion mit dem so genannten Zwei-Phasen-Commit-Protokoll (*two-phase-commit*). Er gewährleistet, dass entweder alle lokalen Transaktionen erfolgreich abgeschlossen oder abgebrochen werden.

Programmierung von JTA

Das *Java Transaction API* (JTA) dient zur Transaktionssteuerung. Bei JTA handelt es sich um den Ausschnitt aus der Schnittstelle des Java Transaction Service (JTS), der für den regulären Applikationsentwickler relevant ist. Da dieses Konzept bei älteren Applikations-Servern noch nicht berücksichtigt wird, muss teilweise auch der Zugriff über JTS benutzt werden.

import-Anweisung für JTA:

```
import javax.transaction.*;
```

import-Anweisung für JTS:

```
import javax.jts.*;
```

Die folgenden Beispiele zu Transaktionen verwenden ausschließlich das Interface *javax.transaction.UserTransaction*.

7.3 Implizite Transaktionssteuerung

7.3.1 Einführung

Die implizite Transaktionssteuerung stellt einen sehr einfachen Zugang zu Transaktionen dar. Der Entwickler der Anwendungslogik muss ein Experte für einen bestimmten Anwendungsbereich sein. Normalerweise wird er nicht gleichzeitig ein Experte für Transaktionen sein. Deshalb soll ihm die Komplexität der Transaktionssteuerung und -regelung weitgehend verborgen bleiben.

Die implizite Transaktionssteuerung basiert auf der Trennung der eigentlichen Anwendungslogik von der Logik für die Transaktionssteuerung. Der Bean-Provider entwickelt die Anwendungslogik, während der Application-Assembler beim Zusammenstellen der vollständigen Applikation die Transaktionssteuerung mittels der Transaktionsattribute definiert.

Bei impliziten Transaktionen übernimmt der EJB-Container die Transaktionssteuerung. Für jeden Aufruf einer Methode einer Bean stellt der EJB-Container sicher, dass eine passende Transaktion existiert. Im Deployment-Deskriptor wird für jede Methode definiert, ob eine Transaktion benötigt wird, und wenn ja, von welcher Art sie sein muss. Bei einem Aufruf mit einer unpassenden oder fehlenden Transaktion versucht der EJB-Container, eine passende Transaktion zu erzeugen. Falls dies nicht möglich ist, verursacht diese Operation einen Fehler, und der EJB-Container löst eine Exception (Ausnahme) aus.

Abbildung 7.5 zeigt ein Beispiel für eine implizite Transaktion. Der Client ruft eine Methode einer Bean auf. Dabei hat der Client keine Verantwortung für die Transaktionssteuerung. Die Ausführung der aufgerufenen Methode erfordert jedoch eine Transaktion. Weil dies dem EJB-Container mit dem Deployment-Deskriptor der Bean mitgeteilt wurde, kann der EJB-Container die nötige Transaktion starten. Nach der erfolgreichen Ausführung der Methode wird der EJB-Container die Transaktion mit einem Commit beenden. Im Fehlerfall, wenn eine Exception aufgetreten ist, bricht der EJB-Container die Transaktion mit einem Rollback ab.

Die aufgerufene Methode kann ihrerseits weitere Methoden aufrufen. Der Transaktionskontext wird dabei weitergegeben. Der EJB-Container prüft, ob eine geeignete Transaktion für die aufgerufene Methode existiert. Im Beispiel kann die bestehende Transaktion weiterverwendet werden.

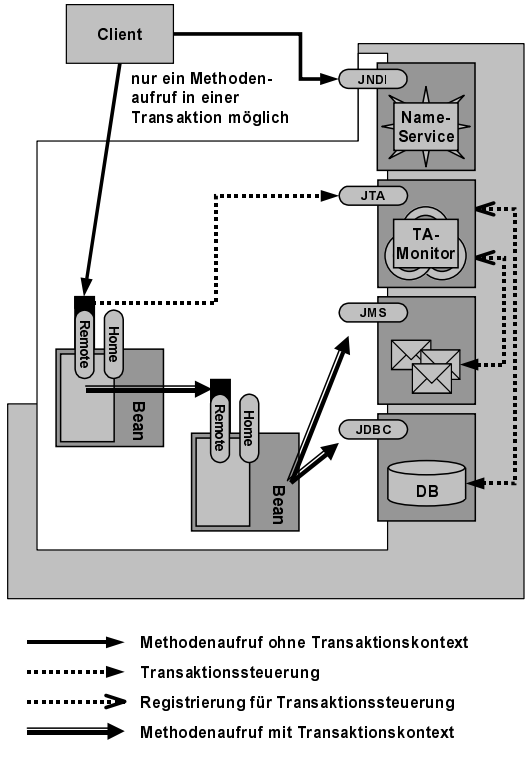

Abbildung 7.5: Beispiel für Datenflüsse bei impliziten Transaktionen

Eine aufgerufene Methode kann auch einen definierten Service der EJB-Architektur verwenden. Beispiele sind der Zugriff auf eine Datenbank oder den Messaging-Service. Eine Entity-Bean verwendet eventuell implizit die Datenbank durch das Spei-

chern ihrer persistenten Daten. Diese transaktionalen Systeme verwalten die von der Transaktion betroffenen Daten. Für die korrekte Verwaltung dieser Daten verwenden sie den übertragenen Transaktionskontext und kommunizieren direkt mit dem Transaktionsservice über JTS.

Aus der besonderen Zielsetzung von impliziten Transaktionen, nämlich der Reduzierung der Komplexität für den Anwendungsentwickler, leiten sich besondere Eigenschaften ab. Die folgende Auflistung soll einen Überblick geben. Die einzelnen Punkte werden aber auch in den folgenden Abschnitten anhand konkreter Programmbeispiele vertieft.

- Implizite Transaktionen sind deklarative Transaktionen. Das transaktionale Verhalten wird im Deployment-Deskriptor deklariert.
- Im Deployment-Deskriptor wird das transaktionale Verhalten einer Bean definiert. Für jede Methode kann ein Transaktionsattribut vergeben werden (siehe Abschnitt 7.3.3).
- Eine implizite Transaktion kann sich nicht über mehr als einen Funktionsaufruf des Clients erstrecken. Im Regelfall ist das jedoch ausreichend, wenn auf der Seite des Servers die entsprechenden Schnittstellen zur Verfügung stehen.
- Entsteht in einer Bean eine Fehlersituation, so kann die Bean erzwingen, dass die laufende Transaktion beim Abschluss rückgängig gemacht wird. Die Bean hat jedoch keine Möglichkeit, die laufende Transaktion zu beenden oder eine neue Transaktion zu starten.
- Eine Session-Bean kann sich vom EJB-Container über die Zustände einer Transaktion benachrichtigen lassen und hat damit die Möglichkeit zusätzliche Konsistenzbedingungen sicherzustellen.

Diese besonderen Eigenschaften der impliziten Transaktionen führen natürlich zu einigen Einschränkungen. Das EJB-Konzept ist jedoch maßgeblich durch die Aufteilung der Verantwortlichkeiten auf unterschiedliche Rollen geprägt. Implizite Transaktionen sind ein wichtiger Schritt, um dieses Ziel zu erreichen, und wurden deshalb sehr gezielt als ein Kernbestandteil des EJB-Konzepts gewählt.

Wer bestmöglich von den Vorteilen der EJB-Architektur profitieren möchte, sollte den Einsatz impliziter Transaktionen forcieren. Der Einsatz der in Abschnitt 7.4 behandelten expliziten Transaktionssteuerung bedeutet immer die teilweise Aufgabe der Rollenverteilung. Die Logik der Transaktionssteuerung wird bei der Implementierung definiert und ist im Deployment-Deskriptor nicht ersichtlich. Die Entscheidung für die explizite Transaktionssteuerung ist eine weitreichende Designentscheidung und sollte die Ausnahme bleiben.

Die folgende Gegenüberstellung der Vorteile und der Beschränkungen impliziter Transaktionen soll die Wahl des richtigen Konzepts für einen Anwendungsfall erleichtern:

Vorteile

- *Qualität und Wartung*: Der Entwickler der Anwendungslogik ist von der Komplexität der Transaktionssteuerung befreit. Dies hat einen doppelt positiven Einfluss auf die Qualität und Wartbarkeit. Jeder Mitarbeiter im Entwicklungsprozess macht, was er am besten kann, und die Komplexität wird für jeden geringer.
- *Wiederverwendbarkeit*: Die Wiederverwendbarkeit einer Bean ist wesentlich besser, da erst zum Zeitpunkt des Deployments das transaktionale Verhalten dieser Komponente bestimmt werden muss.

Beschränkungen

- *Eingeschränkte Flexibilität*: Die Möglichkeiten bei einer Entwicklung rein auf der Seite des Clients sind begrenzt. Mit deklarativen Transaktionen kann der Client die Beans auf dem Server nicht zu neuen Transaktionen kombinieren. Auch gibt es Anwendungsfälle, bei denen die Anwendungslogik der Bean eine bestimmte Transaktionssteuerung verlangt und somit die Trennung ungünstig ist. Aufgrund dieser Einschränkungen sieht das EJB-Konzept auch die explizite Transaktionssteuerung vor (siehe Abschnitt 7.4).
- *Einfache Aufgaben komplex*: Die Rollenverteilung und das daraus resultierende Vorgehensmodell machen die Entwicklung einfacher Prototypen unnötig kompliziert. Das Konzept zahlt sich erst bei größeren Projekten aus.

7.3.2 Beispiel *Producer*

Den leichten Einstieg in die Programmierung soll das Beispiel *Producer* ermöglichen. Es handelt sich um die einfache Simulation einer Versorgungskette. Die Session-Bean *Producer* ist ein Produzent, der aus bestimmten Rohstoffen ein Produkt fertigt. Die Entity-Bean *Stock* simuliert die Lager für die Rohstoffe und das Endprodukt.

Producer ist ein klassisches Beispiel für Transaktionen. Es kann nur produziert werden, wenn die nötigen Rohstoffe vorhanden sind und das Ausgangslager die produzierte Menge fassen kann. Solange es nur einen Client gibt, ist diese Forderung trivial. Durch den Einsatz von Transaktionen bleibt die Forderung auch für parallele Zugriffe mehrerer Clients trivial.

Eine wichtige Methode für den Umgang mit Transaktionen ist *setSessionContext* bei Session-Beans, *setMessageDrivenContext* bei Message-Driven-Beans bzw. *setEntityContext* bei Entity-Beans. Hier wird der Bean ihr Kontext bekannt gegeben, der Informa-

tionen über den Benutzer und die Transaktion enthält. Außerdem gibt der Kontext der Bean begrenzte Möglichkeiten, die Transaktion zu beeinflussen. Die Methode *setRollbackOnly* erzwingt einen Rollback. Mit *getRollbackOnly* kann abgefragt werden, ob die Transaktion noch erfolgreich abgeschlossen werden kann.

Zuerst soll die Entity-Bean *Stock* betrachtet werden. Der Konstruktor verlangt drei Parameter: eine eindeutige ID für jede Stock-Bean, das maximale Fassungsvermögen und den aktuellen Füllstand. Außerdem definiert das Home-Interface (Siehe Listing 7.1) die nötige *findByPrimaryKey*-Methode.

```
package ejb.supplychain.stock;

import javax.ejb.CreateException;
import javax.ejb.EJBHome;
import javax.ejb.FinderException;
import java.rmi.RemoteException;

public interface StockHome extends EJBHome {

    public Stock create(String stockId,
                        int    maxVolume,
                        int    aktVolume)
        throws CreateException,
               RemoteException;

    public Stock findByPrimaryKey(String primaryKey)
        throws FinderException,
               RemoteException;

}
```

Listing 7.1: StockHome – Home-Interface der Entity-Bean Stock

Im Remote-Interface (vgl. Listing 7.2) sehen wir die Funktionalität der *Stock*-Bean. Mit der Methode *get* kann Material aus dem Lager genommen werden, und mit der Methode *put* wird Material hineingelegt. Beide Methoden lösen eine *ProcessingErrorException* aus, wenn die Aktion nicht möglich ist. Die Methode *getVolume* ermöglicht, den aktuellen Füllstand des Lagers abzufragen.

```
package ejb.supplychain.stock;

import java.rmi.RemoteException;
import javax.ejb.EJBObject;

public interface Stock extends EJBObject {

    public void get(int amount)
```

```
        throws ProcessingErrorException, RemoteException;

    public void put(int amount)
        throws ProcessingErrorException, RemoteException;

    public int  getVolume()
        throws RemoteException;

}
```

Listing 7.2: Stock – Remote-Interface der Entity-Bean Stock

Nachdem jetzt die Funktionalität der Bean definiert ist, können wir in Listing 7.3 das Datenbankschema festlegen.

```
CREATE TABLE STOCK
  (STOCKID   VARCHAR(10) NOT NULL,
   VOLUME    INT,
   MAXVOLUME INT)
ALTER TABLE STOCK ADD
  CONSTRAINT PRIMKEY
  PRIMARY KEY (STOCKID)
```

Listing 7.3: Datenbankschema für die Entity-Bean Stock

Die Implementierung der Methoden soll beispielhaft an der Methode *get* gezeigt werden (siehe Listing 7.4). Die Methode setzt voraus, dass sie in einer Transaktion ausgeführt wird. Zuerst wird die neue Stückzahl im Lager berechnet, und dann wird geprüft, ob es sich um einen gültigen Wert handelt.

Bei einem gültigen Wert wird die persistente Variable *stockVolume* geändert. Der EJB-Container stellt sicher, dass diese Änderung transaktionsgesichert in die Datenbank geschrieben wird.

Ein ungültiger Wert bedeutet für die Methode einen Fehler. Auf einen Fehler kann in einer Transaktion immer auf zwei unterschiedliche Weisen reagiert werden:

- *Exception*: Die Methode löst eine Exception (Ausnahme) aus und überlässt damit die Entscheidung, ob die Transaktion rückgängig gemacht werden soll, dem Aufrufer. Wird bei deklarativen Transaktionen die Exception von keiner Bean in der Aufrufhierarchie abgefangen, so bricht der EJB-Container die Transaktion mit einem Rollback ab.
- *setRollbackOnly*: Der Aufruf dieser Methode am Bean-Kontext veranlasst den Abbruch der Transaktion mit einem Rollback. Dies ist z. B. sinnvoll, wenn schon Daten in der Datenbank verändert worden sind und dadurch ein ungültiger Zwi-

schenzustand besteht. Der Rollback wird nicht sofort ausgeführt, sondern erst beim Beenden der Transaktion mit Commit oder Rollback. Die Entscheidung für einen Rollback ist endgültig und macht Maßnahmen für die Wiederherstellung eines gültigen Zustands in dieser Transaktion sinnlos. Um die weitere Bearbeitung der Transaktion zu verkürzen, kann zusätzlich eine Exception ausgelöst werden.

In der *Stock*-Bean werden die Fehler durch das Prüfen der Vorbedingungen aufgedeckt, bevor persistente Daten geändert werden. Deshalb ist ein *setRollbackOnly* überflüssig. Es wird eine Exception ausgelöst und der Aufrufer kann darauf reagieren.

```
...
    public void get(int amount)
        throws ProcessingErrorException
    {
        int newStockVolume = this.getStockVolume() - amount;
        if(newStockVolume >= 0) {
            this.setStockVolume(newStockVolume);
        } else {
            throw new ProcessingErrorException("volume to small");
        }
    }
...
```

Listing 7.4: StockBean.get – Methode verringert den Lagerbestand

Die entscheidende Frage ist jetzt, wie sichergestellt wird, dass die Methoden immer in einer Transaktion ausgeführt werden. Dafür können im Deployment-Deskriptor Attribute für Transaktionen zu jeder Methode angegeben werden. Der Wert *Required* sorgt immer für eine Transaktion. Entweder wird die laufende Transaktion verwendet oder vom EJB-Container eine neue Transaktion gestartet. Abschnitt 7.3.3 gibt eine vollständige Beschreibung aller zulässigen Werte. Listing 7.5 zeigt die für die Stock-Bean relevanten Teile aus dem Deployment-Deskriptor.

```
<?xml version="1.0" ?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0/
/EN" "http://java.sun.com/j2ee/dtds/ejb-jar_2_0.dtd">

<ejb-jar>
  ...
  <enterprise-beans>
    <entity>
      <ejb-name>Stock</ejb-name>
      <home>ejb.supplychain.stock.StockHome</home>
      <remote>ejb.supplychain.stock.Stock</remote>
      <ejb-class>ejb.supplychain.stock.StockBean</ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>java.lang.String</prim-key-class>
```

```
      <reentrant>False</reentrant>
      <cmp-version>2.x</cmp-version>
      <abstract-schema-name>StockBean</abstract-schema-name>
      <cmp-field>
        <field-name>stockId</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>stockVolume</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>maxStockVolume</field-name>
      </cmp-field>
      <primkey-field>stockId</primkey-field>
    </entity>
    ...
  </enterprise-beans>
  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>Stock</ejb-name>
        <method-name>get</method-name>
      </method>
      <method>
        <ejb-name>Stock</ejb-name>
        <method-name>put</method-name>
      </method>
      ...
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>
```

Listing 7.5: Auszug aus dem Deployment-Deskriptor der Entity-Bean Stock

Die Instanzen der Stock-Bean werden von der Producer-Bean verwendet. Die Producer-Bean ist eine zustandsbehaftete Session-Bean. Sie entnimmt eine Einheit aus dem Lager *S1* und zwei Einheiten aus *S2* und erzeugt damit ein Produkt. Dieses legt sie im Lager *T1* ab.

Das Home-Interface der Producer-Bean stellt Listing 7.6 dar; es bietet lediglich die Standard *create*-Methode an.

```
package ejb.supplychain.producer;

import javax.ejb.CreateException;
import javax.ejb.EJBHome;
import java.rmi.RemoteException;

public interface ProducerHome extends EJBHome {
```

```
    public Producer create()
        throws CreateException, RemoteException;

}
```

Listing 7.6: ProducerHome – Home-Interface der Session-Bean Producer

Das Remote-Interface in Listing 7.7 definiert nur die Methode *produce*, die den oben beschriebenen Produktionsvorgang realisiert:

```
package ejb.supplychain.producer;

import java.rmi.RemoteException;
import javax.ejb.EJBObject;

public interface Producer extends EJBObject {

    public int produce(int amount)
        throws UnappropriateStockException, RemoteException;

}
```

Listing 7.7: Producer – Remote-Interface der Session-Bean Producer

Auch die Implementierung der Bean selbst besteht weitgehend aus Bekanntem. Listing 7.8 zeigt den Rahmen für die Klassendefinition.

```
package ejb.supplychain.producer;

import javax.ejb.CreateException;
import javax.ejb.EJBException;
import javax.ejb.FinderException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import javax.ejb.SessionSynchronization;
import javax.naming.NamingException;

import ejb.supplychain.stock.ProcessingErrorException;
import ejb.supplychain.stock.Stock;
import ejb.supplychain.stock.StockHome;
import ejb.util.Lookup;

public class ProducerBean implements SessionBean, SessionSynchronization {

    public static final String STOCK_HOME =
        "java:comp/env/ejb/Stock";
    public static final String SOURCE_ID1 =
        "java:comp/env/idSource1";
```

```
    public static final String SOURCE_ID2 =
        "java:comp/env/idSource2";
    public static final String TARGET_ID  =
        "java:comp/env/idTarget";

    private Stock source1;
    private Stock source2;
    private Stock target;

    private SessionContext sessionCtxt = null;

    ...
}
```

Listing 7.8: ProducerBean – Klassendefinition

Für die Initialisierung wird in der *ejbCreate*-Methode und in der *ejbActivate*-Methode eine interne Methode, *openRessources()*, aufgerufen (siehe Listing 7.9). Diese stellt sicher, dass alle benötigten Ressourcen verfügbar sind. Dazu sucht sie alle verwendeten Stock-Beans und merkt sich eine Referenz in den Variablen *source1* und *source2* für die Lager der Rohstoffe und in der Variable *target* für das Lager des Endprodukts. Die verwendete Stock-Bean wird später als Referenz im Deployment-Deskriptor vermerkt. Die IDs der verwendeten Stock-Beans sind Umgebungsvariablen der Bean und werden mittels JNDI gelesen. Auch diese werden später in den Deployment-Deskriptor aufgenommen. Der Zugriff auf den JNDI ist in der Hilfsklasse *ejb.util.Lookup* gekapselt. Der Quellcode dieser Hilfsklasse ist zusammen mit dem Quellcode dieses Kapitels an der auf dem Umschlag angegebenen Quelle zu finden.

```
...
    public void ejbCreate()
        throws CreateException
    {
        try {
            this.openRessources();
        } catch(Exception ex) {
            throw new CreateException(ex.getMessage());
        }
    }

    public void ejbActivate() {
        try {
            this.openRessources();
        } catch(Exception ex) {
            throw new EJBException(ex.getMessage());
        }
    }
```

```
    private void openRessources()
        throws FinderException, NamingException
    {
        try {
            // get stock IDs
            String idSource1 =
                (String)Lookup.narrow(SOURCE_ID1, String.class);
            String idSource2 =
                (String)Lookup.narrow(SOURCE_ID2, String.class);
            String idTarget  =
                (String)Lookup.narrow(TARGET_ID, String.class);
            // get home
            StockHome stockHome = (StockHome)
                Lookup.narrow(STOCK_HOME, StockHome.class);

            // get stocks
            this.source1 = stockHome.findByPrimaryKey(idSource1);
            this.source2 = stockHome.findByPrimaryKey(idSource2);
            this.target  = stockHome.findByPrimaryKey(idTarget);
        } catch(java.rmi.RemoteException e) {
            throw new EJBException(e.getClass().getName()
                                   + ": " + e.getMessage());
        }
    }
...
```

Listing 7.9: Initialisierung der ProducerBean

Die Producer-Bean speichert den *Session-Kontext*, der ihr in der Methode *setSessionContext* (vgl. Listing 7.10) übergeben wird, in der Variable *sessionCtxt*.

```
...
    public void setSessionContext(SessionContext ctxt) {
        this.sessionCtxt = ctxt;
    }
...
```

Listing 7.10: ProducerBean.setSessionContext – Setzen des Session-Kontexts

Mit der Methode *produce* in Listing 7.11 wird es jetzt interessant. Diese Methode geht davon aus, dass sie in einer Transaktion ausgeführt wird. Der Reihe nach verringert sie die Zahl der gelagerten Rohstoffe und erhöht die Zahl der gelagerten Endprodukte. Es gibt keine vorherige Prüfung, ob die Ausführung aller Aktionen möglich ist. Tritt bei einer Aktion ein Fehler auf, so wird einfach die gesamte Transaktion rückgängig gemacht. Dazu wird im *catch*-Block die Methode *setRollbackOnly* des Session-Kontexts aufgerufen. In diesem Fehlerfall wird also nicht nur eine Exception ausgelöst, sondern auch *setRollbackOnly* aufgerufen, da evtl. schon eine Stock-Bean ihren Wert geändert hat und somit ein ungültiger Zwischenzustand in der Datenbank besteht.

```
...
    public int produce(int amount)
        throws UnappropriateStockException
    {
        int ret;
        try {
            System.out.println("starting produce");
            this.source1.get(amount);
            this.source2.get(amount*2);
            this.target.put(amount);
            ret = amount;
        } catch(ProcessingErrorException e) {
            this.sessionCtxt.setRollbackOnly();
            throw new UnappropriateStockException();
        } catch (java.rmi.RemoteException re) {
            this.sessionCtxt.setRollbackOnly();
            throw new UnappropriateStockException();
        }
        return ret;
    }
...
```

Listing 7.11: ProducerBean.produce – Business-Logik der Session-Bean Producer

Die Definitionen für die Producer-Bean und die Stock-Bean stehen im gleichen Deployment-Deskriptor. Listing 7.12 zeigt die für die Producer-Bean relevanten Teile des Deployment-Deskriptors (Listing 7.5 hatte die für die Stock-Bean relevanten Teile gezeigt). Für die Methode *produce* der Producer-Bean wird ebenfalls *Required* als Transaktionsattribut im Deployment-Deskriptor deklariert. Auch die Umgebungsvariablen für die Producer-Bean werden definiert. Zusätzlich enthält der Deployment-Deskriptor eine Referenz auf die Stock-Bean.

```
...
  <enterprise-beans>
    ...
    <session>
      <ejb-name>Producer</ejb-name>
      <home>ejb.supplychain.producer.ProducerHome</home>
      <remote>ejb.supplychain.producer.Producer</remote>
      <ejb-class>ejb.supplychain.producer.ProducerBean</ejb-class>
      <session-type>Stateful</session-type>
      <transaction-type>Container</transaction-type>
      <env-entry>
        <env-entry-name>idSource1</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>stock1</env-entry-value>
      </env-entry>
      <env-entry>
```

```
        <env-entry-name>idSource2</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>stock2</env-entry-value>
      </env-entry>
      <env-entry>
        <env-entry-name>idTarget</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>stock3</env-entry-value>
      </env-entry>
      <ejb-ref>
        <ejb-ref-name>ejb/Stock</ejb-ref-name>
        <ejb-ref-type>Entity</ejb-ref-type>
        <home>ejb.supplychain.stock.StockHome</home>
        <remote>ejb.supplychain.stock.Stock</remote>
        <ejb-link>Stock</ejb-link>
      </ejb-ref>
    </session>
  </enterprise-beans>
  <assembly-descriptor>
    <container-transaction>
      ...
      <method>
        <ejb-name>Producer</ejb-name>
        <method-name>produce</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>
```

Listing 7.12: Auszug aus dem Deployment-Deskriptor der Session-Bean Producer

Wir haben für die Producer-Bean einen Test auf Basis eines Testframeworks (welches in Kapitel 9 vorgestellt wird) erstellt. Sie finden den kompletten Test neben vielen anderen Tests für die Beispiele dieses Buches unter den Sourcen zu diesem Buch, die sie von der auf dem Umschlag angegebenen Quelle beziehen können.

Listing 7.13 zeigt einen Ausschnitt aus dem Test für die Producer-Bean, aus dem genau zu ersehen ist, wie das erwartete Verhalten der Producer-Bean für oben beschriebene Methode ist.

```
...
Stock stock1 = this.stockHome.create("stock1", 500, 200);
Stock stock2 = this.stockHome.create("stock2", 500, 500);
Stock stock3 = this.stockHome.create("stock3", 100, 0);

int step = 1;

Producer p = this.producerHome.create();
```

```
p.produce(10);
this.assertEquals(step+":volume stock1", 190, stock1.getVolume());
this.assertEquals(step+":volume stock2", 480, stock2.getVolume());
this.assertEquals(step+":volume stock3", 10, stock3.getVolume());
step++;

p.produce(40);
this.assertEquals(step+":volume stock1", 150, stock1.getVolume());
this.assertEquals(step+":volume stock2", 400, stock2.getVolume());
this.assertEquals(step+":volume stock3", 50, stock3.getVolume());
step++;

p.produce(50);
this.assertEquals(step+":volume stock1", 100, stock1.getVolume());
this.assertEquals(step+":volume stock2", 300, stock2.getVolume());
this.assertEquals(step+":volume stock3", 100, stock3.getVolume());
step++;

try {
    p.produce(10);
    this.fail("Expected UnappropriateStockException");
} catch(UnappropriateStockException ex) {
    //as expected
}
this.assertEquals("final:volume stock1", 100, stock1.getVolume());
this.assertEquals("final:volume stock2", 300, stock2.getVolume());
this.assertEquals("final:volume stock3", 100, stock3.getVolume());
...
```

Listing 7.13: Auszüge aus dem Test für die Session-Bean Producer

Zunächst erzeugt der Test die drei Stock-Beans, *stock1*, *stock2* und *stock3*. *Stock1* und *stock2* sind die Läger, aus denen die Producer-Bean Material für die Fertigung des Endproduktes entnimmt. Das fertige Produkt wird dann im Lager *stock3* abgelegt (vgl. dazu nochmals Listing 7.12, Deployment-Deskriptor der Producer-Bean). Nach dem Erzeugen der Producer-Bean, werden 10 Einheiten produziert. Der Test stellt mit der Methode *assertEquals* sicher, dass *stock1* nun einen Bestand von 190 Einheiten hat (200 Einheiten waren der Anfangsbestand; eine Einheit wird zur Produktion von einer Einheit des Endproduktes entnommen), dass *stock2* nun einen Bestand von 480 Einheiten hat (500 Einheiten waren der Anfangsbestand, zwei Einheiten werden zur Produktion von einer Einheit des Endproduktes entnommen) und dass *stock3* einen Bestand von 10 Einheiten des Endproduktes hat (Anfangsbestand war 0 Einheiten). Anschließend werden 40 und dann noch einmal 50 Einheiten produziert und die entsprechenden Bestände geprüft. Im letzten Schritt des Tests wird ein Fehlerfall provoziert. Es werden 10 Einheiten produziert, die durch das Lager *stock3* nicht mehr aufgenommen werden können, da die maximale Kapazität mit 100 Einheiten angegeben wurde. Der Test stellt sicher, dass die Methode *produce* eine Ausnahme wegen des vollen Lagers *stock3* aus-

löst und dass nach dem fehlerhaften Aufruf der Methode *produce* immer noch ein konsistenter Zustand der Bestände herrscht. Würde die Methode *produce* nicht in einer Transaktion ausgeführt werden, so wären die Bestände des Lagers *stock1* und des Lagers *stock2* um die entsprechenden Einheiten erfolgreich reduziert worden, der Bestand des Lagers *stock3* aber nicht entsprechend erhöht, da das Maximum erreicht war. Der Datenbestand wäre in einem inkonsistenten Zustand.

7.3.3 Transaktionsattribute

Mit den Transaktionsattributen wird das transaktionale Verhalten für die Methoden von Enterprise-Beans definiert. Die Attribute werden in den Deployment-Deskriptor eingetragen. Der EJB-Container verwendet sie, um den Methoden einen passenden Transaktionskontext für ihre Ausführung zu geben. Darüber hinaus definieren die Transaktionsattribute das Zusammenspiel der Enterprise-Beans untereinander und mit dem Client in Hinblick auf die Transaktionen.

Im Folgenden sind alle Transaktionsattribute beschrieben, die von der EJB-Version 2.0 unterstützt werden.

NotSupported

Eine Methode, die das Transaktionsattribut *NotSupported* im Deployment-Deskriptor hat, unterstützt keine Transaktionen. Das bedeutet, dass die Methode nicht in einer Transaktion ausgeführt wird.

Bei der Entwicklung der Methode muss dies insofern berücksichtigt werden, als keine Möglichkeit besteht, alle ausgeführten Aktionen mit einem Rollback rückgängig zu machen. Für den Zugriff auf transaktionale Systeme, z. B. eine Datenbank, wird normalerweise nur mit einer lokalen Transaktion gearbeitet.

Eine Methode mit dem Transaktionsattribut *NotSupported* wird vom Client (zur Erinnerung: Client kann auch eine andere Enterprise-Bean sein) im Normalfall ohne globalen Transaktionskontext angesprochen. Sollte der Client jedoch eine Transaktion für den Aufruf verwenden, wird diese nicht in der Methode verwendet. Die Transaktion des Clients bleibt unbeeinflusst. Die Aktionen der Methode sind nicht an diese Transaktion gebunden. Hier sei nochmals darauf hingewiesen, dass das EJB-Konzept keine verschachtelten Transaktionen unterstützt.

Dieses Transaktionsattribut darf nicht von Beans verwendet werden, die das *SessionSynchronisation*-Interface implementieren (siehe auch Abschnitt 7.3.5, *Synchronisation*).

Die Konstante *NotSupported* kann im Deployment-Deskriptor ab der EJB-Version 1.1 verwendet werden. Für den serialisierten Deployment-Deskriptor der EJB-Version 1.0 heißt die entsprechende Java-Konstante *TX_NOT_SUPPORTED*.

Required

Methoden, die das Transaktionsattribut *Required* haben, werden immer in einer Transaktion ausgeführt.

Wenn die Methode vom Client mit einer globalen Transaktion aufgerufen wird, wird diese auch für die Ausführung der Methode verwendet. Falls der Aufruf ohne eine Transaktion geschieht, startet der EJB-Container automatisch eine Transaktion. Somit ist gewährleistet, dass die Methode immer in einer globalen Transaktion ausgeführt wird.

Der Zugriff auf andere Beans und transaktionale Systeme ist damit auch transaktionsgesichert, da der globale Transaktionskontext vererbt wird.

Bei der Verwendung des Transaktionsattributs *Required* sollte stets bedacht werden, dass globale Transaktionen teuer sind. Häufig besteht ein möglicher Weg, um die Performanz zu steigern, im teilweisen Verzicht auf globale Transaktionen.

Für die EJB-Version 1.0 heißt die entsprechende Konstante *TX_REQUIRED*.

Supports

Das Transaktionsattribut *Supports* bedeutet, dass eine Methode entweder mit oder ohne Transaktion ausgeführt werden kann.

Besteht bereits beim Aufruf eine Transaktion, so steht diese auch bei der Ausführung der Methode zur Verfügung. Wird die Methode ohne Transaktion aufgerufen, so steht auch bei der Ausführung keine Transaktion zur Verfügung.

Methoden, für die dieses Transaktionsattribut verwendet werden soll, müssen deshalb mit beiden Fällen umgehen können. Die Implementierung muss ihre Aufgabe mit und ohne Transaktionskontext erfüllen können.

Die EJB-Spezifikation sagt ausdrücklich, dass das Transaktionsattribut *Supports* anstelle von *Required* eingesetzt werden kann, um die Performanz zu verbessern. Wenn die genaue Implementierung der Methode nicht bekannt ist, ist beim Deployment beim Einsatz des Transaktionsattributs *Supports* Vorsicht geboten.

Dieses Transaktionsattribut darf nicht von Beans verwendet werden, die das *SessionSynchronisation*-Interface implementieren (siehe auch Abschnitt 7.3.5, *Synchronisation*) und es kann nicht von Message-Driven-Beans verwendet werden.

Für die EJB-Version 1.0 heißt die entsprechende Konstante *TX_SUPPORTS*.

RequiresNew

Eine Methode mit dem Transaktionsattribut *RequiresNew* hat ihre private Transaktion. Vor dem Start der Methode wird vom EJB-Container immer eine neue, globale Transaktion gestartet. Nach dem Verlassen der Methode versucht der EJB-Container immer, die Transaktion mit einem Commit zu beenden.

Eine Transaktion, mit der der Client (der wie bereits erwähnt auch eine andere Enterprise-Bean sein kann) die Methode aufruft, bleibt vollständig isoliert. Die Aktionen der Methode sind nicht an diese Transaktion gebunden. Die neue Transaktion wird beim Zugriff auf andere Beans oder bei der Verwendung eines transaktionalen Systems wie bekannt vererbt.

Dieses Transaktionsattribut kann nicht von Message-Driven-Beans verwendet werden.

Für die EJB-Version 1.0 heißt die entsprechende Konstante *TX_REQUIRES_NEW*.

Mandatory

Eine Methode mit dem Transaktionsattribut *Mandatory* muss immer vom Client mit einer globalen Transaktion aufgerufen werden. Existiert beim Aufruf keine Transaktion, führt dies zu einem Fehler (*TransactionRequiredException*).

Die Transaktion des Clients wird für die Ausführung der Methode verwendet und auch weitervererbt. Es ist sichergestellt, dass die Methode immer in einem globalen Transaktionskontext ausgeführt wird.

Dieses Transaktionsattribut kann nicht von Message-Driven-Beans verwendet werden.

Für die EJB-Version 1.0 heißt die entsprechende Konstante *TX_MANDATORY*.

Never

Eine Methode mit dem Transaktionsattribut *Never* darf vom Client niemals in einem Transaktionskontext aufgerufen werden. Dies würde zu einem Fehler führen (*RemoteException*).

Die Methode wird stets ohne eine Transaktion ausgeführt. Für die Entwicklung gelten die Einschränkungen wie beim Transaktionsattribut *NotSupported*.

Dieses Transaktionsattribut darf nicht von Beans verwendet werden, die das *SessionSynchronisation*-Interface implementieren (siehe auch Abschnitt 7.3.5, *Synchronisation*) und es kann nicht von Message-Driven-Beans verwendet werden.

Das Transaktionsattribut *Never* wird seit der EJB-Version 1.1 unterstützt.

7.3.4 Transaktionsisolation

Die Ebene der Sichtenkonsistenz (engl. *Isolation Level*) von Transaktionen wurde zu Beginn dieses Kapitels beschrieben. Wir wollen die 4 Konstistenzebenen an dieser Stelle nochmals aufgreifen und etwas vertiefen. Wir verwenden die allgemein gebräuchlichen englischen Begriffe für die verschiedenen Konsistenzebenen:

1. Read Uncommitted
   Diese Ebene ist die schwächste Ebene in Bezug auf Datenkonsistenz, die stärkste Ebene in Bezug auf Performanz. Wird diese Ebene der Sichtenkonsistenz zum Zugriff auf Daten (die sich z.B. in einer Datenbank befinden) in einer Transaktion verwendet, so kann das Problem der Inkonsistenz (Dirty Reads), das Problem des nichtwiederholbaren Lesens (Nonrepeatable Reads) und das Phantom Problem (Phantom Reads) auftreten.

2. Read Committed
   Diese Ebene ist die nächst stärkere Ebene nach Read Uncommitted in Bezug auf Datenkonsistenz. In einer Transaktion mit dieser Sichtenkonsistenz können Daten, die von anderen Transaktionen verändert, aber noch nicht committed wurden, nicht gelesen werden. Gelesen werden kann nur der Zustand, den die Daten seit dem letzten erfolgreich durchgeführten commit haben. Dadurch kann das Problem der Inkonsistenz (Dirty Reads) nicht mehr auftreten. Das Problem des nichtwiederholbaren Lesens (Nonrepeatable Reads) und das Phantom Problem (Phantom Reads) können nach wie vor auftreten.

3. Repeatable Reads
   Im Vergleich zur Sichtenkonsistenz Read Commited können Daten, die gerade von einer anderen Transaktion gelesen werden, in dieser Transaktion nicht verändert werden. Der schreibende Zugriff auf die Daten wird solange blockiert, bis die lesende Transaktion mit *commit* oder *rollback* abgeschlossen wird. Durch dieses Verhalten wird deutlich, dass mit zunehmender Datenkonsistenz bei parallelen Zugriffen die Performanz nachlässt, da Schreibzugriffe bereits in dieser Ebene blockiert werden können. In dieser Ebene der Sichtenkonsistenz kann nur noch das Phantom Problem (Phantom Reads) auftreten.

4. Serializable
   In dieser Ebene der Sichtenkonsistenz können weder das Problem der Inkonsistenz (Dirty Reads), das Problem des Nichtwiederholbaren Lesens (Nonrepeatable Reads) noch das Phantom Problem (Phantom Reads) auftreten. Die Transaktion erhält exklusiven Lese- und Schreibzugriff auf die Daten. Andere Transaktionen können die Daten weder lesen noch verändern. Lese- und Schreibzugriffe anderer Transaktionen werden blockiert, bis die Transaktion mit *commit* oder *rollback* abgeschlossen worden ist. Diese Ebene ist die stärkste Ebene in Bezug auf Datenkonsistenz, die schwächste Ebene in Bezug auf Performanz.

Der Entwickler einer Bean kann bestimmen, mit welcher Konsistenzebene Operationen in den Transaktionen ausgeführt werden. Dies wird in der Regel in den herstellerabhängigen Zusätzen zum Deployment-Deskriptor angegeben. Manche Hersteller bieten auch nur die Möglichkeit, die Sichtenkonsistenz auf Ebene des Datenbanktrei-

bers zu konfigurieren. Greift eine Bean direkt auf die Datenbank zu, so hat sie die Möglichkeit, die Ebene der Sichtenkonsistenz für die aktuelle Transaktion über die Methode *setTransactionIsolation* des Interfaces *java.sql.Connection* zu setzen.

Das Setzen der Konsistenzebene birgt ein großes Optimierungspotential. Es gilt, das richtige Verhältnis zwischen Datenkonsistenz auf der einen und Performanz auf der anderen Seite zu finden. Wie die Gewichtung aussieht, hängt von den Anforderungen der Anwendung ab.

7.3.5 Synchronisation

Für die Synchronisation von Beans mit den Container-Transaktionen kann die Bean das Interface *javax.ejb.SessionSynchronisation* implementieren. Session-Beans, die ihre Transaktionen selbst verwalten, benötigen diesen Mechanismus nicht, da sie selbst die vollständige Kontrolle über die Transaktionen haben. Auch Entity-Beans benötigen dieses Interface nicht. Es ist zustandsbehafteten (stateful) Session-Beans vorbehalten, die vom Container gesteuerte Transaktionen verwenden.

Das Interface besteht aus drei Methoden, die vom EJB-Container aufgerufen werden, wenn sich die Transaktion in bestimmten Zuständen befindet (siehe Listing 7.14). Die Implementierung des Interface ist optional und nur in wenigen Fällen nötig.

Um sicherzustellen, dass die Methoden vom EJB-Container korrekt aufgerufen werden, gibt es Einschränkungen bezüglich der Transaktionsattribute. Beans, die das *SessionSynchronisation*-Interface implementieren, dürfen nur die Transaktionsattribute *Required*, *RequiresNew* und *Mandatory* verwenden.

```
public void afterBegin()
  throws javax.ejb.EJBException,
         java.rmi.RemoteException;

public void beforeCompletion()
  throws javax.ejb.EJBException,
         java.rmi.RemoteException;

public void afterCompletion(boolean committed)
  throws javax.ejb,EJBException,
         java.rmi.RemoteException;
```

Listing 7.14: Definition des Inferface javax.ejb.SessionSynchronisation

Die Methode *afterBegin* wird aufgerufen, um der Bean mitzuteilen, dass die folgenden Methodenaufrufe in einer neuen Transaktion ausgeführt werden. Die Methode wird nicht unbedingt bei Beginn der Transaktion ausgeführt, sondern erst wenn dem EJB-

Container bekannt wird, dass eine bestimmte Bean an einer Transaktion teilnehmen wird. Die Methode wurde eingeführt, um beispielsweise das Caching von Datenbankinhalten für einen performanten Zugriff zu ermöglichen.

Die Methode *beforeCompletion* wird aufgerufen, bevor eine Transaktion abgeschlossen wird. Zu diesem Zeitpunkt ist noch nicht bekannt, ob ein Rollback oder ein Commit ausgeführt wird. Auch kann in dieser Methode noch ein Rollback (Methode *setRollbackOnly*) erzwungen werden. Die Methode wird verwendet, um zwischengespeicherte Datenbankinhalte zurückzuschreiben oder zusätzliche Konsistenzbedingungen zu prüfen.

Die Methode *afterCompletion* wird nach dem Ende einer Transaktion aufgerufen, um der Bean mitzuteilen, ob die Transaktion mit *rollback* oder *commit* beendet wurde. Achtung: Beim Aufruf der Methode ist die Transaktion bereits beendet worden. Die Methode wird außerhalb jeder Transaktion ausgeführt. Eingesetzt wird diese Methode, um Ressourcen freizugeben, die speziell für die Transaktion benötigt wurden.

7.4 Explizite Transaktionssteuerung

7.4.1 Transaktionssteuerung im Client

Die explizite Transaktionssteuerung des Clients bedeutet, dass der Client selbst die Kontrolle über die Transaktionen übernimmt. Vor dem Aufruf von Beans startet er eine globale Transaktion. Mit einem Rollback oder Commit beendet er diese Transaktion wieder.

Die Transaktion des Clients wird beim Aufruf einer Bean auf den Server übertragen und vom EJB-Container verwendet. Abhängig von den Transaktionsattributen der Bean kann der EJB-Container unterschiedlich reagieren. Eine Möglichkeit besteht darin, dass er die Transaktion des Clients für die Ausführung ihrer Methoden verwendet. Im anderen Fall wird eine eigene Transaktion der Bean verwendet, die entweder zuvor vom EJB-Container gestartet wird oder bereits besteht.

Eine explizite Transaktionssteuerung des Clients steht somit nicht im Widerspruch zu der zuvor besprochenen impliziten Transaktionssteuerung. Die Konzepte sind miteinander verflochten und ergänzen sich.

Abbildung 7.6 zeigt ein schematisches Beispiel für die explizite Transaktionssteuerung des Clients. Es wird gezeigt, wie die Beans den Transaktionskontext des Clients verwenden und weiter vererben. Mögliche Transaktionsattribute für die Beans, die die Verwendung einer vom Client gestarteten Transaktion unterstützen, sind *Required*, *Supports* oder *Mandatory*.

Der Client ist für die Transaktionssteuerung verantwortlich. Er greift direkt über das *Java Transaction API* (JTA) auf den Transaktionsmonitor zu. Um dies zu ermöglichen, muss JTA dem Client als Service auf dem Server angeboten werden und im Namensdienst eingetragen sein. Hier weisen die einzelnen Applikations-Server leider Differenzen auf, da der Name im Namensdienst differiert. Die entsprechenden Informationen müssen der Dokumentation des jeweiligen Produkts entnommen werden.

Nachdem sich der Client Zugriff auf das JTA verschafft hat, kann er eine globale Transaktion starten. Der Transaktionsmonitor wird mit dem *Java Transaction Service* (JTS) die Vererbung des Transaktionskontextes sicherstellen.

In der Transaktion kann der Client beliebig viele Methoden auch von unterschiedlichen Beans aufrufen. Nachdem alle Aktionen der Transaktion durchgeführt wurden, beendet der Client die Transaktion mit einem Commit oder Rollback.

In der Transaktion wird bei jedem Methodenaufruf einer Bean auf dem Server auch der Transaktionskontext vererbt. In unserem Beispiel verwendet die aufgerufene Bean die Transaktion des Clients für die Ausführung ihrer Methode.

Beim Aufruf von Methoden an weiteren Beans oder beim Zugriff auf transaktionale Systeme wird der Transaktionskontext stets vererbt. Somit werden alle Aktionen an die Transaktion des Clients gebunden und gemeinsam durchgeführt (Commit) oder gemeinsam rückgängig gemacht (Rollback).

Bei einem Fehler kann jede beteiligte Bean bestimmen, dass die Transaktion bei Abschluss rückgängig gemacht wird. Der Bean-Kontext bietet das entsprechende Interface. Der Entwickler einer Bean muss somit nicht auf JTA zugreifen.

Durch die explizite Transaktionssteuerung des Clients wird viel Programmlogik im Client-Programms realisiert. Dies muss kritisch betrachtet werden, da die Strukturierung der Applikation und die Wiederverwendbarkeit darunter leidet. Sie sollten immer überprüfen, ob die deklarativen Transaktionen des EJB-Konzepts nicht eine bessere Lösung ermöglichen.

Der Vorteil der expliziten Transaktionssteuerung des Clients ist, dass eine Applikationsentwicklung nur mit einer Programmierung am Client möglich wird. Die Menge der existierenden Beans am Server wird einfach verwendet, ohne sie zu erweitern oder zu verändern. Im Gegensatz dazu ist es bei der impliziten Transaktionssteuerung erforderlich, eine neue Bean zu entwickeln, wenn mehrere Methodenaufrufe in einer Transaktion ausgeführt werden sollen.

Die explizite Transaktionssteuerung des Clients ermöglicht somit eine schnellere Entwicklung von Prototypen. Dieser Weg ist auch interessant, wenn der weitaus größte Teil der Applikationslogik aus zugekauften Beans besteht und es keine eigene Entwicklung von Beans gibt.

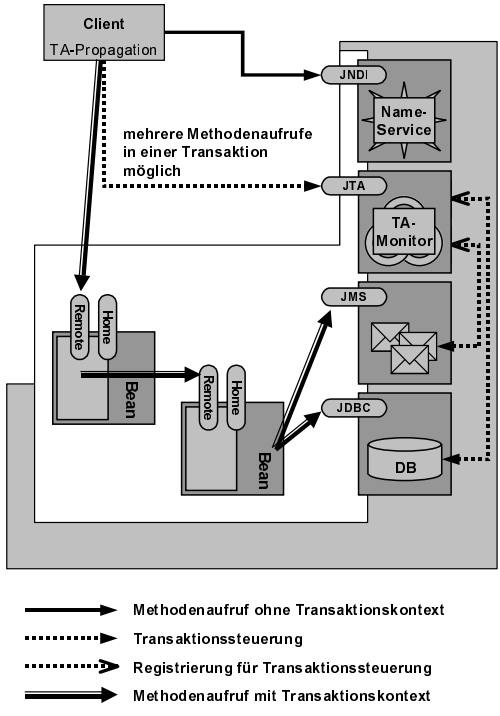

Abbildung 7.6: Beispiel für die Datenflüsse bei der expliziten Transaktionssteuerung des Clients

7.4.2 Beispiel *Producer*

Hier soll wieder das Beispiel *Producer* aufgegriffen werden, das schon bei der impliziten Transaktionssteuerung verwendet wurde.

Es soll die gleiche Logik realisiert werden, diesmal jedoch mit der Transaktionssteuerung im Client. Der Grund dafür könnte sein, dass nur die Entity-Bean *Stock* existiert und wir keine neue Bean schreiben wollen. Der Client wird die gleiche Funktionalität wie die Session-Bean *Producer* haben und zusätzlich die Transaktionssteuerung übernehmen.

Listing 7.15 zeigt, dass sich der Client im Konstruktor Zugriff auf die Transaktionen verschafft. Vom Name-Service bekommt er einen Stub der Klasse *UserTransaction*, ein leichtgewichtiges Objekt, das das JTA-Interface am Client anbietet. *UserTransaction* ist ein zentraler Bestandteil des JTS (Package *javax.jts*) und wird zusätzlich im JTA (Package *javax.transaction*) für den regulären Applikationsentwickler veröffentlicht.

```
package ejb.supplychain.client;

import ejb.supplychain.stock.Stock;
import ejb.supplychain.stock.StockHome;
```

```
import ejb.util.Lookup;

import javax.transaction.UserTransaction;


public class Client {

    private final static String STOCK_HOME =
        "Stock";
    private final static String USER_TA    =
        "javax.transaction.TransactionManager";

    private UserTransaction userTx = null;

    private StockHome stockHome = null;

    private Stock stock1;
    private Stock stock2;
    private Stock stock3;

    public Client() {
        try {
            this.userTx = (UserTransaction)
                Lookup.narrow(USER_TA, UserTransaction.class);
            this.stockHome = (StockHome)
                Lookup.narrow(STOCK_HOME, StockHome.class);
        } catch(javax.naming.NamingException ex) {
            ex.printStackTrace();
            throw new IllegalStateException(ex.getMessage());
        }
    }
...
```

Listing 7.15: ClientTAs –Zugriff auf JTA

Wie bei der impliziten Transaktionssteuerung, initialisiert der Client in Listing 7.16 drei Instanzen der Entity-Bean *Stock*. An dieser Stelle soll darauf hingewiesen werden, dass diese Aktionen ohne eine globale Transaktion ausgeführt werden. Durch das Transaktionsattribut *Mandatory* startet der EJB-Container automatisch eine Transaktion. Das zeigt, wie einfach sich die implizite und die explizite Transaktionssteuerung kombinieren lassen.

```
...
    public void preconditions()
        throws java.rmi.RemoteException
    {
        this.stock1 = this.createStock("stock1", 100, 100);
        this.stock2 = this.createStock("stock2", 100, 100);
        this.stock3 = this.createStock("stock3", 100, 0);
```

```
            System.out.println("Stock1 created. Current Volume: "
                               + this.stock1.getVolume());
            System.out.println("Stock2 created. Current Volume: "
                               + this.stock2.getVolume());
            System.out.println("Stock3 created. Current Volume: "
                               + this.stock3.getVolume());
        }

        private Stock createStock(String name, int max, int cap)
            throws java.rmi.RemoteException
        {
            Stock stock = null;
            try {
                stock = this.stockHome.findByPrimaryKey(name);
                stock.remove();
            } catch(javax.ejb.FinderException ex) {
                //do nothing
            } catch(javax.ejb.RemoveException ex) {
                ex.printStackTrace();
                throw new IllegalStateException(ex.getMessage());
            }
            try {
                stock = this.stockHome.create(name, max, cap);
            } catch(javax.ejb.CreateException ex) {
                ex.printStackTrace();
                throw new IllegalStateException(ex.getMessage());
            }
            return stock;
        }
    ...
```

Listing 7.16: ClientTAs – Initialisierung von drei Instanzen der Entity-Bean Stock

Die eigentliche Funktionalität (siehe Listing 7.17) wird vom Client in einer globalen Transaktion ausgeführt. Für die Transaktionssteuerung wird das Interface *UserTransaction* eingesetzt. Drei Methoden werden für die Transaktionssteuerung gebraucht: *begin*, *commit* und *rollback*.

Mit *begin* startet der Client eine neue Transaktion. Hier sei nochmals darauf hingewiesen, dass keine verschachtelten Transaktionen unterstützt werden. Jede Transaktion in einem Thread muss beendet werden, bevor die nächste gestartet werden darf.

Das bedeutet allerdings nicht, dass ein Client mit einer eigenen Transaktion keine Methode einer Bean aufrufen darf, die selbst eine eigene Transaktion verwendet. Das Client-Programm und die Bean-Methode werden in zwei unterschiedlichen Threads ausgeführt. Der EJB-Container vererbt (propagiert) die Transaktion nur, wenn zu der Methode entsprechende Transaktionsattribute definiert wurden. Somit werden auch in diesem Fall keine verschachtelten Transaktionen benötigt.

Mit *commit* signalisiert der Client, dass die Transaktion für ihn erfolgreich war und beendet werden soll. Die Transaktion wird nur mit einem Commit beendet, wenn keine an der Transaktion beteiligte Bean zuvor *setRollbackOnly* aufgerufen hat.

In der Transaktion kann es zu Fehlern kommen. Beispielsweise kann ein Lager das mögliche Fassungsvolumen überschreiten. Weil die Entity-Bean *Stock* im Fehlerfall nicht mit *setRollbackOnly* reagiert, sondern nur eine Exception auslöst, wäre es möglich, hier ein Verfahren für die Aufhebung dieser Fehlersituation zu realisieren. Das gegebene Beispiel reagiert jedoch einfach mit einem Rollback.

```
...
    public void doProduction(int volume)
        throws java.rmi.RemoteException
    {
        boolean rollback = true;
        try {
            this.userTx.begin();
            System.out.println("Producing " + volume
                               + " units ...");
            this.stock1.get(volume);
            this.stock2.get(volume*2);
            this.stock3.put(volume);
            System.out.println("done.");
            rollback = false;
        } catch(Exception ex) {
            System.out.println("FAILED.");
            System.err.println(ex.toString());
        } finally {
            if(!rollback) {
                try {
                    this.userTx.commit();
                } catch(Exception ex) {}
            } else {
                try {
                    this.userTx.rollback();
                } catch(Exception ex) {}
            }
        }
        System.out.println("Stock1 Volume: "
                           + this.stock1.getVolume());
        System.out.println("Stock2 Volume: "
                           + this.stock2.getVolume());
        System.out.println("Stock3 Volume: "
                           + this.stock3.getVolume());
    }

    public static void main(String[] args)
        throws java.rmi.RemoteException
    {
        Client c = new Client();
```

```
            c.preconditions();
            c.doProduction(10);
            c.doProduction(20);
            c.doProduction(50);
        }
    }
```

Listing 7.17: ClientTAs – der Programmabschnitt, der die Produktion durchführt

Jetzt ist die Funktionalität des Clients vollständig. Listing 7.18 zeigt die Ausgaben, die der Client zur Laufzeit produziert.

```
Stock1 created. Current Volume: 100
Stock2 created. Current Volume: 100
Stock3 created. Current Volume: 0
Producing 10 units ...
done.
Stock1 Volume: 90
Stock2 Volume: 80
Stock3 Volume: 10
Producing 20 units ...
done.
Stock1 Volume: 70
Stock2 Volume: 40
Stock3 Volume: 30
Producing 50 units ...
FAILED.
ejb.supplychain.stock.ProcessingErrorException: volume to small
Stock1 Volume: 70
Stock2 Volume: 40
Stock3 Volume: 30
```

Listing 7.18: ClientTAs – Client Output zur Laufzeit

Im Vergleich zu einer impliziten Transaktionssteuerung hat der Client mit expliziter Transaktionssteuerung deutlich mehr Funktionalität. Wenn die Wahl zwischen beiden Lösungswegen besteht, sollte bei vergleichbarer Funktionalität die implizite Transaktionssteuerung bevorzugt werden. Der Aufwand für die Implementierung der betrachteten Funktionalität ist besser für die Entwicklung einer wieder verwendbaren Serverkomponente investiert.

7.4.3 Transaktionssteuerung in der Bean

Session-Beans können auch selbst die vollständige Kontrolle über die Transaktionen übernehmen (seit der EJB-1.1.-Spezifiktation ist dies für Entity-Beans nicht mehr möglich). Der Transaktionskontext des Clients wird vom EJB-Container nicht an die Bean weitergegeben. Der Transaktionskontext der Bean bleibt von diesem vollständig isoliert.

Auch Message-Driven-Beans können Transaktionen selbst verwalten. Allerdings kann die Bean eine Transaktion nur innerhalb der *onMessage*-Methode beginnen. Die Transaktion muss noch vor Beendigung der *onMessage*-Methode abgeschlossen sein (entweder mit *commit* oder *rollback*). Eine Transaktion kann sich nicht über mehrere *onMessage*-Aufrufe erstrecken. Auch der Empfang einer Nachricht über die *onMessage*-Methode kann nicht Bestandteil einer von der Message-Driven-Bean gesteuerten Transaktion sein. Zu der Transaktion können nur Aktionen gehören, die die Message-Driven-Bean auf den Empfang einer Nachricht hin in der *onMessage*-Methode ausführt.

Eine Enterprise-Bean hat die Möglichkeit, mit globalen oder lokalen Transaktionen zu arbeiten. Bei der Verwendung von globalen Transaktionen darf es immer nur eine aktive Transaktion zu einem Zeitpunkt geben. Das bedeutet, dass vor dem Start einer globalen Transaktion alle lokalen Transaktionen abgeschlossen sein müssen und auch die vorhergehende globale Transaktion beendet sein muss.

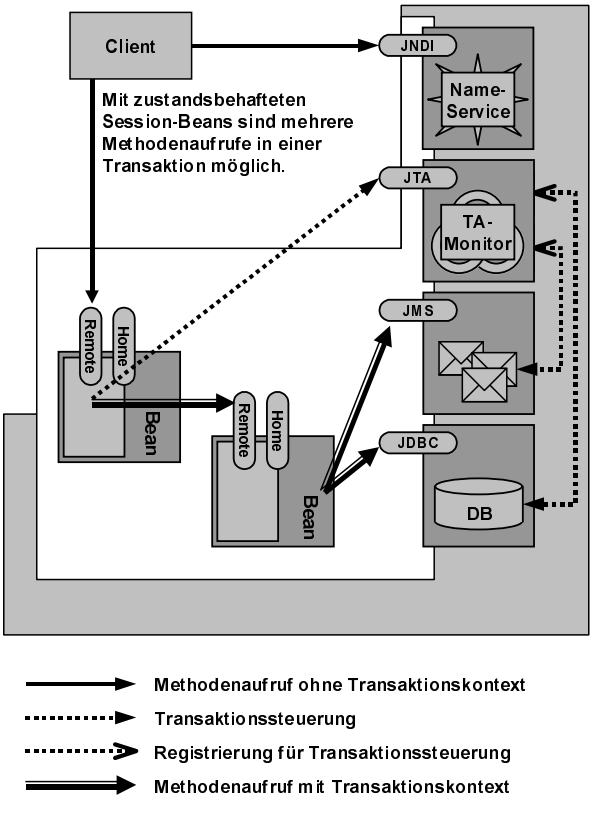

Abbildung 7.7: Beispiel für die Datenflüsse bei der expliziten Transaktionssteuerung der Bean mit einer globalen Transaktion

Abbildung 7.1 zeigt ein schematisches Beispiel für die explizite Transaktionssteuerung einer Bean, die mit einer globalen Transaktion arbeitet. Die Bean erbt nicht den Transaktionskontext des Clients, sondern startet ihre eigene globale Transaktion durch den direkten Zugriff auf den Transaktionsservice über JTS. Wie bekannt ist, wird die globale Transaktion vererbt, wenn die Bean andere Beans oder transaktionale Systeme verwendet.

Die Programmierung eines solchen Szenarios kann aus dem Beispiel für einen Client mit expliziter Transaktionssteuerung abgeleitet werden. Insbesondere der Zugriff auf JTA ist identisch.

Die Bean kann die gleiche Transaktion über mehrere Methodenaufrufe des Clients hinweg verwenden. Der EJB-Container verwaltet den Transaktionskontext für die Bean. Wenn die Bean eine globale Transaktion nicht vor dem Ende einer Methode abschließt, wird ein folgender Methodenaufruf automatisch mit dem gleichen Transaktionskontext assoziiert.

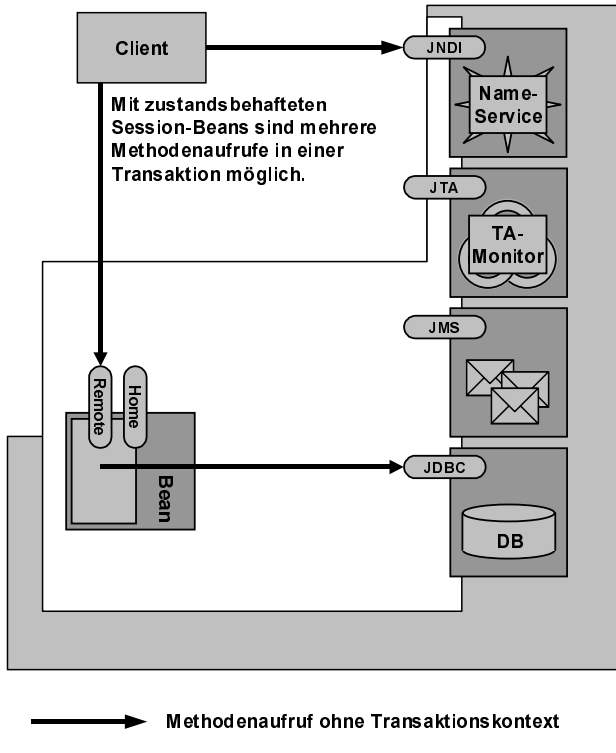

Abbildung 7.8: Beispiel für die Datenflüsse bei der expliziten Transaktionssteuerung der Bean mit einer lokalen Transaktion

Die Transaktionssteuerung in der Bean muss jedoch nicht mit globalen Transaktionen arbeiten. Durch die Beschränkung auf lokale Transaktionen kann der Entwickler der Bean freier auf die Eigenarten der verwendeten Datenbanken oder anderen transaktionalen Systeme eingehen. Abbildung 7.8 zeigt ein schematisches Beispiel für die explizite Transaktionssteuerung einer Bean, die mit lokalen Transaktionen arbeitet.

Systeme, die an einer globalen Transaktion beteiligt sind, müssen die JTS-Interfaces anbieten oder vom EJB-Container speziell angesprochen werden. Beispielsweise unterstützt erst JDBC 2.0 die globalen Transaktionen des JTS. Ältere Datenbanksysteme können nur dadurch verwendet werden, dass die EJB-Container die älteren Schnittstellen berücksichtigen. Der ausschließliche Einsatz von lokalen Transaktionen in einer Bean ermöglicht es, beliebige transaktionale Systeme zu verwenden. Dabei verwendet die Bean die spezifischen Schnittstellen des jeweiligen Systems für die Steuerung einer lokalen Transaktion.

Im folgenden Abschnitt wird das Beispiel *Migration* den Umgang mit lokalen Transaktionen in der Bean verdeutlichen.

7.4.4 Beispiel *Migration*

Das folgende Beispiel zeigt, wie eine Bean die lokalen Transaktionen einer Datenbank explizit steuert. Das Programm könnte ebenso gut als Beispiel für die Programmierung von JDBC dienen. Lokale Datenbanktransaktionen werden von Beans allein über JDBC gesteuert.

Das Beispiel *Migration* zeigt, wie die Währung in einer Datenbanktabelle von DM auf Euro umgestellt wird. Eine Session-Bean nimmt diese Umstellung in einer lokalen Transaktion vor.

Migration ist nur ein einfaches Beispiel. Die hier verwendeten lokalen Transaktionen würden der Bean jedoch weitergehende Freiheiten geben. Für vergleichbare Vorgänge könnten jedoch auch spezielle Langzeittransaktionen oder das Sperren ganzer Relationen nötig sein.

Zuerst soll in Listing 7.19 der Deployment-Deskriptor gezeigt werden. Beachten Sie, dass im Abschnitt *transaction-type* das Schlüsselwort *Bean* angegeben wird. Dies kennzeichnet, dass die Bean die explizite Transaktionssteuerung übernimmt. Gleichzeitig verbietet dieser Wert die Definition von Transaktionsattributen im Assembly-Deskriptor.

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0/
/EN" "http://java.sun.com/j2ee/dtds/ejb-jar_2_0.dtd">

<ejb-jar>
  <enterprise-beans>
```

```
    <session>
      <ejb-name>Migration</ejb-name>
      <home>ejb.migration.MigrationHome</home>
      <remote>ejb.migration.Migration</remote>
      <ejb-class>ejb.migration.MigrationBean</ejb-class>
      <session-type>Stateless</session-type >
      <transaction-type>Bean</transaction-type>
      <resource-ref>
        <description>
          Zu migrierende Datenbank
        </description>
        <res-ref-name>jdbc/Migration</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
      </resource-ref>
    </session>
  </enterprise-beans>
  <assembly-descriptor>
  </assembly-descriptor>
</ejb-jar>
```

Listing 7.19: Deployment-Deskriptor der Session-Bean Migration

Das Home- (vgl. Listing 7.20) und das Remote-Interface (vgl. Listing 7.21) zeigen die einfache Schnittstelle. Die Konvertierung wird durch den Aufruf der Methode *migrate* angestoßen. Die Ausnahme *MigrationErrorException* signalisiert einen Fehler bei der Ausführung.

```
package ejb.migration;

import javax.ejb.CreateException;
import javax.ejb.EJBHome;
import java.rmi.RemoteException;

public interface MigrationHome extends EJBHome {

    public Migration create()
        throws CreateException, RemoteException;

}
```

Listing 7.20: Home-Interface der Session-Bean Migration

```
package ejb.migration;

import java.rmi.RemoteException;
import javax.ejb.EJBObject;
```

```
public interface Migration extends EJBObject {

    public void migrate()
        throws MigrationErrorException, RemoteException;

}
```

Listing 7.21: Remote-Interface der Session-Bean Migration

Listing 7.22 zeigt die Implementierung der Bean-Klasse. Die Bean holt sich in der Methode *ejbCreate* eine Referenz auf eine Data-Source, über die Verbindungen zur Datenbank erzeugt werden können. In der Methode *ejbRemove* wird diese Referenz wieder gelöscht. Die ganze Funktionalität der Bean ist in der Methode *migrate* realisiert.

```
package ejb.migration;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.Statement;
import java.sql.SQLException;

import javax.ejb.CreateException;
import javax.ejb.EJBException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import javax.sql.DataSource;

import ejb.util.Lookup;

public class MigrationBean implements SessionBean {

    public static final float  EXCHANGE_RATE = 1.98f;
    public static final String RESOURCE_REF =
        "java:comp/env/jdbc/Migration";

    private SessionContext sessionCtx;
    private DataSource     dataSource;

    public MigrationBean() { }

    public void ejbCreate()
        throws CreateException
    {
        try {
            this.dataSource = (DataSource)
                Lookup.narrow(RESOURCE_REF, DataSource.class);
        } catch(Exception ex) {
            String msg = "Cannot get DataSource:"
                       + ex.getMessage();
```

```
            throw new EJBException(msg);
        }
    }

    public void ejbRemove() {
        this.dataSource = null;
    }

    public void ejbActivate() { }

    public void ejbPassivate() { }

    public void setSessionContext(SessionContext ctx) {
        this.sessionCtx = ctx;
    }

    private static final String QUERY1 =
        "UPDATE INVOICE SET AMOUNT=AMOUNT/? " +
        "WHERE CURRENCY='DEM'";

    private static final String QUERY2 =
        "UPDATE INVOICE SET CURRENCY='EU' " +
        "WHERE CURRENCY='DEM'";

    public void migrate()
        throws MigrationErrorException
    {
        Connection        con     = null;
        PreparedStatement st1     = null;
        Statement         st2     = null;
        boolean           success = false;
        try {
            con = this.dataSource.getConnection();
            con.setAutoCommit(false);
            st1 = con.prepareStatement(QUERY1);
            st1.setFloat(1, EXCHANGE_RATE);
            st1.executeUpdate();
            st2 = con.createStatement();
            st2.executeUpdate(QUERY2);
            success = true;
        } catch(SQLException ex) {
            String msg = "Failed migrating data:"
                       + ex.getMessage();
            throw new MigrationErrorException(msg);
        } finally {
            if(success) {
                try { con.commit(); } catch(Exception ex) {}
            } else {
                try { con.rollback(); } catch(Exception ex) {}
            }
            try { st1.close(); } catch(Exception ex) {}
```

```
                try { st2.close(); } catch(Exception ex) {}
                try { con.setAutoCommit(true); } catch(Exception ex) {}
                try { con.close(); } catch(Exception ex) {}
            }
        }

    }
```

Listing 7.22: Bean-Klasse der Session-Bean Migration

Die Konvertierung von DM nach EURO findet zu Zwecken der besseren Veranschaulichung in 2 getrennten Statements statt. Nachdem über die Data-Source eine Datenbankverbindung beschafft wurde, wird zunächst der *auto-commit*-Modus abgeschaltet. Im *auto-commit*-Modus wird nach dem Ausführen eines Statements auf der aktiven Transaktion automatisch ein *commit* durchgeführt. In unserem Fall könnten wir so nicht gewährleisten, dass sich die Daten am Ende der Methode in einem konsistenten Zustand befinden.

Durch das Abschalten des *auto-commit*-Modus sorgen wir dafür, dass beide Statements in der selben (lokalen) Datenbanktransaktion ausgeführt werden. Tritt in einer der beiden Statements ein Fehler auf, wird der Schalter *success* nicht auf *true* gesetzt. Das führt dazu, dass im *finally*-Block ein Rollback auf der Transaktion durchgeführt wird. Sind beide Statements erfolgreich, wird ein Commit durchgeführt. Bevor wir die Datenbankverbindung mit *Connection.close* wieder freigeben, setzen wir den *auto-commit*-Modus wieder auf seinen Standardwert zurück (Connection-Pool!). Die Bean hätte auch die Möglichkeit, über die Methode *Connection.setTransactionIsolation* einen bestimmten Isolationslevel für die Transaktion zu setzen.

Zusammenfassend kann gesagt werden, dass eine Session-Bean mit expliziter Transaktionssteuerung wie ein regulärer Datenbank-Client programmiert wird, wenn sie mit lokalen Transaktionen arbeitet. Wird mit globalen Transaktionen gearbeitet, ist die Programmierung mit einem EJB-Client mit expliziter Transaktionsteuerung vergleichbar (siehe Abschnitt 7.4.2, *Beispiel Producer*, mit expliziten Transaktionen des Clients).

7.5 Transaktionen im Deployment-Deskriptor

Die transaktionalen Eigenschaften einer Bean werden in drei Schritten definiert:

Erster Schritt

Im ersten Schritt definiert der Bean-Provider, ob die Transaktionen vom EJB-Container oder von der Bean gesteuert werden. Der Bean-Provider macht ausschließlich Angaben bei der Strukturinformation der Bean.

Bei expliziter Transaktionssteuerung der Bean definiert der Bean-Provider den Transaktionstyp *Bean* in der Strukturinformation der Bean im Abschnitt *enterprise-beans*:

```
<transaction-type>Bean</transaction-type>
```

Bei deklarativer Transaktionssteuerung wird bei der Strukturinformation der Bean keine Angabe zu den transaktionalen Eigenschaften gemacht. Diese werden erst in den folgenden Schritten mit den Transaktionsattributen festgelegt.

Zweiter Schritt

Im zweiten Schritt definiert der Application-Assembler die Transaktionsattribute. Der Application-Assembler macht ausschließlich Angaben über die Zusammenstellung der Bean im Abschnitt *assembly-descriptor*.

Zu jeder erlaubten Methode kann ein eigenes Transaktionsattribut angegeben werden. Dieses definiert das transaktionale Verhalten der Beans mit impliziter Transaktionssteuerung durch den EJB-Container.

Die EJB-Spezifikation macht genaue Angaben darüber zu welchen Methoden aus Home- und Remote-Interface ein Transaktionsattribut vergeben werden darf:

- bei Session-Beans nur für die definierten Business-Methoden,
- bei Entity-Beans für die definierten Business-Methoden und zusätzlich für die Methoden *find*, *create* und *remove* aus dem Home-Interface.

Die EJB-Spezifikation gibt drei unterschiedliche Schreibweisen an, um die Definition im Deployment-Deskriptor einfach und verständlich zu halten:

1. Für alle Methoden einer Bean wird das gleiche Transaktionsattribut vergeben.

```
<method>
  <ejb-name>Stock</ejb-name>
  <method-name>*</method-name>
</method>
```

2. Ein Transaktionsattribut wird für einen Methodennamen vergeben. Falls es mehrere Methoden mit demselben Namen, aber unterschiedlichen Parametern gibt, wird das Transaktionsattribut all diesen Methoden zugewiesen.

```
<method>
  <ejb-name>Stock</ejb-name>
  <method-name>get</method-name>
</method>
```

3. Ein Transaktionsattribut wird an genau eine Methode vergeben. Die Methode wird vollständig durch ihren Namen und die Parametertypen spezifiziert.

```
<method>
  <ejb-name>Stock</ejb-name>
  <method-name>get</method-name>
  <method-param>int</method-param>
</method>
```

Es ist möglich, dass im Home- und Remote-Interface Methoden mit einer identischen Signatur existieren. Mit dem zusätzlichen Attribut *method-intf* kann zwischen *Home* und *Remote* differenziert werden.

```
<method>
  <ejb-name>Stock</ejb-name>
  <method-intf>Home</method-intf>
  <method-name>create</method-name>
  <method-param>
    java.lang.String
  </method-param>
  <method-param>int</method-param>
  <method-param>int</method-param>
</method>
```

Die drei Schreibweisen können auch gleichzeitig verwendet werden. Dabei hat immer die präzisere Definition Vorrang vor der weniger genauen.

Dritter Schritt

Im dritten Schritt kann der Deployer die Angaben des Application-Assemblers verändern und ergänzen. Der Deployer muss sicherstellen, dass für jede Bean mit impliziter Transaktionssteuerung ein eindeutiges Transaktionsattribut für jede Methode definiert ist.

8 Sicherheit

8.1 Einleitung

Dieses Kapitel beschäftigt sich mit dem Schutz von Daten gegen nicht autorisierten Zugriff in einer Enterprise JavaBeans-Anwendung. Die Sicherheit von Systemen für den unternehmensweiten Einsatz ist ein sehr umfangreicher Themenkomplex. Bei der Konzeption entsprechender Systeme ist eine Strategie für die folgenden Themen erforderlich:

- Identifikation der Benutzer (Authentifizierung),
- Zugriffssicherheit (Autorisierung),
- sichere Datenübertragung (Verschlüsselung),
- Verwaltung von Benutzerdaten, Zertifikaten, etc.

Die entsprechenden Sicherheitskonzepte sind Bestandteil der Java 2 Enterprise Edition (J2EE) und einiger Erweiterungen. Die wichtigsten Erweiterung der Java-Plattform zum Thema Sicherheit sind die Java Cryptography Extension (JCE, vgl. [JCE]), der Java Authentication and Authorization Service (JAAS, vgl. [JAAS]) und die Java Secure Socket Extension (JSSE, vgl. [JSE]).

JCE ist grob gesagt eine Bibliothek für die Verschlüsselung von Daten, die sich über die Security-API (Package *java.security.**), die Bestandteil der Java-Laufzeitumgebung ist, nahtlos in die Java-Plattform integrieren lässt. JSSE ist eine Bibliothek für verschlüsselte Datenübertragung über Netzwerk-Sockets (Secure Socket Layer, kurz SSL), die sich genauso nahtlos in die Java-Plattform integrieren lässt wie die JCE. Dank der flexiblen Architektur der Java-Laufzeitumgebung (vgl. *java.net.SocketImplFactory*) lassen sich normale Netzwerk-Sockets durch sichere Netzwerk-Sockets der JSSE ersetzen, ohne dass die darüberliegenden Schichten davon betroffen sind. JAAS ist ein Dienst für die Authentifizierung und die Autorisierung von Benutzern, der einer Applikation über eine separate API zur Verfügung gestellt wird.

Ein Produkt, das diese Sicherheitsmechanismen unterstützt, ermöglicht die Konfiguration und Administration der entsprechenden Dienste und stellt dem Entwickler die entsprechenden Java-Interfaces zur Verfügung. Diese Sicherheitsmechanismen sind grundlegender Natur und betreffen Enterprise JavaBeans nur indirekt. Dieses Kapitel konzentriert sich auf die Sicherheitskonzepte der EJB-Architektur, die jedem EJB-Container von der EJB-Spezifikation vorgeschrieben werden. Weitergehende Informationen zu oben genannten Themen sind in [Jamie/Perone 2000] und [Scott 1999] zu finden.

Dieses Kapitel behandelt die folgenden Themen:

- *Definition von Rollen*:
  Die Enterprise-Beans arbeiten mit definierten Rollen, die auf Benutzer oder Benutzergruppen abgebildet werden. Benutzer und Benutzergruppen werden außerhalb des EJB-Containers definiert.
- *Zugriff auf die Identität eines Benutzers und seiner Rollen*:
  Im Kontext jeder Enterprise-Bean (Klasse *EJBContext*) stehen Methoden für den Zugriff auf den Benutzer zur Verfügung. Ferner ist die Prüfung seiner Rollenzugehörigkeit möglich.
- *Zugriffsschutz für Methoden*:
  Für die Methoden des Home- und Remote-Interfaces einer Enterprise-Bean kann definiert werden, welche Rollen die Benutzer innehaben müssen, um die Methoden aufrufen zu dürfen.
- *Ausführung von Enterprise-Beans mit einem definierten Benutzerkontext*:
  Es ist möglich eine Enterprise-Bean so zu konfigurieren, dass ihre Methoden nicht im Kontext des angemeldeten Benutzers, sondern in einem anderen Benutzerkontext ausgeführt werden.

Die folgenden Abschnitte werden sich diesen Themen ausführlich widmen. Abbildung 8.1 gibt einen zusammenfassenden Überblick.

Über die definierten Sicherheitskonzepte hinaus kann jeder EJB-Container weitere Dienste anbieten, wie beispielsweise eine Verwaltung für Zertifikate oder einen Verschlüsselungsservice. Der EJB-Container stellt Dienste dieser Art über JNDI in der Umgebung der Bean zur Verfügung. Wer jedoch solche Services verwendet, gefährdet die Portabilität seiner Enterprise-Beans, da ein anderer EJB-Container die Dienste eventuell nicht oder nur in veränderter Form anbietet.

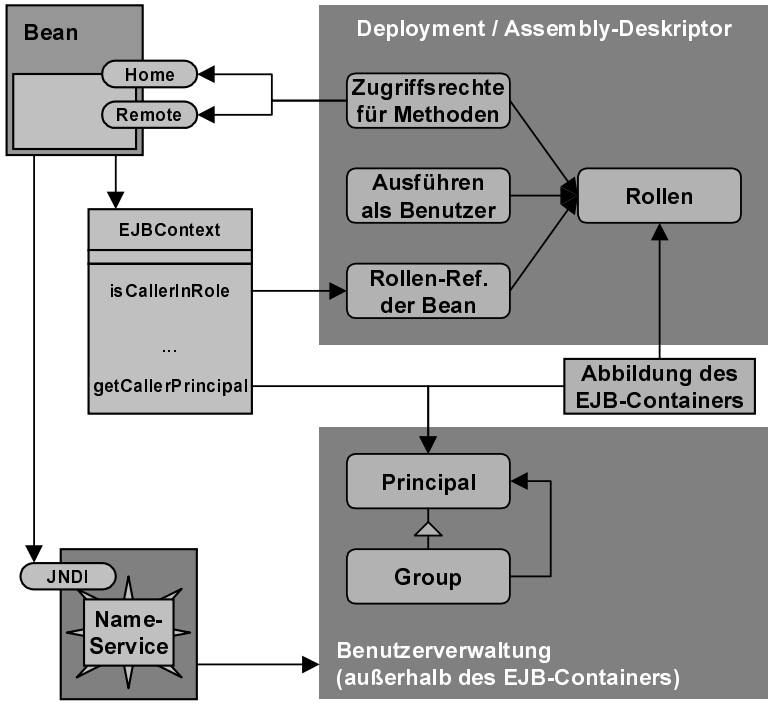

Abbildung 8.1: Schematische Darstellung der EJB-Sicherheit

8.2 Programmierung

8.2.1 Rechte für Methoden

Eine Methode Anwendungssysteme zu schützen ist, den Zugriff auf bestimmte Funktionsbereiche auf ausgewählte Benutzer zu begrenzen. Die EJB-Architektur unterstützt diese Vorgehensweise, indem es die Definition von Rollen (die auf existierende Benutzer abgebildet werden) ermöglicht. Der Zugriff auf die Methoden von Enterprise-Beans ist nur Benutzern mit definierten Rollen gestattet.

Um den Zugriff auf Methoden auf diese Weise zu beschränken, müssen die folgenden Schritte durchgeführt werden:

1. Einrichtung der benötigten Benutzer und Gruppen über die Benutzerverwaltung. Diese liegt außerhalb des EJB-Containers,
2. Definition der Rollen im Assembly-Deskriptor,
3. Definition der Zugriffsrechte für Methoden,
4. Zuordnung der definierten Rollen zu den real existierenden Benutzern und Gruppen.

Im Folgenden werden die erforderlichen Schritte erläutert, um die Methoden der Session-Bean *EuroExchangeSL* (siehe Beispiel aus Kapitel 4, Session-Beans) zu schützen. Beachten Sie dabei, dass für die Definition der Zugriffsrechte von Methoden keine Programmzeile geschrieben werden muss. Diese Konfiguration kann erst bei der Zusammenstellung der Anwendung durch den Application-Assembler geschehen.

Ein Applikations-Server hat entweder eine eigene Benutzerverwaltung oder er verwendet die Benutzerverwaltung eines anderen Systems. So kann ein Applikations-Server auch die Benutzer des Betriebsystems verwenden, um den doppelten Verwaltungsaufwand zu vermeiden. Entsprechend vielseitig sind auch die Benutzeroberflächen, die zur Benutzeradministration verwendet werden. Allen gemeinsam ist, dass sie die Definition von Benutzern und Passwörtern ermöglichen und die Benutzer in Gruppen strukturieren. Die beispielhafte Definition in Listing 8.1 legt die Benutzer *system*, *UserA* und *UserB* mit den jeweiligen Passwörtern und eine Benutzergruppe *Administrators* an. Sie werden im weiteren Verlauf dieses Kapitels für die Beispiele verwendet.

```
user.system=systempassword
user.UserA=password1
user.UserB=password2
group.Administrators=system, ...
```

Listing 8.1: Fiktive Benutzerdefinition eines Applikationsservers

Der Assembly-Deskriptor beschreibt das Zusammenspiel aller Enterprise-Beans in einem Java-Archiv (JAR). Hier werden auch die Benutzerrollen definiert, die unterschieden werden sollen. Die Rollen werden auch verwendet, um den Zugriff auf Methoden zu beschränken. Das Beispiel in Listing 8.2 definiert die Rollen *everyone* und *admins*.

```
<assembly-descriptor>
  ...
  <security-role>
    <description>
      Alle Benutzer der euroExchangeBean
    </description>
    <role-name>everyone</role-name>
  </security-role>
  <security-role>
    <description>
      Alle Administratoren
    </description>
    <role-name>admins</role-name>
  </security-role>
  ...
</assembly-descriptor>
```

Listing 8.2: Definition von Benutzerrollen im Deployment-Deskriptor

Der Zugriff auf die Enterprise-Bean wird jetzt so konfiguriert, dass nur Benutzer mit der Rolle *everyone* auf die Methoden der Bean zugreifen dürfen. Die Methoden zum Setzen der Wechselkurse sind besonders geschützt und dürfen nur von Benutzern aufgerufen werden, die die Rolle *admins* innehaben.

```
</assembly-descriptor>
  ...
  <method-permission>
    <role-name>everyone</role-name>
    <method>
      <ejb-name>EuroExchangeSL</ejb-name>
      <method-name>
        changeFromEuro
      </method-name>
    </method>
    <method>
      <ejb-name>EuroExchangeSL</ejb-name>
      <method-name>
        changeToEuro
      </method-name>
    </method>
  </method-permission>
  <method-permission>
    <role-name>admins</role-name>
    <method>
      <ejb-name>EuroExchangeSL</ejb-name>
      <method-name>
        setExchangeRate
      </method-name>
    </method>
  </method-permission>
  ...
</assembly-descriptor>
```

Listing 8.3: Definition der Zugriffsrechte für Methoden im Deployment-Deskriptor

Die Definition der Zugriffsrechte gestattet die gleiche Syntax wie die Definition der Transaktionsattribute (vergleiche Kapitel 7.5, *Transaktionen im Deployment-Deskriptor*). Für eine Enterprise-Bean können die Rechte für alle Methoden, alle Methoden mit einem bestimmten Namen oder eine Methode mit einem bestimmten Namen und bestimmten Parametern definiert werden. Auch können die Methoden aus Home- und Remote-Interface unterschieden werden.

Jetzt werden den Benutzern und Gruppen des Applikationsservers ihre Rollen zugewiesen. Dies geschieht mit den Deployment-Werkzeugen des EJB-Containers. Hier wird die Zuweisung beispielhaft anhand eines XML-Deskriptors gezeigt. Das Benutzerhandbuch ihres EJB-Containers enthält genaue Angaben, wie dieser Schritt durchzuführen ist.

```
<security-role-assignment>
  <role-name>everyone</role-name>
  <principal>UserA</principal>
  <principal>Administrators</principal>
</security-role-assignment>
<security-role-assignment>
  <role-name>admins</role-name>
  <principal>UserB</principal>
  <principal>Administrators</principal>
</security-role-assignment>
```

Listing 8.4: Beispielhafte Zuweisung von Benutzer und Gruppen zu Rollen

Nachdem Sie alle diese Definitionen gemacht haben und das Deployment der Enterprise-Bean erneut durchgeführt haben, können Sie das Client-Programm mit unterschiedlichen Benutzern testen. Ein unerlaubter Zugriff löst eine Exception aus (*java.rmi.RemoteException*).

Soll eine Methode ausdrücklich ohne Sicherheitsprüfung ausgeführt werden, so kann sie im Assembly-Deskriptor mit dem Attribut *unchecked* versehen werden. Listing 8.5 zeigt ein Beispiel, in dem die Methoden *changeFromEuro* und *changeToEuro* der EuroExchangeSL-Bean ohne Sicherheitsüberprüfung aufgerufen werden können. Jeder beliebige Benutzer kann die Methoden aufrufen. Lediglich die Methode *setExchangeRate* bleibt den Benutzern der Rolle *admin* vorbehalten.

```
</assembly-descriptor>
  ...
  <method-permission>
    <unchecked/>
    <method>
      <ejb-name>EuroExchangeSL</ejb-name>
      <method-name>
        changeFromEuro
      </method-name>
    </method>
    <method>
      <ejb-name>EuroExchangeSL</ejb-name>
      <method-name>
        changeToEuro
      </method-name>
    </method>
  </method-permission>
  <method-permission>
    <role-name>admins</role-name>
    <method>
      <ejb-name>EuroExchangeSL</ejb-name>
      <method-name>
        setExchangeRate
      </method-name>
```

```
      </method>
    </method-permission>
    ...
  </assembly-descriptor>
```

Listing 8.5: Verwendung des Attributs unchecked im Assembly-Deskriptor

8.2.2 Manuelle Zugriffsprüfung

Als zweite Möglichkeit, bestimmte Anwendungsfunktionalitäten gegen nicht autorisierten Zugriff zu schützen, können die Identität und die Rollen des Benutzers in der Anwendungslogik geprüft werden. Dadurch wird es zur Laufzeit möglich, zusätzliche Bedingungen in die Prüfung der Zugriffsrechte einzubeziehen.

Die folgenden Schritte sind erforderlich, wenn die Enterprise-Bean auf die Identität und die Rollen des Benutzers zugreifen möchte:

1. Einrichtung der benötigten Benutzer und Gruppen über die Benutzerverwaltung; diese liegt außerhalb des EJB-Containers,
2. Definition der Rollen-Referenzen,
3. Definition der Rollen im Assembly-Deskriptor,
4. Zuordnung der Rollenreferenzen zu den Rollen,
5. Verwendung der Methoden *getCallerPrincipal* und *isCallerInRole* in der Anwendungslogik der Enterprise-Bean.

Das folgende Beispiel verwendet die gleichen Benutzer und Gruppen wie das vorhergehende Beispiel. Die endgültige Definition der Rollen wird wie zuvor erst vom Application-Assembler vorgenommen. Der Bean-Entwickler kennt deshalb die Namen dieser Rollen noch nicht und muss in der Programmierung mit vorläufigen Namen arbeiten. Diese vorläufigen Namen werden Rollen-Referenzen (*security-role-ref*) genannt. Das folgende Beispiel definiert die beiden Rollen-Referenzen *exchange* und *setCurrency* für unsere Session-Bean.

```
<enterprise-beans>
  <session>
    ...
    <security-role-ref>
      <description>
        Rollen-Referenz zum Wechseln
      </description>
      <role-name>exchange</role-name>
    </security-role-ref>
    <security-role-ref>
      <description>
        Rolle-Referenz zum
        setzen der Wechselkurse
```

```
        </description>
        <role-name>setCurrency</role-name>
      </security-role-ref>
      ...
    <session>
  </enterprise-beans>
```

Listing 8.6: Definition von Rollen-Referenzen im Deployment-Deskriptor

Jetzt können in der Programmierung die Namen der definierten Rollen-Referenzen verwendet werden. Für den Zugriff auf den Benutzer stellt *EJBContext* zwei Methoden zur Verfügung:

- *java.security.Principal getCallerPrincipal()*
  Die Methode gibt die Identität des aktuellen Benutzers als Objekt der Klasse *Principal* zurück. Dieses Objekt kann auch für *Access-Control-Lists* verwendet werden.
- *boolean isCallerInRole(java.lang.String name)*
  Diese Methode bestimmt, ob der aktuelle Benutzer eine bestimmte Rolle innehat. Der hier angegebene Name entspricht dem Namen der Rollen-Referenz.

Als Beispiel für die Verwendung dieser Methoden dient die Methode *checkSecureAccess* (siehe Listing 8.7), die bei jedem Zugriff den Namen des Benutzers über den Fehlerausgabestrom (*stderr*) ausgibt. Zusätzlich wird geprüft, ob der Benutzer die Rollen-Referenz *setCurrency* innehat. In der Programmierung werden ausschließlich die Namen der Rollen-Referenzen und nicht die der Rollen verwendet.

```
...

public class EuroExchangeBean implements SessionBean {

...

    public void setExchangeRate(String currency,
                                float  euro,
                                float  foreignCurr)
    {
        if(currency == null) {
            throw new EJBException("illegal argument: currency");
        }
        if(!checkSecureAccess()) {
            throw new EJBException("Access denied");
        }

        ...

    }

    private boolean checkSecureAccess() {
        java.security.Principal principal;
```

```
        String name;
        boolean mayAccess;

        mayAccess = ctx.isCallerInRole("setCurrency");
        principal = ctx.getCallerPrincipal();
        name = principal.getName();

        if(mayAccess){
            System.err.println("Accessed granted to " + name);
        } else {
            System.err.println(name + ": Access denied!");
        }
        return mayAccess;
    }

    ...

}
```

Listing 8.7: Methode der Bean für den Zugriffsschutz

Erst nach der Fertigstellung der Enterprise-Bean werden die Rollen definiert und den Benutzern und Gruppen zugeordnet. Erst dann stehen die richtigen Namen für die im Programm verwendeten Rollen fest. Natürlich wird jetzt nicht die Programmierung geändert, sondern die Beziehung zwischen den Rollen-Referenzen und den Rollen definiert. Im Beispiel in Listing 8.8 wird der Rollen-Referenz *exchange* die Rolle *everyone* und der Rollen-Referenz *setCurrency* die Rolle *admins* zugewiesen. Dafür wird der Deployment-Deskriptor beim Deployment angepasst.

```
<enterprise-beans>
  <session>
    ...
    <security-role-ref>
      <description>
        Rolle zum Wechseln von Euro
      </description>
      <role-name>exchange</role-name>
      <role-link>everyone</role-link>
    </security-role-ref>
    <security-role-ref>
      <description>
        Rolle zum setzen der Wechselkurse
      </description>
      <role-name>setCurrency</role-name>
      <role-link>admins</role-link>
    </security-role-ref>
    ...
  <session>
</enterprise-beans>
```

Listing 8.8: Zuweisung der Rollen-Referenzen zu Rollen im Deployment-Deskriptor

Abbildung 8.2 zeigt noch einmal die Kette der Abbildungen. Die Enterprise-Bean verwendet die Namen der Rollen-Referenzen (Deployment-Deskriptor). Diese verweisen auf die beim Deployment definierten Rollen (Assembly-Deskriptor). Den Rollen werden die Benutzer (*Principal*) und Gruppen (*Group*) des Applikationsservers zugewiesen.

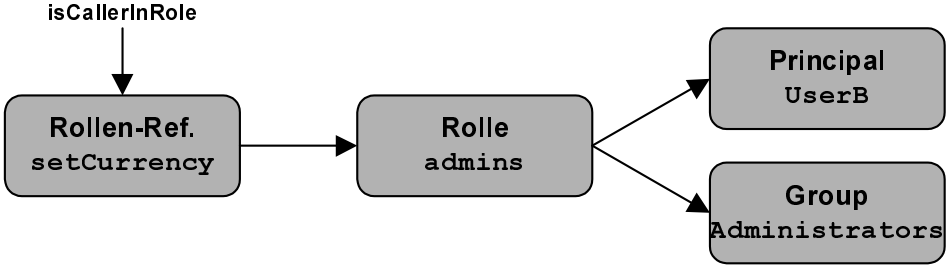

Abbildung 8.2: Kette der Abbildungen für Rollen

Diese Abbildungskette ist sehr förderlich für die klare Aufgabenteilung zwischen den Rollen Bean-Entwickler, Application-Assembler und Deployer (siehe 3.9, *EJB-Rollenverteilung*). Für den Entwickler kann dies jedoch eine zusätzliche Belastung bedeuten, wenn im Rahmen von Tests alle Rollen in seiner Person vereinigt werden.

8.2.3 Enterprise-Beans mit definiertem Benutzerkontext

Wenn eine Enterprise-Bean eine andere Enterprise-Bean verwendet, wird auch der Benutzerkontext weitergegeben. Die Methode *getCallerPrincipal* gibt überall den Benutzer zurück, der sich am Client angemeldet hat. Dies ermöglicht ein durchgängiges Sicherheitskonzept.

Es ist jedoch möglich, diese Kette gezielt zu durchbrechen und einen anderen Benutzer für die Ausführung einer Bean zu definieren (siehe Abbildung 8.3). Bei Message-Driven-Beans, die nicht direkt durch einen Benutzer, sondern durch eine asynchrone Nachricht aufgerufen werden, ist die Definition eines Benutzers immer erforderlich.

Der definierte Benutzer gilt immer für eine ganze Enterprise-Bean. Bei Session- und Entity-Beans wird der definierte Benutzerkontext für die Ausführung aller Methoden aus dem Home- und Remote-Interface verwendet. Bei Message-Driven-Beans wir die Methode *onMessage* mit dem gegebenen Benutzerkontext ausgeführt. Auch alle Enterprise-Beans, die von einer anderen Enterprise-Bean mit vorgegebenen Benutzerkontext aufgerufen werden, arbeiten dann mit diesem Benutzer. Dieses Vorgehen ist in vielen Anwendungsfällen sinnvoll, z. B. wenn der Benutzer in bestimmten Fällen mit erweiterten Rechten arbeitet.

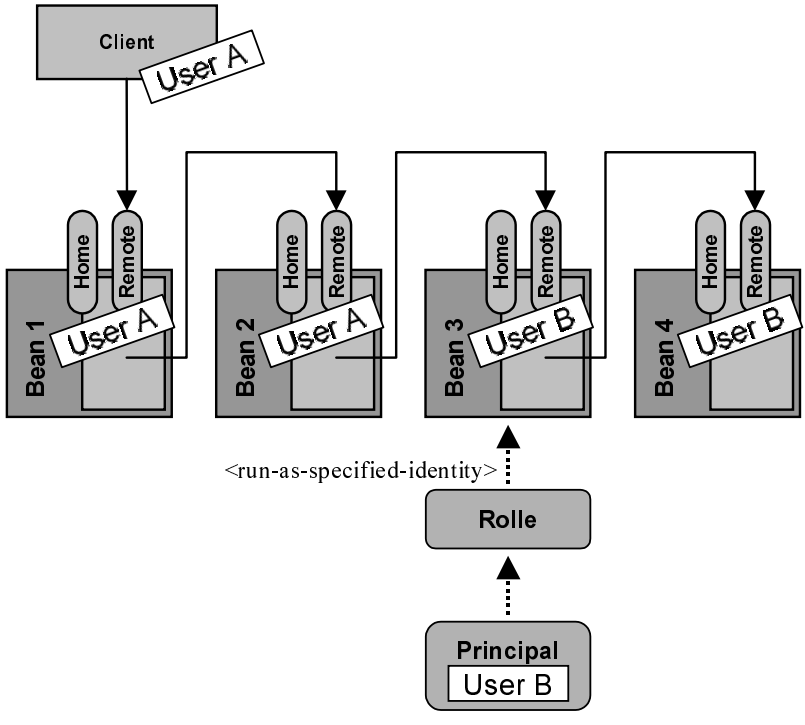

Abbildung 8.3: Benutzerkontext bei einer Kette von Aufrufen

```
<enterprise-beans>
  <session>
    <ejb-name>ejb/EuroExchangeSL</ejb-name>
    ...
    <security-identity>
      <run-as-specified-identity>
        <role-name>Special</role-name>
      </run-as-specified-identity>
    </security-identity>
    ...
  </session>
...
</enterprise-beans>
```

Listing 8.9: Definition eines Benutzerkontextes für eine Bean

Im angegebenen Auszug aus dem Deployment-Deskriptor der Session-Bean *EuroExchangeSL* in Listing 8.9 wird definiert, daß die Bean mit den Rechten der Rolle *Special* ausgeführt werden soll. Diese Angaben macht der Application-Assembler. Er muss auch die zugehörige Rolle definieren. Das folgende Beispiel zeigt die zugehörige Definition für den Benutzer *Special*.

```
<enterprise-beans>
  <session>
    ...
    <security-role-ref>
      <role-name>Special</role-name>
    </security-role-ref>
    ...
  <session>
<enterprise-beans>
```

Listing 8.10: Definieren der Benutzerrolle »Special«

Der Deployer wird später diese Rolle einem einzelnen Benutzer zuweisen. Dieser Benutzer wird dann für die Ausführung verwendet und auch von der Methode *getCallerPrincipal* als Ergebnis zurückgegeben.

8.2.4 Zusammenfassung

Es wäre ein Fehler, von dem vergleichsweise geringen Umfang dieses Kapitels auf die Bedeutung des Themas Sicherheit im EJB-Umfeld zu schließen. In der Entwicklung von Java wurde sehr früh das Thema Sicherheit berücksichtigt, wodurch die Sicherheitskonzepte ein fundamentaler Bestandteil der Sprache wurden. Beispielsweise gehört zu Java ein *Security-Manager*, der es ermöglicht, Programmteile mit unterschiedlichen Berechtigungen auszuführen und so den Zugriff auf sicherheitsrelevante Systemfunktionen zu begrenzen. Zu den Sprachkonzepten kommen die erwähnten Dienste und Klassen-Bibliotheken für *Authentifizierung*, *Autorisierung*, *Verschlüsselung* und für die Verwaltung der zugehörigen Objekte hinzu. Sprachkonzepte und Dienste zusammen bilden ein vollständiges Sicherheitskonzept.

Ein J2EE-konformer Applikations-Server stellt die Infrastruktur zur Verfügung, um die Sicherheitsdienste nutzen zu können. Er realisiert eine Benutzerverwaltung oder bindet eine bestehende ein. Der Applikations-Server kann so konfiguriert werden, dass nur verschlüsselte Netzwerkverbindungen zum Client akzeptiert werden. Es können unterschiedliche Authentifizierungsverfahren unterstützt werden, wie die Anmeldung mit Benutzernamen und Passwort oder mittels eines Zertifikats.

Der Applikationsserver ist auch dafür verantwortlich, die Identität des Benutzers durch alle Schichten der Anwendung weiterzureichen, beispielsweise von der Anmeldung auf einer Web-Seite über JSP zu den Enterprise-Beans und weiter bis zur Datenbank oder einem anderen System. Dadurch kann jede Schicht die Verantwortung für ihre Sicherheit übernehmen und in ihrer Funktionalität dem Benutzer mehr oder weniger Rechte einräumen.

Auch die Enterprise-Beans im EJB-Container übernehmen ihren Teil der Verantwortung für die Sicherheit des gesamten Systems. Durch Konfiguration können Zugriffsrechte für Methoden vergeben werden. Durch Programmierung kann die Identität des

Benutzers für Entscheidungen in der Geschäftslogik herangezogen werden. Außerdem kann das automatische Weiterreichen der Identität des Benutzers unterbrochen werden und ein Benutzer für die Ausführung einer Enterprise-Bean bestimmt werden. Die vielen anderen Belange der Sicherheit fallen nicht in den Zuständigkeitsbereich der Enterprise-Beans, sondern werden vom Applikations-Server oder den Persistenzsystemen übernommen.

Diese Vorgehensweise funktioniert so lange, wie das Thema Sicherheit nicht der eigentliche Gegenstand der Anwendung ist. Wenn also eine Verwaltung für Zertifikate geschrieben wird oder auch nur eine Access Control List definiert werden soll, sind die hier beschriebenen Verfahren nicht ausreichend. In diesem Fall wird eine Enterprise-Bean wieder eine zusätzliche Ressource benötigen, die spezialisierte Sicherheitsdienste anbietet. Zum Beispiel wird eine Enterprise-Bean für die Pflege von Benutzern anstelle einer Datenbank als Persistenzsystem, einen Dienst für die Benutzerverwaltung verwenden. Der Applikations-Server hat die Aufgabe, einen entsprechenden Dienst zur Verfügung stellen, der der Enterprise-Bean den Zugriff auf die Benutzer gestattet. Durch die Verwendung der bekannten Ressource-References kann dieser Dienst der Enterprise-Bean bekannt gegeben werden. Damit die Enterprise-Bean portabel bleibt, muss hier natürlich beachtet werden, dass der verwendete Dienst normiert ist und auch auf Applikations-Servern von anderen Herstellen verfügbar ist.

Zusammenfassend kann gesagt werden, dass ein J2EE-konformer Applikations-Server die Entwicklung von nach dem Stand der Technik sicheren Anwendungen unterstützt. Die Konzepte sind für unterschiedliche Sicherheitsstrategien geeignet und dafür ausgelegt, den Applikations-Server in eine bestehende Umgebung einzubetten. Da das bei Applikations-Servern anderer Plattformen nicht selbstverständlich ist, hat diese Tatsache EJB einen deutlichen Vorteil auf dem Markt verschafft.

9 Aspekte der praktischen Anwendung

In diesem Kapitel wollen wir Themen aufgreifen, die von der Spezifikation in der Version 2.0 gar nicht oder nur oberflächlich behandelt werden. Oft sind gerade diese Gesichtspunkte für die Entwicklung praxistauglicher Systeme hilfreich. Das soll nicht bedeuten, dass wir die Spezifikation für unvollständig oder lückenhaft halten. Ziel dieses Kapitels ist es, Anregungen für Implementierungen zu ausgewählten Themen in beispielhafter Form zu geben und diese zu diskutieren.

9.1 Performanz

Wie bereits in Kapitel 2 angesprochen, leidet die Programmiersprache Java nach wie vor unter dem Ruf einer schlechten Performanz. Gerade im Bereich Geschäftsanwendungen, zweifelsohne dem Schwerpunkt von Enterprise JavaBeans, ist dies ein kritischer Gesichtspunkt. Zu langsame Anwendungen verursachen für eine Firma Kosten, da die Mitarbeiter weniger effizient arbeiten können. Systembedingte Wartezeiten sind für die Arbeitsmotivation nicht förderlich. Der Arbeitsfluss des Mitarbeiters wird gehemmt und er wird in seiner Konzentration gestört. Deshalb ist die Akzeptanz langsamer Anwendungen bei den Benutzern geringer.

Dieser Abschnitt hat sich zur Aufgabe gesetzt, dem Leser die Eigenheiten der Architektur von Enterprise JavaBeans noch einmal vor Augen zu führen, um ein performanzbezogenes Bewusstsein beim Umgang mit EJB zu schaffen. Zu diesem Zweck wollen wir die Abläufe, die sich innerhalb des EJB-Containers abspielen, etwas genauer betrachten als in Kapitel 3.

Am abstrakten Beispiel einer einfachen Session- und einer einfachen Entity-Bean sollen die beteiligten Objekte sowie die Abläufe und Zusammenhänge unter ihnen durchleuchtet werden. Wir gehen davon aus, dass beide Beans den Remote-Client-View unterstützen. Die folgenden Ausführungen mögen nicht im Detail für jede Server- bzw. Containerimplementierung zutreffen, aber das Prinzip – und darauf kommt es an – ist bei allen Implementierungen gleich.

Beispiel: Währungsumrechnung

Die Entity-Bean (wir wollen von container-managed Persistence ausgehen) soll den Wechselkurs einer bestimmten Währung gegenüber dem Euro repräsentieren. Die Session-Bean soll einen Euro-Betrag anhand eines Wechselkurses in eine andere Währung umrechnen können. Die beiden Beans, die der Bean-Provider entwickelt, deklarieren jeweils ein Home- und ein Remote-Interface (*WechselkursHome*, *WechselkursRemote*, *UmrechnungHome* und *UmrechnungRemote*) und stellen jeweils eine Bean-Klasse mit der eigentlichen Implementierung zur Verfügung (*WechselkursBean* und *UmrechnungBean*). Beide Beans benutzen deklarative Transaktionen. Die Entity-Bean (*WechselkursBean*) hat zwei Attribute: die Währung als *String*-Variable und den Wechselkurs gegenüber dem Euro als *Float*-Variable. Die Währung dient dabei als Primärschlüssel (repräsentiert durch die Klasse *WechselkursPK*). Die Bean bietet eine Methode *wechselkursAlsFloat()* an, um auf den Wechselkurs als *Float*-Wert für Berechnungen zugreifen zu können. Die Session-Bean (*UmrechnungBean*) weist in ihrem Remote-Interface eine Methode *umrechnen()* auf. Diese Methode erwartet als Parameter den Euro-Betrag und den Wechselkurs in Form eines *Float*-Wertes und liefert als Ergebnis den Betrag in der entsprechenden Zielwährung.

Bei der Installation beider Beans in einem EJB-Container werden durch die Deployment-Tools des Container-Providers weitere Klassen erzeugt. Für die Home- und Remote-Interfaces werden jeweils aus den Angaben im Deployment-Deskriptor Implementierungsklassen generiert (die wir *WechselkursHomeImpl*, *WechselkursRemoteImpl*, *Umrechnung-HomeImpl* und *UmrechnungRemoteImpl* nennen und bereits als *EJBHome*- und *EJBObjekt*-Klasse in Kapitel 3 kennen gelernt haben). Da die Implementierungsklassen des Home- und des Remote-Interfaces zur Laufzeit als Remote-Objekte über RMI ansprechbar sein müssen, sind dafür entsprechende RMI-Klassen zu generieren (für jede Implementierungsklasse jeweils eine Stub- und eine Skeleton-Klasse). Außerdem wird für die abstrakte Bean-Klasse der Entity-Bean eine Implementierungsklasse generiert, die den Code der Bean um den Code des Persistence-Managers für den Zugriff auf die Datenbank erweitert. Um diese zwei primitiven Beans in einem Server zu installieren, sind demnach 16 Klassen und 4 Interfaces notwendig. Die 4 Interfaces, die beiden Bean-Klassen und die Primärschlüsselklasse werden durch den Bean-Provider erstellt, die restlichen Klassen werden durch die Tools des Container- bzw. Persistence-Manager-Providers generiert (wobei die tatsächliche Anzahl der generierten Klassen letzten Endes von der jeweiligen Implementierung des Containers bzw. des Persistence-Managers abhängt).

Der Clientcode für die Benutzung der beiden Beans könnte vereinfacht wie folgt aussehen (ein fixer Betrag von 100 Euro soll in US-Dollar umgerechnet werden):

```
…
InitialContext ctx = new InitialContext();
////////////
//Schritt 1 : Der Client erfragt den passenden Wechselkurs.
Object o = ctx.lookup("Wechselkurs");
WechselkursHome wh = (WechselkursHome)
```

```
        PortableRemoteObject.narrow(o, WechselkursHome.class);
//Suchen des Wechselkurses
WechselkursPK pk = new WechselkursPK("US-Dollar");
WechselkursRemote wr = (WechselkursRemote)
                          wh.findByPrimaryKey(pk);
//Auslesen des Wechselkurses
Float wechselkurs = wr.wechselkursAlsFloat();
////////////
//Schritt 2 : Umrechnen des Betrags in die gewünschte Währung
Object o1 = ctx.lookup("Umrechnung");
UmrechnungHome uh = (UmrechnungHome)
        PortableRemoteObject.narrow(o1, UmrechnungHome.class);
//Erzeugen der Session-Bean
UmrechnungRemote ur = (UmrechnungRemote)uh.create();
//Umrechnen von 100 Euro in die Zielwährung
float ergebnis = ur.umrechnen(100, wechselkurs);
...
```

Listing 9.1: Clientcode Währungsumrechnung

Im Folgenden wird betrachtet, welche Mechanismen durch diese wenigen Codezeilen des Clientprogramms zur Laufzeit auf dem Server (bzw. im Container) in Gang gesetzt werden und welche Objekte daran beteiligt sind.

Bereits beim Starten des Servers werden für die installierten Beans die Implementierungen der Home-Interfaces instanziiert und die dazugehörigen Stub-Objekte im Naming-Service zur Verfügung gestellt.

Schritt 1: Der Client erfragt den passenden Wechselkurs (vgl. Abbildung 9.1).

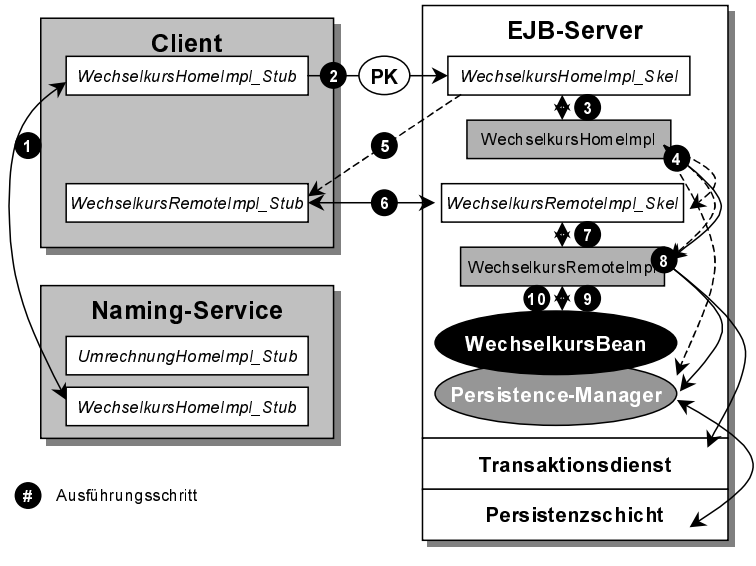

Abbildung 9.1: Beispiel: Erfragen des Wechselkurses

Ausführungs-schritt	Stichwortartige Erläuterungen zu Abbildung 9.1
1	JNDI-Lookup auf das Home-Interface der Wechselkurs-Bean (Netzwerkzugriff erforderlich)
2	Erzeugen eines *PrimaryKey*-Objektes und Aufruf der *findByPrimaryKey*-Methode (Netzwerkzugriff erforderlich)
3	Aufruf der *findByPrimaryKey*-Methode bei der zum Skeleton gehörenden Instanz der Home-Interface-Implementierung
4	Suchen der Daten über den Persistence-Manager, Erzeugen der Bean-Instanz und des Remote-Interface-Skeleton bzw. Rückgriff auf gepoolte Instanzen, Initialisierung der Instanzen
5	Remote-Interface-Stub wird als Ergebnis des *findByPrimaryKey*-Aufrufs an den Client geliefert
6	Client ruft die Methode *wechselkursAlsFloat()* auf, um den Wechselkurs zu erfragen (Netzwerkzugriff erforderlich)
7	Aufruf von *wechselkursAlsFloat()* beim Remote-Interface-Skeleton
8	Die Implementierung des Remote-Interface-Skeleton öffnet eine Transaktion. Der Persistence-Manager holt persistente Daten und setzt die Daten bei der Bean-Instanz.
9	Aufruf von *ejbLoad()* und *ejbActivate()* bei der Bean-Instanz
10	Aufruf der Methode *wechselkursAlsFloat()* bei der Bean-Instanz und Rückgabe des Ergebnisses. In der Implementierung des Remote-Interface-Skeleton wird nach Beendigung des Methodenaufrufs an der Bean-Instanz die Transaktion geschlossen und das Ergebnis an den Client zurückgegeben.

Über einen JNDI-Lookup holt sich der Client ein Stub-Objekt der Home-Interface-Implementierung in seinen Prozessraum. Um den gewünschten Wechselkurs zu finden, instanziiert und initialisiert der Client ein *PrimaryKey*-Objekt und übergibt es als Parameter beim Aufruf der *findByPrimaryKey*-Methode. Das *PrimaryKey*-Objekt wird dazu im Stub-Objekt serialisiert und mit dem RMI-Protokoll über das Netz an das zugehörige Skeleton-Objekt geschickt. Das Skeleton-Objekt deserialisiert die Daten, wertet sie aus und veranlasst einen entsprechenden Methodenaufruf beim Objekt der Home-Interface-Implementierung.

Die Home-Interface-Implementierung überprüft mit Hilfe des Persistence-Managers, ob ein Datensatz mit dem entsprechenden Primärschlüssel vorhanden ist. Dieser Vorgang erfordert einen Zugriff auf die Persistenzschicht. Danach wird überprüft, ob sich eine verfügbare Instanz der Bean-Klasse im Pool befindet. Falls nicht, wird eine neue Instanz erzeugt. Anschließend wird ein Skeleton-Objekt des Remote-Interface instanziiert. Die Bean-Instanz wird mit einem *EntityKontext*-Objekt versorgt. Als Ergebnis der Operation wird die *RemoteInterface*-Instanz an das Skeleton-Objekt der Wechselkurs-Home-Interface-Implementierung zurückgeliefert. Dort wird das zugehörige Stub-Objekt serialisiert und per RMI-Protokoll übers Netz als Rückgabewert an den Client geliefert.

Um den Wechselkurs in Form einer *Float*-Variable zu bekommen, ruft der Client eine entsprechende Methode des soeben erhaltenen Remote-Interface-Stubs der Wechselkurs-Bean auf (*wechselkursAlsFloat()*). Der Aufruf geht wiederum über Netz per RMI-Protokoll vom Stub an das zugehörigen Skeleton-Objekt auf dem Server, das seinerseits den Aufruf an die Remote-Interface-Implementierung weitergibt. Bevor der Aufruf von der Remote-Interface-Implementierung an eine Bean-Instanz weitergegeben werden kann, muss geklärt werden, ob eine Bean-Instanz mit der passenden Bean-Identität bereits aktiviert ist. Falls ja, verwendet der EJB-Container einfach diese Instanz. Steht keine Bean-Instanz mit der passenden Identität im Zustand Ready zur Verfügung wird eine Instanz aus dem Pool verwendet und mit der gefragten Bean-Identität initialisiert. Dann veranlasst der EJB-Container den Persistence-Manager, die persistenten Daten aus der Persistenzschicht zu holen und die Attribute der Bean-Instanz zu initialisieren. Abschließend ruft der EJB-Container die Methode *ejbActivate* der Bean-Instanz auf, um sie über den Wechsel in den Zustand Ready zu informieren. Steht im Pool keine freie Instanz mehr zur Verfügung, dann muss erst eine aktive Bean-Instanz passiviert werden oder der EJB-Container erzeugt eine neue Instanz. Erst wenn eine Bean-Instanz mit der richtigen Bean-Identität im Zustand Ready bereit steht, kann der EJB-Container den Methodenaufruf an diese weiterleiten.

Für die Betrachtung der Performanzaspekte wollen wir davon ausgehen, dass die Transaktionsattribute und der Transaktionskontext des Clients so geartet sind, dass der EJB-Container eine neue Transaktion starten und nach Abarbeitung der Methode wieder beenden muss. Normalerweise verwendet der EJB-Container immer globale Transaktionen. Das bedeutet, dass immer ein Transaktionsservice die Transaktionen koordiniert. Der EJB-Container greift über das Interface *UserTransaktion* auf den Transaktionsservice zu und steuert dadurch die Transaktion. Die Kommunikation des EJB-Containers mit dem Transaktionsservice kann unterschiedliche Ausprägungen haben. Eine typische Konstellation ist, dass der Transaktionsservice zum Lieferumfang des Applikationsservers gehört und auf dem gleichen Rechner in einem eigenen Prozess arbeitet. Bei einem großen System treten jedoch die Aspekte der Skalierung in den Vordergrund. Greifen mehrere EJB-Container auf die gleichen Daten zu, benötigen sie einen zentralen Transaktionsservice. In diesem Fall arbeitet der Transaktionsservice meist auf einer eigenen Servermaschine. Jeder Zugriff eines EJB-Containers auf den Transaktionsservice geht dann über eine Netzwerkstrecke.

Zuerst startet der EJB-Container eine globale Transaktion durch Aufruf der Methode *begin* im Interface *UserTransaction*. Jetzt besitzt der Thread einen Transaktionskontext. Jedem System, das an der Transaktion beteiligt ist, muss dieser Kontext mitgeteilt werden, damit es sich beim Transaktionsservice anmelden kann. Nur bei Aufrufen innerhalb des Threads wird der Kontext automatisch weitergegeben. Soll der Transaktionskontext auch bei Aufrufen über das Netz weitergegeben werden, muss ein spezieller Stub dies gewährleisten.

Wird innerhalb einer Transaktion beispielsweise auf die Datenbank zugegriffen, so überträgt der JDBC-Treiber (auch eine Art Stub) den Transaktionkontext an den Datenbankservice. Der Datenbankservice meldet sich mit der Transaktions-ID aus dem Kontext beim Transaktionsservice an. Dann wird eine lokale Transaktion in der Datenbank begonnen, die mit dem Transaktionskontext assoziiert ist. Wenn die globale Transaktion zu einem späteren Zeitpunkt beispielsweise mit *commit* abgeschlossen wird, wird der Datenbankservice vom Transaktionsservice darüber benachrichtigt. Der verwendete Zwei-Phasen-Commit benötigt dabei mindestens zwei Aufrufe (vgl. dazu auch Kapitel 7, *Transaktionen*).

Dieses Vorgehen ist für alle globalen Transaktionen identisch, unabhängig davon, ob die Bean oder der Persistence-Manager auf die Datenbank zugreift. Wenn wir davon ausgehen, dass Applikationsserver, Datenbank und Transaktionsservice auf eigenen Servermaschinen laufen, entstehen durch eine Transaktion mindestens vier zusätzliche Aufrufe (*begin*, Registrierung, zweimal *commit*) über eine Netzwerkstrecke.

In unserem Beispiel wird nach dem *begin* der eigentliche Methodenaufruf an die Bean-Instanz weitergegeben. Nach der Ausführung der Methode beendet der EJB-Container die Transaktion wieder mit *commit* oder *rollback*, abhängig vom Auftreten eines Fehlers in der Methode. Dann wird das Ergebnis der Methode in Form des Rückgabewerts zurück an den Skeleton gereicht. Dort wird – wie bei *findByPrimaryKey* – das Ergebnis für die Netzwerkübertragung serialisiert und zurück an den Client-Stub gesendet. Dieser deserialisiert die Daten und stellt sie dem aufrufenden Objekt in Form des Rückgabewerts zur Verfügung.

Schritt 2 : Umrechnen des Betrags in die gewünschte Währung (vgl. Abbildung 9.2)

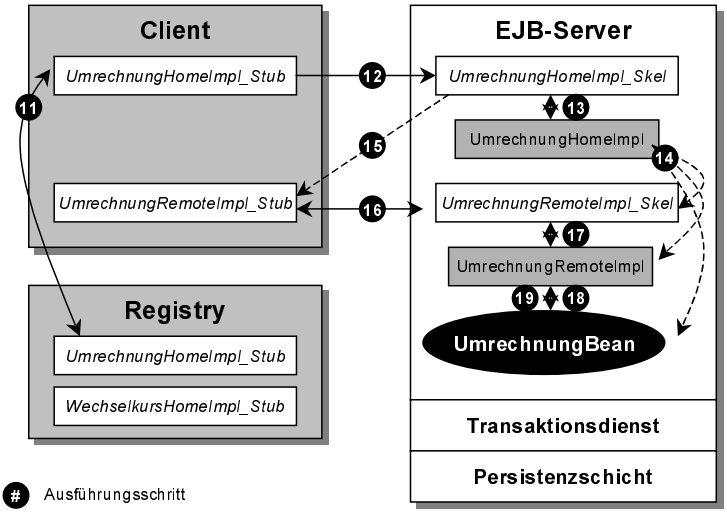

Abbildung 9.2: Beispiel : Umrechnen in die Zielwährung

Ausführungs-schritt	Stichwortartige Erläuterungen zu Abbildung 9.2
11	JNDI-Lookup auf das Home-Interface der Umrechnung-Bean
12	Aufruf der *create*-Methode (RMI)
13	Aufruf der *create*-Methode beim Home-Interface-Skeleton der UmrechnungBean
14	Erzeugen der Bean-Instanz und Remote-Interface-Skeleton bzw. Rückgriff auf Instanzen im Pool, Initialisierung der Instanzen
15	Remote-Interface-Stub wird als Ergebnis des *create*-Aufrufes an den Client geliefert
16	Client ruft die Methode *umrechnen()* auf, um den Betrag in die Zielwährung umzurechnen (RMI)
17	Aufruf von *umrechnen()* beim Remote-Interface-Skeleton
18	Evtl. Öffnen einer Transsaktion, Rückgriff auf Bean-Instanz im Pool
19	Aufruf der Methode *umrechnen()* bei der Bean-Instanz und Rückgabe des Ergebnisses an den Client; evtl. Schließen der Transaktion

Bei diesem Schritt wiederholt sich im Wesentlichen das, was bereits im ersten Schritt detailliert dargestellt wurde. Anstatt eines *findByPrimaryKey*-Aufrufs findet nun ein *create*-Aufruf statt (da es sich um eine Session-Bean handelt). Außerdem fehlen die persistenzbezogenen Operationen. Unter Umständen kann je nach Serverimplementierung die Behandlung von Entity- und Session-Beans bezüglich des Poolings unterschiedlich ausfallen.

Die sehr detaillierte Darstellung der Abläufe in den Container-Klassen in unserem Beispiel dürfte zu erkennen gegeben haben, dass Operationen auf Enterprise-Beans relativ aufwändig sind. Bei einem Komponentenmodell wie Enterprise JavaBeans ist ein gewisser Overhead nicht zu vermeiden.

Die Kommunikation über RMI ist für den Entwickler zwar einfach und bequem, allerdings geht diese Bequemlichkeit zu Lasten der Ausführungsgeschwindigkeit. Der für jeden Bean-Typ entstehende Container-Code kann nicht – oder nur in sehr geringem Umfang – unter den verschiedenen Bean-Typen geteilt werden, da er aus den spezifischen Angaben des Deployment-Deskriptors einer Enterprise JavaBean generiert wird. Zur Laufzeit ist pro Bean-Typ ein Skeleton-Objekt des Home-Interface erforderlich (gilt für Entity- und Session-Beans). Bei Entity-Beans ist für jede Bean im Zustand Ready ein Skeleton-Objekt des Remote-Interface nötig. Bei Session-Beans ist eines je Bean-Typ und verbundenem Client erforderlich.

Der Umfang des Container-Overheads (sprich Umfang der zur Laufzeit vorhandenen Container-Objekte) ist direkt proportional zur Anzahl der installierten und genutzten Beans (bei Session-Beans spielt dabei auch die Anzahl der zum Server verbundenen Clients eine Rolle). Der Umfang der Bean selbst hat relativ wenig mit dem generierten

Container-Code zu tun. Eine Bean mit einem kleinen Home- bzw. Remote-Interface hat nicht wesentlich weniger Container-Code und Laufzeitobjekte als eine Bean mit einem umfangreichen Home- bzw. Remote-Interface.

Konsequenzen

Ein wichtiger Aspekt beim Umgang mit Enterprise JavaBeans im Hinblick auf eine gute Performanz ist das Design der Netzwerkschnittstelle. Damit sind die Home-, vor allem aber die Remote-Interfaces der Beans gemeint. Ziel ist es, möglichst wenige Methoden anzubieten, die möglichst viel Funktionalität enthalten und den Austausch möglichst großer Datenmengen erlauben. Wie obige Ausführungen zeigen, sind Aufrufe von Bean-Methoden aufwändige Operationen. Daher lohnt es sich im Hinblick auf die Optimierung der Verarbeitungsgeschwindigkeit an dieser Stelle bereits zum Designzeitpunkt anzusetzen. Feingranulare Operationen (wie z.B. das Abfragen des Wechselkurses in unserem Beispiel) sind durch ein gewisses Missverhältnis zwischen dem Umfang der Funktionalität und dem Container-Overhead gekennzeichnet. Sie wirken sich negativ auf die Performanz des Anwendungssytems aus.

Eine nicht minder große Rolle bei der Performanz von EJB-Anwendungen spielt das Design des Datenmodells. Entity-Beans sind schwergewichtige Konglomerate aus Laufzeitobjekten. Feingranulare Entitäten, d.h. Datensätze mit wenigen Datenfeldern, die in großer Anzahl vorkommen, sind nach Möglichkeit zu vermeiden. Ein Negativbeispiel wäre die Entität Wechselkurs, wie sie in diesem Abschnitt verwendet wird. Wie bei der Netzwerkschnittstelle lohnt es sich, grobgranulare Entitäten zu definieren. Ziel dabei ist es, die Anzahl der Entity-Bean- und Container-Klassen-Instanzen möglichst gering zu halten. Die Definition grobgranularer Entitäten und die Modellierung von Methoden, die viel Funktionalität anbieten und den Transfer großer Datenmengen erlauben, verringert zudem die Anzahl der notwendigen Netzwerkzugriffe.

Eben dargestellte Ausführungen relativieren sich, wenn eine Enterprise-Bean über den Local-Client-View angesprochen wird. Da sich der Client und die Enterprise-Bean wissentlich im selben Prozess befinden, entfällt der Overhead der Netzwerkkommunikation. Durch den Wegfall der Serialisierung von Übergabeparametern und Rückgabewerten und durch den Wegfall der Latenz des Netzwerks hat der Local-Client-View gegenüber dem Remote-Client-View einen Vorteil bezüglich der Performanz. Ob die Schnittstelle der Enterprise-Bean fein- oder grobgranular ist, spielt in diesem Fall eine eher untergeordnete Rolle. Was auch beim Local-Client-View bleibt, sind der Container-Overhead, der Overhead globaler Transaktionen und im Fall von Entity-Beans der Transport der Daten von und zur Datenbank.

Der Local-Client-View gibt dem Design von EJB-Anwendungen eine neue Dimension. Durch die optimale Kombination von Enterprise-Beans mit Remote-Client-View (grobgranulare Schnittstelle für Remote-Clients) und Local-Client-View (Lokale Schnitt-

stelle, die in erster Linie von anderen Enterprise-Beans angesprochen wird) läßt sich bereits zum Designzeitpunkt maßgeblich Einfluss auf die Performanz des späteren Systems nehmen.

Das Performanzverhalten unterscheidet sich generell nach Art der Bean (unabhängig davon, ob der Remote- oder Local-Client-View zum Einsatz kommt). Bei Session-Beans sind aus performanzorientierter Sicht zustandslose gegenüber zustandsbehafteten Session-Beans zu bevorzugen. Der EJB-Container kann durch Pooling zustandslose Session-Beans sehr viel effizienter verwalten als zustandsbehaftete Session-Beans. Bei letzteren wirkt sich die Objektserialisierung durch den Aktivierungs- bzw. Passivierungsmechanismus zusätzlich negativ auf die Ausführungsgeschwindigkeit aus. Bei Entity-Beans spielt die gewählte Form des Persistenzmechanismus eine Rolle. Bei der Verwendung von container-managed Persistence ist man – was die Performanz der Datenspeicherungsroutinen angeht – von der Implementierung des Persistence-Managers abhängig. Der Persistence-Manager-Provider muss generische Routinen zur Verfügung stellen, die gegenüber speziell auf einem bestimmten Datenmodell beruhenden Algorithmen immer im Nachteil sind (für den Fall der bean-managed Persistence). So kann es in bestimmten Fällen notwendig sein, auf die Bequemlichkeit von container-managed Persistence zugunsten der Ausführungsgeschwindigkeit zu verzichten.

Message-Driven-Beans nehmen insofern eine gewisse Sonderstellung ein, als sie keine öffentliche Schnittstelle haben. Sie können von einem Client nicht direkt angesprochen werden. Außerdem sind Message-Driven-Beans nicht persistent. Der Overhead des EJB-Containers ist deswegen bei diesem Bean-Typ wesentlich geringer als bei Entity- und Session-Beans. Message-Driven-Beans können vom EJB-Container ähnlich effizient wie zustandslose Session-Beans verwaltet werden.

Wegen der Möglichkeit zur parallelen Verarbeitung erhöhen Message-Driven-Beans tendenziell die Performanz eines Systems. Damit parallele Verarbeitung möglich ist, müssen mehrere Threads zur Verfügung stehen. Der EJB-Container übernimmt die Verwaltung der Threads und die Zuordnung zu Instanzen von Message-Driven-Beans. Je mehr unterschiedliche Typen von Message-Driven-Beans eingesetzt werden, desto mehr Threads benötigt der EJB-Container, um die parallele Verarbeitung für die verschiedenen Typen von Message-Driven-Beans zu gewährleisten.

Threads sind eine kostbare Ressource des Betriebssystems, die viel Verwaltungsoverhead mit sich bringen. Die Verwendung von Message-Driven-Beans zur Verbesserung der Performanz sollte aus diesem Grund nicht überstrapaziert werden, da der Effekt der Performanzsteigerung durch den erhöhten Aufwand des Betriebssystems mit der Verwaltung der Threads beeinträchtigt werden kann. Außerdem wird nicht jeder Typ einer Message-Driven-Bean über einen eigenen Pool von Threads verfügen können. Innerhalb eines Applikationsservers (der EJB-Container ist neben anderen Containern

Bestandteil eines solchen Servers) steht in der Regel ein zentraler Thread-Pool zur Verfügung, der unter den verschiedenen Diensten bzw. Containern aufgeteilt wird. Je mehr unterschiedliche Typen von Message-Driven-Beans verwendet werden, desto leichter kann die Ressource *Thread* knapp und somit zu einem Engpass werden.

Die Verarbeitung eines Prozesses durch die Verwendung mehrerer Threads zu parallelisieren heißt nicht, dass dadurch automatisch eine bessere Gesamt-Performanz erzielt werden kann. Ob die Verarbeitung schneller oder langsamer ist, hängt von vielen Faktoren ab. Einer dieser Faktoren ist z.B. die Hardware, auf der das System zum Einsatz kommt. Ein Rechner mit einer CPU kann immer nur einen Thread arbeiten lassen, während die anderen Threads warten müssen, bis sie CPU-Zeit zugeteilt bekommen. Einen CPU-intensiven Prozess über mehrere Threads zu parallelisieren hat auf einem Rechner mit einer CPU zur Folge, dass die Gesamtperformanz schlechter wird. Zusätzlich zu der Zeit die die CPU für die Berechnung des Vorgangs braucht kommt jetzt noch die Zeit, die für die Verwaltung der CPU-Zeit für mehrere Threads und das Umschalten zwischen den Threads notwendig ist. Ist die Verarbeitung jedoch von der Form, dass z.B. eine E-Mail-Nachricht abgeholt werden, Daten aus der Datenbank gelesen und eine Datei von einem FTP-Server geholt werden muss, kann sich die Aufteilung in mehrere Threads auch bei einem Rechner mit nur einer CPU lohnen. Während der eine Thread auf die Antwort des E-Mail-Servers wartet (also keine CPU-Zeit braucht), kann der andere Thread bereits mit dem Laden der Daten aus der Datenbank beginnen.

Auch Transaktionen sind für die Performanz eines Anwendungssystems eine Belastung. Sollte es nicht möglich sein, ohne Transaktionen auskommen zu können, so ist es ratsam, die Benutzung globaler Transaktionen zu vermeiden. Da bei globalen Transaktionen sehr viel Kommunikation über Netz erforderlich ist, sind lokale Transaktionen sehr viel performanter. Globale Transaktionen können vermieden werden, indem man z.B. nicht mehrere Entity-Beans in eine Transaktion einbezieht. Eine globale Transaktion ist beispielsweise auch dann erforderlich, wenn eine Entity-Bean auf mehrere Datenbanken zugreift.

Abschließend sei angemerkt, dass sich die Beispiele dieses Buches nicht immer mit den in diesem Abschnitt beschriebenen Sachverhalten decken. Die Beispiele der übrigen Abschnitte und Kapitel haben das Verständnis EJB-bezogener Sachverhalte zum Ziel, nicht jedoch die Performanzoptimierung.

9.2 Prozesse, Geschäftsobjekte und Dienste

Die architektonische Gestaltung von Anwendungen, wie komplex sie auch sein mögen, stellt Softwaredesigner immer wieder vor neue Herausforderungen. Mit jedem weiteren Schritt in Richtung einer neuen Technologie und mit jeder neuen Anforde-

rung, die aus dem Bereich der Wirtschaft an die IT-Abteilungen herangetragen wird, wächst der Designaspekt um eine neue Dimension. Enterprise JavaBeans nimmt dem Softwareentwickler durch das Komponentenmodell bereits einige Designentscheidungen ab. Für Probleme wie Verteilung, Skalierung, Persistenz etc. findet man in der Spezifikation entsprechende Antworten und man endeckt bei genauerer Betrachtung das eine oder andere bekannte Design-Pattern (z. B. entspricht das *EJBObject* einer Kombination aus Proxy- und Decorator-Pattern und das Home-Interface dem Factory-Pattern [Gamma, 1996]). Oft ist der Entwickler jedoch auf der Suche nach Antworten auf Fragen wie:

- Für welche Aufgaben verwendet man Entity-Beans?
- Für welche Aufgaben verwendet man Session-Beans?
- Für welche Aufgaben verwendet man Message-Driven-Beans?
- Kommuniziert der Client mit allen Komponenten oder nur mit bestimmten?
- Wie kooperieren und kommunizieren Komponenten?

Das Ziel dieses Abschnitts ist es, einige Anregungen in Bezug auf den Einsatz von Entity-, Session- und Message-Driven-Beans zu geben.

Um Geschäftsvorfälle und Geschäftsabläufe besser und leichter handhaben zu können, bildet man sie auf computergestützten Systemen ab. Dabei versucht man sich in der Regel an den realen Gegebenheiten zu orientieren. Die Realität im täglichen Geschäftsleben begegnet uns in Form von Prozessen, Geschäftsobjekten und Diensten. Was wir unter Prozessen, Geschäftsobjekten und Diensten verstehen und wie sie auf Enterprise JavaBeans abgebildet werden können, verdeutlichen die folgenden Ausführungen.

Geschäftsobjekte

Mit Geschäftsobjekten sind hier reale Gegenstände oder Dinge eines konkreten Anwendungsbereichs gemeint. Beispiele wären aus dem kaufmännischen Bereich etwa ein Kunde, ein Lieferant oder eine Rechnung, aus dem technischen Bereich etwa eine Maschine, eine Stückliste oder eine Fertigungsanweisung. Der unmittelbare Bezug dieser Objekte zur Geschäftswelt wird mit dem Begriff *Geschäftsobjekt* oder *Business-Objekt* untermauert. Wesentlich bei Geschäftsobjekten ist, dass sie durch ihre Daten repräsentiert und wiedererkannt werden. So hat ein Kunde einen Namen, eine Lieferanschrift, eine Rechnungsanschrift etc. In einem EDV-System wird ihm zur eindeutigen Identifikation eine Kundennummer zugeteilt. Gleiches gilt etwa für Lieferanten. Zukaufteile werden duch die Daten ihrer technischen Eigenschaften repräsentiert und durch eindeutige Teilenummern identifiziert. All diese Eigenschaften sprechen dafür, derartige Geschäftsobjekte auf Entity-Beans abzubilden (vgl. dazu Kapitel 5). Entity-Beans sind persistente Objekte und können über einen eindeutigen Schlüssel

identifiziert bzw. wiedergefunden werden. Geschäftsobjekte sind nicht isoliert zu betrachten, sondern stehen in gewissen Beziehungen zueinander. Ein Fertigungsauftrag bezieht sich in der Regel auf eine Bestellung (vertikale Beziehung), eine Stückliste setzt sich aus mehreren Modulen zusammen, die wiederum in einzelne Teile gegliedert sind (horizontale Beziehung). Business- bzw. Geschäftsobjekte spiegeln damit die *aufbauorganisatorischen Aspekte* einer Organisationsform wider.

Prozesse

Prozesse repräsentieren die *ablauforganisatorischen Aspekte* einer Organisation. Unter Prozessen verstehen wir eine definierte Abfolge von Einzeloperationen. Der Prozess befindet sich je nach Fortschritt in verschiedenen Zuständen, die es erlauben, die Prozessdurchführung zu kontrollieren und zu steuern. Die Zustandsübergänge sind ebenfalls definiert. Die Einzeloperationen von Prozessen werden auf den Daten der Business-Objekte durchgeführt (d.h., sie werden benutzt und/oder verändert). Als Beispiel eines Prozesses ließe sich hier das Erfassen einer eingegangenen Bestellung anführen.

Prozessschritt	**Zustand**
Eingang des Bestelldokuments	Anstoß des Prozesses
Überprüfung der Bestelldaten	wird geprüft
Übernahme in den Bestand unbearbeiteter Bestellungen	in Bearbeitung
Vervollständigen der Bestelldaten	in Bearbeitung
Freigabe der Bestellung	freigegeben

Tabelle 9.1: Prozessschritte zum Erfassen einer Bestellung

In diesen Prozess könnten (je nach Geschäftsfeld) die Business-Objekte *Kunde* (der Besteller), *Endprodukt* (der Gegenstand der Bestellung), *Rohmaterial* (für die Bestandskontrolle zwecks Lieferzusage) und *Bestellung* einbezogen sein.

Prozesse lassen sich ideal auf Session-Beans abbilden (vgl. dazu auch Kapitel 4). Da Prozesse zustandsbehaftet sind, muss es sich demnach um zustandsbehaftete Session-Beans handeln (stateful Session-Beans). Zustandsbehaftete Session-Beans sind transient und stehen einem Client exklusiv für eine Sitzung zur Verfügung.

Dienste

Mit Diensten sind alle Operationen gemeint, die nicht eindeutig zu einem Business-Objekt oder zu einem Business-Prozess gehören. Derartige Operationen haben eine unterstützende Funktion. Sie lassen sich objekt- oder prozessbezogen gruppieren. Dienste lassen sich auf zustandslose Session-Beans (stateless Session-Beans) oder Message-Driven-Beans abbilden. Zustandslose Session-Beans modellieren *synchrone* Dienste, Message-Driven-Beans modellieren *asynchrone* Dienste.

Synchrone Dienste finden dann Verwendung, wenn die weitere Verarbeitung vom Ergebnis der Inanspruchnahme des Dienstes abhängig ist. Spielt das Ergebnis keine Rolle oder beeinflusst das Ergebnis die weitere Ausführung des Prozesses nicht (d.h. auf das Ergebnis kann später zugegriffen werden), kann der Dienst asynchron sein. Die Inanspruchnahme asynchroner Dienste kann die Performanz verbessern, wenn die Verarbeitung des Dienstes auf einem anderen Rechner oder auf einem anderen Prozessor stattfinden kann, während die Verarbeitung des eigentlichen Prozesses parallel dazu fortgesetzt wird. Asynchrone Dienste können auch benutzt werden, um Prozesse asynchron anzustoßen bzw. um Prozesse parallel auszuführen. In einem solchen Fall nimmt eine Message-Driven-Bean gegenüber einer (stateful) Session-Bean die Rolle des EJB-Clients ein. Über die Nachricht wird die Message-Driven-Bean mit allen relevanten Informationen versorgt, die für den Anstoß und die Durchführung des Prozesses notwendig sind.

Die beiden folgenden Beispiele sollen die Abbildung von Geschäftsvorgängen auf Objekte, Prozesse und Dienste verdeutlichen.

Beispiel: Buchungssystem

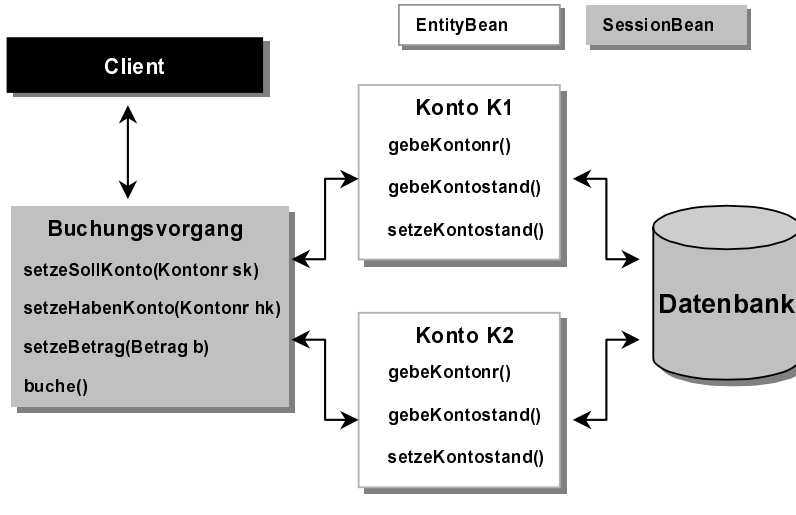

Abbildung 9.3: Beispiel Buchungssystem

Die (stateful) Session-Bean *Buchungsvorgang* in Abbildung 9.3 enthält die Logik für die Durchführung einer doppelten Buchung. Sie kann die Durchführungsreihenfolge der Einzelschritte erzwingen (z.B durch Auslösen einer Ausnahme – in Java *Exception* genannt – im Fall eines Verstoßes gegen die vorgebene Reihenfolge) und so für einen geregelten Ablauf des Prozesses sorgen. *SetzeSollKonto()* initialisiert einen neuen

Buchungsvorgang und überprüft durch einen *findByPrimaryKey*-Aufruf, ob für die übergebene Kontonummer ein Konto existiert. Zugleich beschafft sich die Session-Bean damit eine Referenz auf eine Entity-Bean, um die Buchung auf diesem Konto später durchführen zu können. Gleiches gilt für *setzeHabenKonto()*. Beim Aufruf von *setzeBetrag()* kann die Session-Bean verschiedene Bedingungen überprüfen, z.B ob ein Konto überzogen wird. Der Aufruf von *buche()* wird einen Transaktionskontext öffnen, der die beteiligten Konten einbezieht, und wird die Buchung über *setzeKontostand()* ausführen. Ein Konto wird entsprechend den obigen Ausführungen durch eine Entity-Bean repräsentiert, da ein Konto ein Objekt mit einem dauerhaften Zustand ist. Die Komponente bietet eine entsprechende Schnittstelle an, um den Zustand des Kontos modifizieren zu können.

Der Vorgang für eine doppelte Buchung kann über die (stateful) Session-Bean von jedem Client genutzt werden. Die Logik ist im gesamten System einmal an zentraler Stelle vorhanden. Ferner kommuniziert der Client ausschließlich über die Session-Bean. Für die Buchung ist es nicht notwendig, direkt mit den Entity-Beans zu kommunizieren. In diesem Fall würde die Session-Bean den Remote-Client-View, die Entity-Bean den Local-Client-View unterstützen.

Eine Mindestanforderung an ein noch so primitives Buchhaltungssystem ist die Führung eines Buchungsjournals, in dem alle Buchungsvorgänge protokolliert werden und mit dessen Hilfe somit Kontostände nachvollzogen werden können. Die Protokollierung von Einzelbuchungen ist insbesondere dann wichtig, wenn Buchungen zu einem späteren Zeitpunkt storniert werden müssen. Das Buchungsjournal setzt sich aus einzelnen Einträgen zusammen. Ein Eintrag besteht im Minimalfall aus einer eindeutigen Buchungsnummer, aus dem Soll- und dem Habenkonto, dem Buchungsbetrag, dem Buchungsdatum, der Uhrzeit der Buchung sowie dem Benutzer, der die Buchung durchgeführt hat. Das Anfügen eines Journaleintrags sowie das Abfragen von Informationen aus dem Journal ist kein Prozess, sondern eine zustandslose Aneinanderreihung von Einzeloperationen. Das Buchungsjournal ist auch nicht als typisches Business-Objekt einzustufen. Da das Buchungsjournal gigantische Größen annehmen kann, wäre die Abbildung als Entität unter Performanzgesichtspunkten ohnehin kritisch. Das Buchungsjournal ist in unserem Sinne als klassischer Dienst zu verstehen, der das Protokollieren von Buchungsvorgängen und die Recherche in dem dadurch entstandenen Datenbestand anbietet.

Die Message-Driven-Bean *JournalEintrag* und die (stateless) Session-Bean *JournalService* (vgl. Abbildung 9.4) stellen die Schnittstelle zum Buchungsjournal dar. Das Buchungsjournal ist im Beispiel in der Datenbank abgelegt. Ebenso wäre beispielsweise die Ablage in einem Archivsystem denkbar. Das Schreiben eines Journaleintrags ist ein asynchroner Dienst und kann über das Schicken einer Nachricht an die *JournalEintrag*-Bean durchgeführt werden. Ein asynchroner Dienst hat den Vorteil, dass die Bean *Buchungsvorgang* nicht warten muss, bis die Daten gespeichert wurden. Noch während

des Schreibens eines Journaleintrags kehrt der Aufruf wieder zum Client zurück. Die Bean *Buchungsvorgang* erwartet von diesem Vorgang außerdem kein Ergebnis. Der Java Message Service sorgt für die notwendige Datensicherheit. Um Daten im Buchungsjournal zu recherchieren, wird ein synchroner Dienst in Form der *JournalService*-Bean benutzt. Für die Recherche könnte zum einen eine begrenzte Anzahl von Methoden mit fester Funktionalität vorgegeben werden (wie der Einfachheit halber in obigem Bild angedeutet ist), zum anderen wäre auch die Entwicklung einer primitiven Abfragesprache denkbar. Für umfangreiche und langdauernde Recherchen könnte ebenfalls ein asynchroner Dienst eingesetzt werden. Nachdem der Client die Recherche angestoßen hat, könnte er mit anderen Aktionen fortfahren. Das Ergebnis der Recherche wird ihm dann später z. B. über den Java Message Service oder eine E-Mail mitgeteilt.

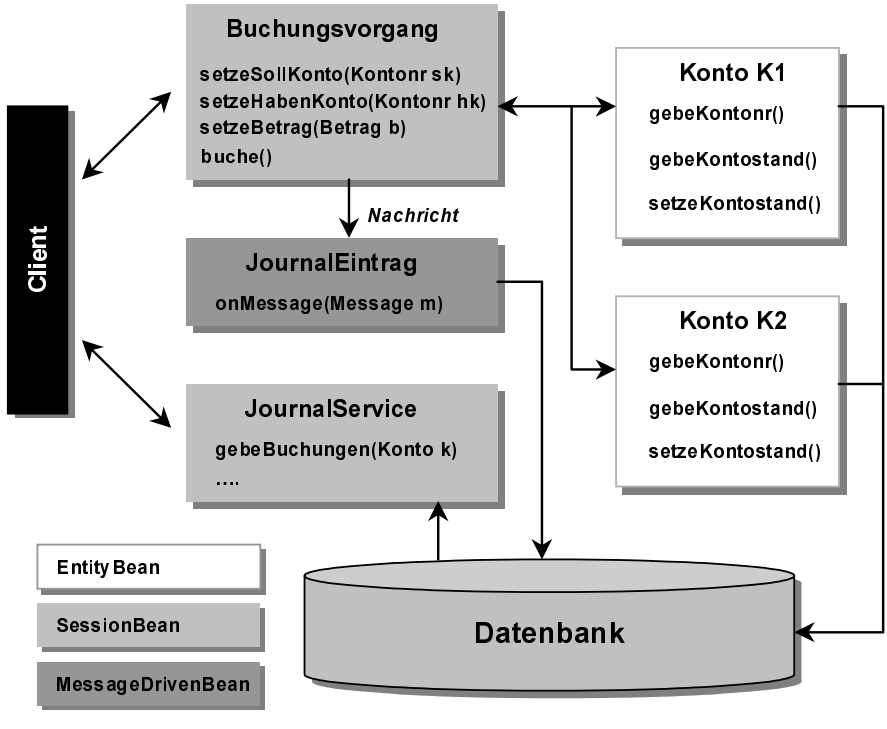

Abbildung 9.4: Beispiel Buchungsvorgang mit Erweiterung

Beispiel: Fertigungsüberwachung

Die (stateful) Session-Bean *Fertigungsüberwachung* in Abbildung 9.5 enthält die Logik für die Überwachung des Fertigungsprozesses eines Produkts, das von einer einzigen Maschine gefertigt wird. Der Client ist in diesem Beispiel eine (real existierende) Maschine, die das Produkt P1 fertigen kann und direkt an das System angebunden ist

(z.B. über eine Java- oder CORBA-Schnittstelle). Mit in den Prozess einbezogen werden die Business-Objekte *Maschine* (die logische Repräsentanz der Maschine im System), *Produkt* (welches von der Maschine gefertigt wird, ebenfalls in Form der logischen Repräsentanz) und *Fertigungsauftrag* (der sich auf das Produkt bezieht und detailliert vorgibt, welche Stückzahl in welchem Zeitraum von welcher Maschine zu bearbeiten ist).

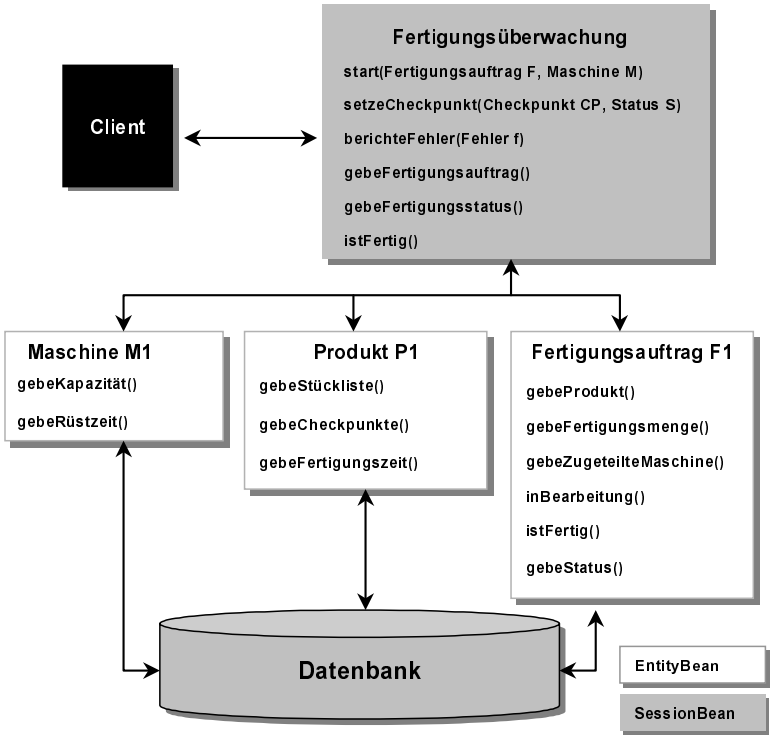

Abbildung 9.5: Beispiel Fertigungsüberwachung

Über einen Service beispielsweise bekommt die (reale) Maschine alle Fertigungsaufträge, die ihr durch einen Optimierungsprozess zugeteilt wurden. Dadurch weiß sie, welches Produkt in welcher Stückzahl gefertigt werden muss. Jeder Fertigungsauftrag wird von der (realen) Maschine unter Benutzung der Session-Bean *Fertigungsüberwachung* abgearbeitet. Die Session-Bean kann zunächst beim Aufruf der Methode *start()* überprüfen, ob die (reale) Maschine mit der im Fertigungsauftrag zugeteilten (logischen) Maschine übereinstimmt. Im Verlauf des Fertigungsprozesses werden von der (realen) Maschine über *setzeCheckpunkt()* Meldepunkte gesetzt (z.B. kann nach jedem Prozessschritt, nach der Fertigstellung eines Stücks oder nach der Fertigstellung einer bestimmten Losgröße ein Checkpunkt gesetzt werden). Die Session-Bean vergleicht

die gesetzten Meldepunkte mit der durch das Produkt P1 vorgegebenen Liste von Meldepunkten (*Produkt.gebeCheckpunkte()*). Die vorgegebenen Meldepunkte ergeben sich aus der technischen Spezifikation des Produktes und werden von Technikern in das System bzw. in die Maschinenkonfiguration eingepflegt. Für den Fall, dass die Meldepunkte nicht zusammenpassen oder sonstige Produktionsstörungen vorliegen (die von der realen Maschine über *berichteFehler()* gemeldet werden), kann die Session-Bean entsprechende Benachrichtigungsmechanismen in Gang setzen (z. B. kann über Pager ein Techniker informiert werden). Nachdem die entsprechende Stückzahl abgearbeitet ist, wird der Fertigungsauftrag als *fertig* zurückgemeldet. Somit kann von anderer Seite aus der Fortschritt des gesamten Fertigungsprozesses überwacht werden.

Es lassen sich nicht für jeden Sachverhalt eines bestimmten Geschäftsfeldes Prozesse (abbildbar über stateful Session-Beans), Business-Objekte (abbildbar über Entity-Beans) oder Services (abbildbar über stateless Session-Beans oder Message-Driven-Beans) definieren. Ebensowenig ist jede Entity-Bean automatisch ein Business-Objekt, eine zustandsbehaftete Session-Bean ein Prozess oder eine zustandslose Session-Bean bzw. eine Message-Driven-Bean ein Dienst. Die Abbildung eines Problems auf Enterprise JavaBeans dürfte jedoch erheblich leichter fallen, wenn man bei der Problemanalyse stets die Semantik dieser drei Einheiten im Hinterkopf behält.

9.3 Kopplung von Enterprise-Beans (Aggregation)

Die Thematik, die wir in diesem Abschnitt ansprechen wollen, bezieht sich auf die (mitunter laufzeitdynamische) Kopplung von Enterprise-Beans bzw. die Schaffung eines Kooperationsrahmens für Enterprise-Bean-Komponenten auf der *Serverseite*. Wir wollen verschiedene Standardalternativen kurz erörtern, um dann im weiteren Verlauf dieses Abschnitts eine anwendungsbezogene Kopplungsmöglichkeit vorzustellen.

Alternative 1: *Statische Kopplung über die Sicht des Clients*

Eine (statische) Kopplung der Komponenten ist möglich, indem sie sich über die Sicht eines Clients benutzen (vgl. 4.2.3). Im Abschnitt *ejb-ref* des Deployment-Deskriptors werden Informationen über den Typ derjenigen Enterprise-Beans hinterlegt, die von der zu installierenden Bean über die Client-Sicht benutzt werden (vgl. 4.3.7). Durch das Feld *ejb-link* bindet der Application-Assembler diese Referenzen an konkrete, installierte Implementierungen dieser Bean-Typen. Durch den Einsatz den Local-Client-Views kann die Kommunikation zwischen den Enterprise-Beans optimiert werden, wenn sie im selben Applikationsserver installiert sind.

Alternative 2: *Dynamische Kopplung über die Metadaten-Schnittstelle*

Für eine laufzeitdynamische Kopplung bietet die Spezifikation der Enterprise JavaBeans die Metadaten-Schnittstelle an (*javax.ejb.EJBHome.getEJBMetaData*). In Kombination mit der Java-Reflection-API besteht die Möglichkeit, dynamische Methodenaufrufe

an Enterprise-Beans zu programmieren. Eine derartige Kopplung ist sehr generisch und relativ aufwändig in der Programmierung. Die Überprüfung der Datentypen bei der Kompilierung geht dadurch verloren. Außerdem ist der Einsatz der Java-Reflection-API für die Ausführungsgeschwindigkeit stets kritisch. Man wünscht sich oft einen weniger generischen, dafür aber performanteren Weg, um Enterprise-Beans zur Laufzeit dynamisch koppeln zu können. Ausserdem ist diese Art der dynamischen Kopplung auf den Remote-Client-View beschränkt. Die Methode *getEJBMetaData* ist in *javax.ejb.EJBLocalHome* nicht verfügbar.

Alternative 3: *Dynamische Kopplung über Events*

Eine gängige Art der dynamischen Kopplung von Komponenten sind beispielsweise Events (vergleiche dazu auch Abschnitt 9.5). Ein Event-Modell, ähnlich dem der JavaBeans, bei dem die Enterprise-Beans über den EJB-Container Ereignisse auslösen und Ereignisse empfangen können, wäre ideal für die dynamische Kopplung von Enterprise-Bean-Komponenten zur Laufzeit. Wenn das Auslösen und das Empfangen eines Events nur über den EJB-Container möglich ist, kommt es auch nicht zu Konflikten mit dem Lebenszyklusmanagment oder mit Client-Aufrufen. Der EJB-Container kann z. B. Client-Aufrufe, die Passivierung oder die Löschung einer Enterprise-Bean solange blockieren, bis die Verarbeitung eines empfangenen Events abgearbeitet ist bzw. die Zustellung von Events an passivierte oder gelöschte Enterprise-Beans unterdrücken. Die Spezifikation der Enterprise JavaBeans sieht kein Event-Modell für Enterprise-Beans vor, daher ist eine Kopplung über Events nicht möglich.

Alternative 4: *Dynamische Kopplung über Nachrichten*

Eine andere gängige Art der dynamischen Kopplung ist der Austausch von Nachrichten über einen Messaging-Dienst wie dem Java Message Service. Allerdings ist der Java Message Service auf die Kommunikation zwischen Prozessen über das Netzwerk ausgelegt. Für eine rein prozessinterne Kommunikation ist er wenig effizient. Außerdem kann eine Entity- oder eine Session-Bean die asynchrone Zustellung von Nachrichten über das *javax.jms.MessageListener*-Interface (vgl. Kapitel 6.2) nicht nutzen, da die Spezifikation dies ausdrücklich verbietet. Würde sie die Benutzung dieses Mechanismus für Entity- und Session-Beans zulassen, könnte es zu Konflikten zwischen dem Lebenszyklusmanagement des EJB-Containers und der Nachrichtenzustellung durch den JMS-Provider kommen. Außerdem wäre damit die Existenzberechtigung für Message-Driven-Beans in Frage gestellt. Für eine dynamische Kopplung von Enterprise-Beans ist der Java Message Service daher nicht geeignet.

Alternative 5: *Statische Kopplung über die Schaffung einer Enterprise-Bean*

Der Application-Assembler kann eine Kopplung der Enterprise-Beans mit den Mitteln der Spezifikation ermöglichen, indem er eine neue Enterprise-Bean programmiert. Der Client würde die Enterprise-Bean des Application-Assembler benutzen, welche ihrerseits andere Beans für die Erfüllung der gewünschten Aufgaben benutzt beziehungsweise deren Kooperation ermöglicht.

Alternative 6: *Kopplung von Entity-Beans über persistente Beziehungen*

Persistente Beziehungen (container-managed Relationships) sind in gewisser Weise auch ein Mittel, um Enterprise-Bean Komponenten zu Aggregaten zusammenzufassen. Es handelt sich einerseits um eine statische Kopplung, weil die Beziehungen im Deployment-Deskriptor festgelegt werden und damit auf bestimmte Typen von Enterprise-Beans beschränkt sind. Die eigentliche Kopplung zwischen den Instanzen der an der Beziehung beteiligten Enterprise-Beans ist aber dynamisch, da die Verbindung zwischen den Instanzen (die z. B. über eine Fremdschlüsselbeziehung in der Datenbank realisiert wird) zur Laufzeit hergestellt bzw. gelöscht werden kann. Das Beispiel Lagerverwaltung in Kapitel 5 veranschaulicht diesen Sachverhalt. Die Kommunikation zwischen den auf diese Weise aggregierten Enterprise-Beans erfolgt über den Local-Client-View und ist damit relativ effizient. Die Auflösung der über die Beziehung enstandenen Referenzen zwischen den Entity-Bean-Instanzen übernimmt dabei der EJB-Container. Persistente Beziehungen sind damit die einzige Form von Aggregation, die vom EJB-Container aktiv unterstützt wird. Diese Art der Kopplung ist allerdings nur für Entity-Beans relevant.

Neben der Alternative 6 sind auch die Alternativen 1, 2 und 5 für die Kopplung von Enterprise-Beans relevant. Wir wollen im restlichen Teil dieses Abschnitts eine weitere Alternative für die Kopplung von Enterprise-Beans und die Schaffung eines Kooperationsrahmens zeigen.

Durch ein serverseitiges, *anwendungsbezogenes* Framework kann ein Kooperationsrahmen geschaffen werden, der es ermöglicht, Enterprise-Beans dynamisch zu verbinden und gegebenenfalls auszutauschen, ohne dass andere Beans davon betroffen sind. Zur Veranschaulichung soll ein anwendungsbezogenes Framework für eine einfache Buchungsanwendung entwickelt werden. Dieses anwendungsorientierte Framework (das die fachlichen Aspekte bedient) setzt auf das systemtechnisch orientierte Framework Enterprise JavaBeans (das die technischen Belange abdeckt) auf und bietet die Möglichkeit zur losen und laufzeitdynamischen Kopplung von Enterprise-Beans (ohne dafür eine neue Bean programmieren zu müssen). Dazu verwenden wir das Beispiel des Buchungsvorgangs aus Abschnitt 2 dieses Kapitels.

Zunächst soll die herkömmliche Art der Implementierung dargestellt werden, um die Unterschiede zur Verwendung eines Frameworks (die im Anschluss gezeigt wird) deutlich zu machen.

Abbildung 9.6 zeigt die Session-Bean *Buchungsvorgang*, die dem Client eine Schnittstelle für die Durchführung einer doppelten Buchung zur Verfügung stellt. Sie benötigt die Entity-Bean *Konto* und die Session-Bean *JournalService*, um die Buchung auszuführen.

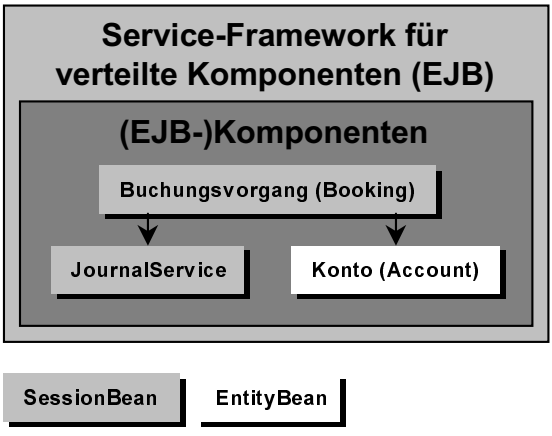

Abbildung 9.6: Beispiel Buchungsvorgang

Der Code eines herkömmlichen Clientprogramms könnte folgendermaßen aussehen:

```
...
InitialContext ctx = new InitialContext();
//Holen des Home-Interfaces der Session-Bean Booking
final String BOOKING = "java:comp/env/ejb/Booking";
Object o = ctx.lookup(BOOKING);
BookingHome bh = (BookingHome)
       PortableRemoteObject.narrow(o, BookingHome.class);
//Erzeugen der BookingBean und Holen des Remote-Interface
Booking b = bh.create();
//Durchführen der Buchung
b.setDebitAccount("0815");
b.setCreditAccount("0915");
b.setAmount(100);
b.book();
b.remove();
...
```

Listing 9.2: Beispiel für einen herkömmlichen Client

Die Implementierung der entsprechenden Methoden des Remote-Interface in der *BookingBean*-Klasse sähe nach der herkömmlichen Vorgehensweise etwa so aus:

```
...
private Account            debitAccount;
private Account            creditAccount;
private AccountHome        accountHome;
private JournalServiceHome journalHome;
...
public void ejbCreate()
  throws RemoteException, CreateException
{
```

```
    final String ACCHOME = "java:comp/env/ejb/Account";
    final String JHOME   = "java:comp/env/ejb/JournalService";
    ...
    try {
     //Holen der Home-Interfaces der Entity-Bean Konto und
     //der Session-Bean JournalService
     Context ctx = new InitialContext();
     Object o = ctx.lookup(ACCHOME);
     accountHome = (AccountHome)
         PortableRemoteObject.narrow(o, AccountHome.class);
     o = ctx.lookup(JHOME);
     journalHome = (JournalServiceHome)
         PortableRemoteObject.narrow
                    (o,JournalServiceHome.class);
    }
    ...
  }
...
  public void setDebitAccount(String accno)
    throws RemoteException, BookingException
  {
   ... //Prüfroutinen
   //Erzeugen eines PrimaryKey-Objekts
   AccountPK pk = new AccountPK(accno);
   try {
    //Suchen des Sollkontos
    debitAccount = accountHome.findByPrimaryKey(pk);
   }
   ... //Fehlerbehandlung
  }

  public void setCreditAccount(String accno)
    throws RemoteException, BookingException
  {
   ... //Prüfroutinen
   //Erzeugen eines PrimaryKey-Objekts
   AccountPK pk = new AccountPK(accno);
   try {
    //Suchen des Habenkontos
    creditAccount = accountHome.findByPrimaryKey(pk);
   }
   ... //Fehlerbehandlung
  }
  ...
  public void setAmount(float amount)
    throws RemoteException, BookingException
  {
   ... //Prüfroutinen
   try {
    //Überprüfung, ob die Buchung des Betrags zulässig ist.
    debitAccount.checkAmount(amount * (-1));
    creditAccount.checkAmount(amount);
```

```
    }
    ... //Fehlerbehandlung
  }
  ...
  public void book()
    throws RemoteException, BookingException
  {
   ... //Prüfroutinen
   //Buchung des Betrags durch Verändern der Kontostände
   debitAccount.setBalance(debitAccount.getBalance()-
                                theAmount);
   creditAccount.setBalance(creditAccount.getBalance()+
                                theAmount);
   ... //Fehlerbehandlung
   //Protokollieren des Vorgangs im Journal
   JournalService js = journalHome.create();
   js.record(debitAccount,
             creditAccount,
             theAmount,
             theContext.getCallerPrincipal());
   ... //Fehlerbehandlung
  }
  ...
```

Listing 9.3: Beispiel für eine herkömmliche Session-Bean-Implementierung

Die Abhängigkeit der Session-Bean *Buchungsvorgang* (*Booking*) von der Entity-Bean *Konto* (*Account*) und der Session-Bean *JournalService* wid im Deployment-Deskriptor in Listing 9.4 dokumentiert.

```
…
<ejb-jar>
  <description>
    Dieses Jar-File beinhaltet alle Komponenten,
    die für die Buchhaltungsanwendung benötigt werden.
  </description>

    <enterprise-beans>
      <session>
        <description>
          Die Session-Bean Buchungsvorgang stellt
          eine Implementierung für die Durchführung
          einer doppelten Buchung zur Verfügung.
        </description>
        <ejb-name>Booking</ejb-name>
        ...
        <ejb-ref>
          <ejb-ref-name>ejb/JournalService</ejb-ref-name>
          <ejb-ref-type>Session</ejb-ref-type>
          <home>ejb.beispiel.JournalServiceHome</home>
          <remote>ejb.beispiel.JournalService</remote>
```

```
            <ejb-link>JournalService</ejb-link>
          </ejb-ref>
            <ejb-ref-name>ejb/Konto</ejb-ref-name>
            <ejb-ref-type>Entity</ejb-ref-type>
            <home>ejb.beispiel.KontoHome</home>
            <remote>ejb.beispiel.Konto</remote>
            <ejb-link>Account</ejb-link>
          </ejb-ref>
          ...
        </session>
        ...
      </enterprise-beans>
...
</ejb-jar>
```

Listing 9.4: Beispiel herkömmlicher Deployment-Deskriptor

Wie man aus den Codefragmenten ersehen kann, arbeitet die Session-Bean *Buchungsvorgang* direkt mit den anderen Beans zusammen. Auch im Deployment-Deskriptor ist der Typ der verwendeten Beans fest definiert und muss mit dem verwendeten Typ der Implementierung übereinstimmen. Falls nun die Notwendigkeit besteht, z. B. die *Konto*-Bean aufgrund mangelnder Fähigkeiten durch eine Bean anderen Typs zu ersetzen oder je nach Bedarf verschiedene Typen eines Kontos (die ebenfalls als Komponenten zur Verfügung stehen) zu benutzen, wäre das nur über eine Änderung des Codes der Session-Bean *Buchungsvorgang* möglich. Über das *ejb-link*-Element des Deployment-Deskriptors bestünde noch die Möglichkeit, die Bean-Referenz an eine andere Implementierung des gleichen Typs zu binden. Allerdings wäre dann eine Neuinstallation der Session-Bean *Buchungsvorgang* notwendig.

Um die starre Bindung der Beans untereinander aufzuheben, werden sie in ein anwendungsspezifisches Framework eingebettet, über das sie auf abstrakter Ebene zusammenarbeiten können (vgl. Abbildung 9.7).

Die Session-Bean *Buchungsvorgang* ist der Geschäftsprozess, den der Client bedienen will, und der somit der Einstieg in den Server bleibt. Die Benutzung der *JournalService*-Bean und der *Konto*-Bean soll über die abstrakten Schnittstellen eines Frameworks erfolgen, das dem Buchungsvorgang die Einbeziehung von Konten und eines Journal-Services erlaubt.

Das Framework, wie es Abbildung 9.8 darstellt, besteht in unserem Beispiel lediglich aus den drei gezeigten Interfaces. Konkretisiert und anwendbar wird das Framework durch die Implementierung dieser Interfaces in Java-Klassen. Die Funktionalität, die wir von einem Konto und einem Journal-Service erwarten, steckt bereits in den Enterprise-Beans. Um die Brücke zwischen der Frameworkimplementierung und den Enterprise-Beans zu schlagen, bedienen wir uns des Bridge-Patterns (siehe [Gamma, 1996]). Dadurch können die verschiedenen Abstraktionen (Konto, Journal) von ihren Implementierungen entkoppelt werden und bleiben so variierbar.

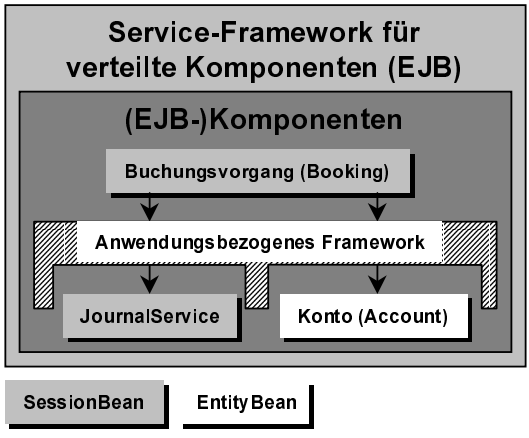

Abbildung 9.7: Buchungsvorgang mit Framework

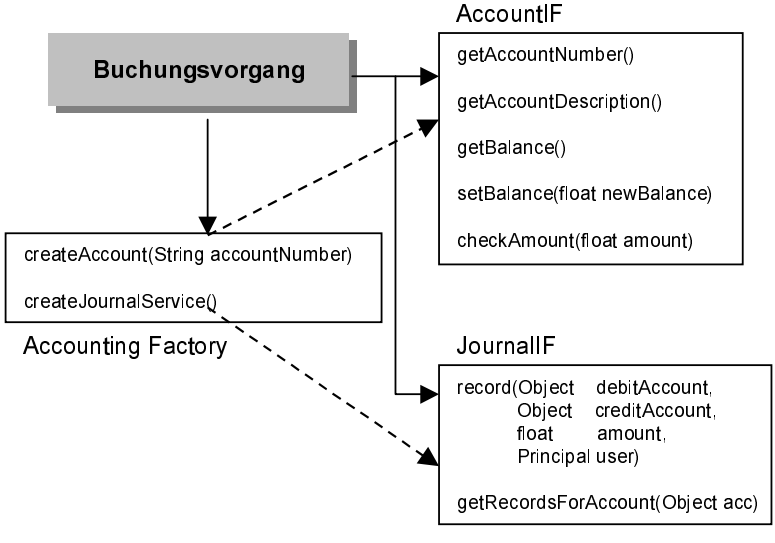

Abbildung 9.8: Anwendungsbezogenes Kooperationsframework

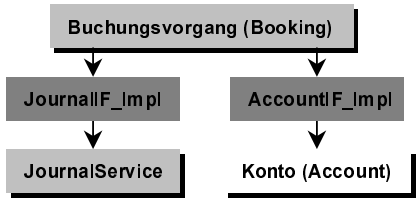

Abbildung 9.9: Beispielkooperation im Framework

Abbildung 9.9 zeigt die Klassen *JounalIF_Impl* und *AccountIF_Impl*. Sie werden auf Initiative der Session-Bean *Buchungsvorgang* durch die Implementierung des Interface *AccountingFactory* erzeugt und für die notwendigen Operationen benutzt. Die Implementierungsklassen delegieren die jeweiligen Methodenaufrufe an andere Enterprise-Beans (in unserem Fall an die *JournalService*-Bean und die *Konto*-Bean). Damit sind die Beans *JournalService* und *Konto* völlig von der Komponente *Buchungsvorgang* entkoppelt. Von den Bridge-Klassen hängt es letzten Endes ab, welche Enterprise-Beans tatsächlich benutzt werden. Vor allem kann (durch eine entsprechende Implementierung) diese Entscheidung zur Laufzeit unter Beachtung der momentanen Umstände getroffen werden.

Die Referenzen auf die *Konto*- und die *JournalService*-Bean müssen *nicht* mehr im Deployment-Deskriptor der Session-Bean *Buchungsvorgang* als Bean-Referenz angegeben werden. Der Zugriff auf die beiden Beans von der Session-Bean *Buchungsvorgang* aus erfolgt nicht mehr über JNDI, sondern über das Framework. Empfehlenswert ist jedoch, im Deployment-Deskriptor zu vermerken, dass die Session-Bean die Implementierungsklassen des Frameworks nutzt. Der Deployment-Deskriptor sollte auf jeden Fall in seiner Eigenschaft als zentrale Installationsanweisung genutzt werden. Der sich daraus ergebende Dokumentationscharakter ist von großem Nutzen. Leider gibt es in den Abschnitten des Deployment-Deskriptors keinen Ort, an dem sich die Factory-Klasse (die der Einstieg in das Framework ist) eintragen ließe. Hinsichtlich der Konfigurationsmöglichkeiten wäre ein Eintrag als Resource-Factory-Reference (vgl. [Sun Microsystems, 2000]) am besten geeignet. Die Implementierungsklasse des *AccountingFactory*-Interfaces und unser Framework entspricht jedoch in keiner Weise dem, was die Spezifikation unter einer Resource-Factory und einer Ressource versteht. Wir werden uns deshalb auf einen Hinweis im Beschreibungsfeld der Session-Bean Buchungsvorgang beschränken.

```
…
<ejb-jar>
  <description>
    Dieses Jar-File beinhaltet alle Komponenten,
    die für die Anwendung Buchhaltung benötigt werden.
  </description>

    <enterprise-beans>
      <session>
        <description>
          Die Session-Bean Buchungsvorgang stellt
          eine Implementierung für die Durchführung
          einer doppelten Buchung zur Verfügung.
          HINWEIS: Diese Bean nutzt das
          Accounting-Framework !
        </description>
        <ejb-name>Booking</ejb-name>
```

```
        ...
      </session>
      ...
    </enterprise-beans>
</ejb-jar>
```

Listing 9.5: Beispiel Deployment-Deskriptor mit Framework

Die Implementierungsklasse des *AccountingFactory*-Interface (*FactoryImpl*) wird als Singleton (vgl. [Gamma et al., 1996]) implementiert (sie wird im Anschluss an die geänderte Implementierung der *Buchungsvorgang*-Klasse behandelt). Dadurch wird die Erzeugung einer Instanz durch die Bean-Klasse vermieden und die Verwaltung der Framework-Klassen kann an zentraler Stelle im Prozess vollzogen werden. Die Implementierung in der Bean-Klasse *Buchungsvorgang* würde sich folgendermaßen ändern:

```
...
  private AccountFactory   theFactory;
  private AccountIF        debitAccount;
  private AccountIF        creditAccount;
  ...
  public void ejbCreate()
    throws RemoteException, CreateException
  {
    ...
    //Holen der FactoryImplementierung
    theFactory = FactoryImpl.getFactory();
    ...
  }
...
  public void setDebitAccount(String accno)
    throws RemoteException, BookingException
  {
   ... //Prüfroutinen
   try {
    //Suchen des Sollkontos
       debitAccount = theFactory.createAccount(accno);
   }
       .. //Fehlerbehandlung
  }

  public void setCreditAccount(String accno)
      throws RemoteException, BookingException
  {
   ... //Prüfroutinen
   try {
    //Suchen des Habenkontos
    creditAccount = theFactory.createAccount(accno);
   }
   ... //Fehlerbehandlung
  }
```

```
...
public void setAmount(float amount)
  throws RemoteException, BookingException
{
  //Keine Änderungen
 ...
}
...
public void book()
  throws RemoteException, BookingException
{
 ...
 //Keine Änderungen
 ...
 //Protokollieren des Vorgangs im Journal
 JournalIF js = theFactory.createJournalService();
 js.record(debitAccount,
           creditAccount,
           theAmount,
           theContext.getCallerPrincipal());
 ...
}
...
```

Listing 9.6: Beispiel Session-Bean mit Framework

Die Implementierung des *AccountingFactory*-Interface wird in Listing 9.7 dargestellt.

```
...
public class FactoryImpl implements AccountingFactory,
                                    java.io.Serializable
{

    private static AccountingFactory theFactory =
                                        new FactoryImpl();

    private FactoryImpl() {}

    public static AccountingFactory getFactory() {
        return theFactory;
    }

    public AccountIF createAccount(String accountNumber)
        throws AccountException
    {
        return new AccountIF_Impl(accountNumber);
    }

    public JournalIF createJournalService()
        throws JournalException
    {
```

```
        return new JournalIF_Impl();
    }
}
```

Listing 9.7: Beispiel Factory-Interface-Implementierung

Die Implementierung des Interface *java.io.Serializable* ist notwendig, damit Probleme bei der Passivierung von zustandsbehafteten Session-Beans, die eine Referenz auf ein Objekt dieser Klasse in einer Member-Variablen speichern, vermieden werden.

Die Klassen *AccountIF_Impl* und *JournalIF_Impl* implementieren die jeweiligen Interfaces (inklusive des Interface *java.io.Serializable*) und delegieren die Aufrufe an die entsprechenden Beans. Zur Veranschaulichung zeigen wir in Listing 9.8 den Konstruktor und eine Methode der Klasse *AccountIF_Impl*.

```
...
public class AccountIF_Impl implements AccountIF,
                                       java.io.Serializable
{
    //Das Remote-Interface der Account-Bean
    Account theAccount;

    public AccountIF_Impl(String accountNumber)
        throws AccountException
    {
        final String ACCHOME = "java:comp/env/ejb/Account";

        AccountPK pk = new AccountPK(accountNumber);
        //An dieser Stelle kann zur Laufzeit entschieden
        //werden, welche Bean benutzt werden soll
        try {
            AccountHome home;
            InitialContext ctx = new InitialContext();
            Object o = ctx.lookup(ACCHOME);
            home = (AccountHome)
                      PortableRemoteObject.narrow
                          (o, AccountHome.class);
            theAccount = home.findByPrimaryKey(pk);
        }
        catch(NamingException nex) {
            throw new AccountException(nex.getMessage());
        }
        catch(FinderException fex) {
            throw new AccountException(fex.getMessage());
        }
        catch(RemoteException rex) {
            throw new AccountException(rex.getMessage());
        }
    }
  ...
```

```
    public String getAccountNumber()
        throws AccountException
    {
        String ret;
        try {
            ret = theAccount.getAccountNumber();
        }
        catch(RemoteException rex) {
            throw new AccountException(rex.getMessage());
        }
        return ret;
    }

    public String getAccountDescription()
        throws AccountException
    {
        String ret;
        try {
            ret = theAccount.getAccountDescription();
        }
        catch(RemoteException rex) {
            throw new AccountException(rex.getMessage());
        }
        return ret;
    }
  ...
}
```

Listing 9.8: Beispiel Interface-Implementierung

Zusammenfassung

Ein anwendungsbezogenes Framework auf der Serverseite bietet die Möglichkeit, Enterprise-Beans lose und laufzeitdynamisch zu koppeln. Es lässt sich ein anwendungsbezogener Kooperationsrahmen schaffen. Darüber hinaus ist es dem Application-Assembler möglich, zusätzlichen, für die korrekte Kopplung der Enterprise-Beans notwendigen Code (so genannten *Glue-Code*) einzubringen. Diese Möglichkeit besteht für ihn auch dann, wenn er zu Zwecken der Kopplung eine eigene Enterprise-Bean entwickelt. Die Enterprise-Beans bleiben dank des anwendungsbezogenen Frameworks (auch nach dem Deployment) austauschbar, ohne den Deployment-Vorgang wiederholen oder den Code einer Enterprise-Bean ändern zu müssen. Zur Laufzeit kann entschieden werden, welche Komponenten benutzt werden. Dies ist beispielsweise dann sinnvoll, wenn es nicht eine *Konto*-Bean gibt, sondern mehrere und anhand des Nummernkreises der Kontonummer entschieden werden muss, welcher Bean-Typ verwendet wird (z.B. Devisenkonto oder Wertpapierkonto). Die Handhabung unterschiedlicher Kontotypen bleibt damit ebenfalls vor der Session-Bean verborgen.

Das Framework müsste in unserem Fall vom Provider der Session-Bean *Buchungsvorgang* (*Booking*) geliefert werden (da er direkt gegen dessen Schnittstellen programmiert). Der Application-Assembler würde die Interfaces des Framework entsprechend implementieren und die durch die Implementierungsklassen verwendeten Beans ebenfalls im Server installieren. Empfehlenswert ist es, in den Beschreibungsfeldern der Beans zu vermerken, dass sie von den Frameworkklassen benutzt werden, um die zentrale Dokumentationsfunktion des Deployment-Deskriptors aufrecht zu erhalten. Der Application-Assembler entscheidet durch seine Implementierung auch, welche Enterprise-Beans als Konten bzw. Journal-Service benutzt werden und kann weiteren, für die Anwendung notwendigen Code dort unterbringen.

9.4 Vererbung

Vererbung spielt in der objektorientierten Programmierung eine große Rolle. Der Nutzen liegt im Wesentlichen in der Möglichkeit zur Wiederverwendung von bestehendem Code sowie in der (durch Vererbung entstehenden) Polymorphie von Objekten. Wie beim Komponentenparadigma treffen wir auch bei diesem Thema auf den Aspekt der Wiederverwendung. Während die Enterprise-JavaBeans-Spezifikation 1.0 das Thema Vererbung an sich gar nicht behandelt, ist ihm in der Version 1.1 ein Absatz des Anhangs gewidmet. Dort wird erwähnt, dass das Konzept der Vererbung bei Komponenten nicht beschrieben wird. Dem Bean-Entwickler wird jedoch erlaubt, die Standardmechanismen der Sprache Java zu benutzen, nämlich die Vererbung von Interfaces und Klassen. Gleiches gilt für Version 2.0 der EJB-Spezifikation. Allerdings wird dort erwähnt, dass Komponentenvererbung ein Thema in einer der nächsten Versionen der EJB-Spezifikation sein wird (vgl. [Sun Microsystems, 2001]).

Führt man sich die wesentlichen Aspekte einer Komponente vor Augen (siehe Kapitel 2), so drängt sich die Frage auf, welchen Sinn die Vererbung von Komponenten hat. Komponenten sind in sich geschlossene Bausteine, die eine bestimmte Funktionalität zur Verfügung stellen und deren Verhalten nur innerhalb eines gewissen Rahmens über eine Konfiguration von außen anpassbar sein soll. Die Komponente verbirgt ihr Inneres und repräsentiert sich anderen gegenüber nur durch ihre Schnittstelle (sog. Black-Box). Ein Framework z. B. ist typischerweise ein so genannter White-Box-Ansatz, d.h., man kann sich Einblick verschaffen, man sieht die Architektur und die beteiligten Klassen und es ist sogar notwendig, durch Vererbung eine Spezialisierung vorzunehmen, um nutzbare Funktionalität aus dem Framework zu gewinnen. Bei der Implementierung einer Komponente ist es sicherlich gängige Praxis, auf die Klassen- und Interfacevererbung zurückzugreifen. Wie aber sieht es mit der Vererbung bezogen auf Komponenten (bzw. in unserem Fall Enterprise-Beans) selbst aus?

Wir wollen diese Frage am Beispiel eines Bankkontos diskutieren. Dieses gibt es z.B. in den Ausprägungen Girokonto, Sparkonto und Devisenkonto. Alle Arten haben gemeinsame Eigenschaften (sie alle besitzen eine Kontonummer und einen Kontostand) und allen ist eine gewisse Funktionalität gemeinsam (man kann den aktuellen Bestand abfragen, man kann Beträge zu- oder abbuchen etc.).

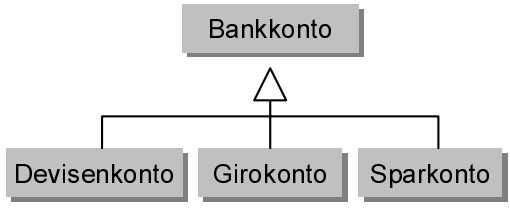

Abbildung 9.10: Beispiel Vererbung: Bankkonto

Abbildung 9.10 zeigt die Klasse *Bankkonto*, welche die Basisfunktionalität zur Verfügung stellt. Die davon abgeleiteten Klassen verändern gegebenenfalls die bestehende Funktionalität (z.B. darf beim Girokonto bei einer Abbuchung der Kontostand ein gewisses Limit nicht unterschreiten). Die abgeleiteten Klassen erweitern die Basisklasse durch zusätzliche Eigenschaften (z.B. einen Haben-Zinssatz beim Sparkonto, einen Soll-Zinssatz beim Girokonto) und bringen zusätzliche Funktionen mit ein (z.B. das Berechnen der Verzinsung am Ende eines Abrechungszeitraums). *Bankkonto* ist der Sammelbegriff für Devisen-, Giro- und Sparkonten. Würde Abbildung 9.10 die grundlegenden Strukturen eines Frameworks für Bankanwendungen zeigen, wäre die Klasse *Bankkonto* sicherlich eine abstrakte Klasse. Keine Bank führt einfach nur ein Bankkonto. Tatsächlich hat ein Kunde dort immer ein spezielles Konto, wie z.B. ein Girokonto. Demnach ist eine Instanz vom Typ Bankkonto unbrauchbar, weil die Klasse Bankkonto nur grundlegende Funktionalität zur Verfügung stellt, die für sich allein genommen wenig sinnvoll ist. Erst wenn sie zu einem konkreten Typ eines Bankkontos wie einem Girokonto spezialisiert wird, kann der Code in der Klasse Bankkonto sinnvoll eingesetzt werden.

Übertragen wir diesen Sachverhalt auf Enterprise JavaBeans und Komponenten. Ist es sinnvoll, eine Enterprise-Bean (Komponente) vom Typ Bankkonto mit dem Ziel zu modellieren, eine Basis für spezialisierte Ableitungen zu haben? Eine Komponente also, die einen Sammelbegriff modelliert? Eine solche Komponente müsste konsequenterweise eine abstrakte Komponente sein, da es keinen Sinn ergibt, sie zur Laufzeit zu benutzen. Wie sieht der Deployment-Deskriptor einer abstrakten Komponente aus? Erschwerend kommt die Tatsache hinzu, dass der Begriff der Komponentenvererbung nicht eindeutig definiert ist. Außerdem herrscht keine Einigkeit darüber, wie die Komponentenvererbung aussehen soll.

Da die Spezifikation der Enterprise JavaBeans Komponentenvererbung ausdrücklich nicht unterstützt, kann es keine sinnvolle Vererbung auf Enterprise-Bean-Ebene geben. Der Grund dafür liegt in den *create*- und *find*-Methoden des Home-Interface einer Enterprise-Bean. Gäbe es eine (abstrakte) Enterprise-Bean Bankkonto, könnte man über deren *create*-Methode keine Sparkonto-Bean erzeugen. Das Bankkonto kann in diesem Fall nicht als Factory zum Erzeugen von abgeleiteten Klassen dienen. Welchen Sinn hat aber eine *create*-Methode einer abstrakten Komponente, wenn nicht zum Erzeugen von abgeleiteten Komponenten?

Das Problem mit den *find*-Methoden ist ähnlich gelagert. Nehmen wir an, das Home-Interface der Bankkonto-Bean weist eine *find*-Methode auf, die alle Konten zurückliefert, deren Bestand negativ ist. Ruft man diese Methode auf, kann man sicher sein, dass sich die Ergebnismenge ausschliesslich aus Giro- und Devisenkonten zusammensetzt. Sparkonten können keinen negativen Bestand haben und die Bank führt keine Konten vom Typ Bankkonto. Für die Ergebnismenge dieser *find*-Methode soll eine Mahnliste erstellt werden. Bei den Girokonten sollen zusätzlich Dispositionszinsen berechnet werden. Der EJB-Container, der für die Instanziierung der Enterprise-Beans und deren Remote-Interface-Implementierungen zuständig ist, würde konsequenterweise Instanzen vom Typ Bankkonto erzeugen, da die *find*-Methode am Home-Interface der Bankkonto-Bean aufgerufen wurde. Er würde keine Instanzen vom Typ Giro- oder Devisenkonto erzeugen, da er von der Vererbungsbeziehung nichts weiß. Es gibt auch keinen Weg, ihn davon in Kenntnis zu setzten. Eine Methode an einem Element der Ergebnismenge aufzurufen, wäre demnach gefährlich, da die spezialisierte Funktionalität der abgeleiteten Enterprise-Bean-Typen nicht verfügbar ist. Bestimmte Methoden wie die Berechnung der Dispositionszinsen sind überhaupt nicht verfügbar. Ein Type-Casting würde zu einer Ausnahme führen, da der EJB-Container tatsächlich nur Bankkonto-Instanzen erzeugt hat. Polymorphie wäre in diesem Fall nur möglich, wenn der EJB-Container von der Vererbungsbeziehung wüsste. Er würde dann nämlich die tatsächlichen Typen von Enterprise-Beans und deren Remote-Interface-Implementierungen erzeugen, die dann dem Client gegenüber in der Ergebnismenge einheitlich als Bankkonto auftreten. Der Client wäre dann auch in der Lage, ein Type-Casting durchzuführen bzw. mit Hilfe des *instanceof*-Operators den tatsächlichen Typ des Kontos festzustellen.

Eine Enterprise-Bean kann in einem gewissen Rahmen polymorph sein. Ein Interface *Bankkonto* könnte als Basis für die Remote-Interfaces der speziellen Enterprise-Bean-Typen dienen. Damit könnten Spar-, Giro- oder Devisenkonten einheitlich als Bankkonten behandelt werden. Zu Zwecken der Wiederverwendung könnte eine abstrakte Klasse Bankkonto modelliert werden, von der die Enterprise-Bean-Klassen der anderen Kontotypen erben. Diese Vorgehensweise wird von der Spezifikation ausdrücklich gebilligt. Eine Enterprise-Bean *Bankkonto* zu modellieren, von der die spezialisierten Enterprise-Beans erben, wäre nur dann sinnvoll, wenn über deren *find*- und *create*-Methoden die tatsächlichen Konten polymorph benutzt werden könnten. Dazu fehlt (wie bereits erwähnt) von Seiten der Spezifikation noch jegliche Unterstützung.

Eine Alternative zur Vererbung bei Komponenten ist die Entwicklung von konfigurierbaren Komponenten. In dem oben diskutierten Beispiel könnte eine (konkrete) Enterprise-Bean *Bankkonto* entwickelt werden, die sich je nach Konfiguration wie ein Spar-, Giro- oder Devisenkonto verhält. Durch die Entwicklung konfigurierbarer Komponenten kann eben auch eine gewisse Art von Polymorphie erreicht werden. Die Polymorphie bezieht sich dabei jedoch nicht auf den Typ der Komponente, sondern auf die Konfigurationswerte der jeweiligen Instanz. Um eine solche Komponente zu realisieren, kann ein Framework eingesetzt werden, das auf Klassenvererbung aufgebaut ist. Abbildung 9.11 stellt eine solche Architektur schematisch dar.

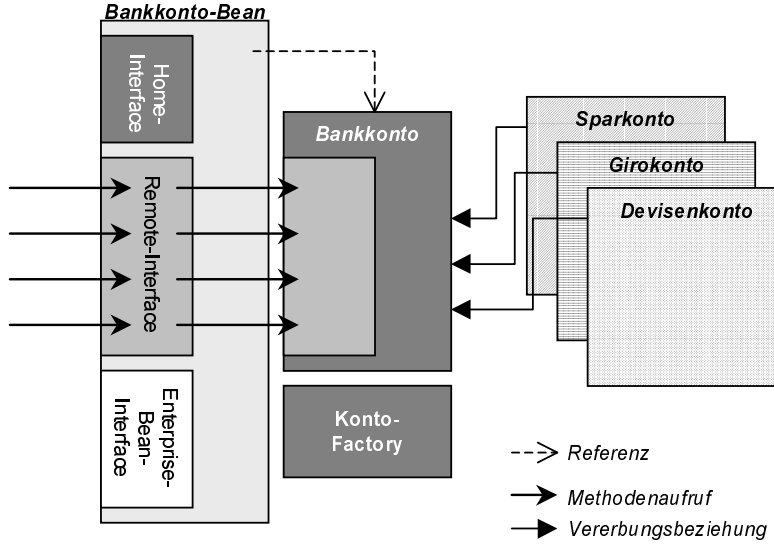

Abbildung 9.11: Architektur einer konfigurierbaren Komponente

Die Klasse *Bankkonto* ist eine abstrakte Basisklasse, die die Methoden des Remote-Interfaces der Enterprise-Bean Bankkonto zur Verfügung stellt. Davon abgeleitet sind die konkreten Klassen *Sparkonto*, *Girokonto* und *Devisenkonto*. Wenn die Bankkonto-Bean in einer der Methoden des Enterprise-Bean-Interfaces initialisiert wird, liest sie z.B. ihre Konfiguration aus dem Bean-Environment. Sie gibt ihre Konfiguration an eine Instanz der Klasse *KontoFactory* weiter, die eine für die Konfiguration passende Bankkonto-Implementierung auswählt (Spar-, Giro- oder Devisenkonto). Die *KontoFactory*-Klasse erzeugt eine Instanz und initialisiert diese entsprechend der Konfiguration. Dieser Mechanismus könnte auch durch die persistenten Daten des jeweiligen Kontos mit den notwendigen Informationen versorgt werden statt mit den Konfigurationsdaten aus dem Environment. Die Enterprise-Bean Bankkonto hält eine Referenz auf die durch die Konto-Factory erzeugte Instanz des jeweiligen Kontos. Alle Aufrufe der Methoden des Remote-Interface werden an diese Instanz delegiert.

Durch eine derartige Architektur bleibt die Komponente klar strukturiert und leicht erweiterbar. Voraussetzung ist, dass eine sinnvolle Menge an Methoden des Remote-Interface für die verschiedenen Typen definiert werden kann. Durch diese Vorgehensweise kann die Komponentenvererbung elegant umgangen werden. Die Komponente wahrt ihren Blackbox-Charakter, während sie intern ein Framework (Whitebox) für die notwendige Flexibilität nutzt. Die *create*- und *find*-Methoden des Home-Interface sind in vollem Umfang nutzbar. Polymorphie entsteht nicht durch den Typ der Komponente, sondern durch die an eine bestimmte Instanz gebundenen Werte. Entsprechend dieser instanzbezogenen Werte bestimmt sich auch das Verhalten der Komponente. Der Client könnte den tatsächlichen Typ eines Bankkontos statt mit dem *instanceof*-Operator z. B. durch eine Methode *getAccountType* des Remote-Interfaces herausfinden. Bestimmte Methoden des Remote-Interfaces könnten außerdem nur für bestimmte Kontotypen aufrufbar sein. Die jeweiligen Methoden könnten dies z. B. durch das Werfen einer Ausnahme vom Typ *IllegalOperationException* dokumentieren. Aus der Dokumentation der Komponenten müsste jedoch eindeutig hervorgehen, für welche Kontotypen diese Methoden aufgerufen werden dürfen.

Konfigurierbare Komponenten, die intern Klassen- und Interface-Vererbung in Zusammenhang mit einem Framework für die notwendige Flexibilität verwenden, sind eine ernstzunehmende Alternative zur Komponentenvererbung. Eine vollständige Unterstützung für Vererbung im Komponentenmodell der Enterprise JavaBeans dürfte die Komplexität wesentlich erhöhen. Der Container müsste von der Vererbungsbeziehung in Kenntnis gesetzt werden, indem die Vererbungsstrukturen deklarativ im Deployment-Deskriptor beschrieben werden. Bei Entity-Beans bringt die Persistenz noch zusätzliche Komplexität mit sich. Komponentenvererbung würde sich auch auf den Persistence-Manager auswirken. Auch er müsste von der Vererbungsbeziehung wissen, um die betroffenen Tabellen entsprechend in Relation setzen zu können. Außerdem müsste er die Formulierung von Suchanfragen für polymorphe Komponenten ermöglichen. Der zusätzliche Laufzeit-Overhead beim Einsatz von *finder*-Methoden im EJB-Container dürfte die Performanz beeinträchtigen. Bei jedem Element der Ergebnismenge müsste der tatsächliche Typ festgestellt werden. Es bleibt daher die Frage, ob sich eine derartige zusätzliche Komplexität durch den gewonnenen Nutzen rechtfertigt.

Abschließend möchten wir noch auf die Ausführungen zur Verwendung von Vererbung im Komponentenumfeld in [Szyperski, 1998] und [Griffel, 1998] hinweisen sowie auf die Ausführungen zur Bean-Vererbung in [IBM, 1999].

9.5 Enterprise JavaBeans und Events

Events sind eingetretene Ereignisse, die als Objekte bestimmter Ereignisklassen erfasst werden (im weiteren Verlauf werden die Begriffe Ereignis und Event synonym verwendet). Events können benutzt werden, um über das Eintreten bestimmter Zustände

oder das Ausführen bestimmter Aktionen zu informieren. Das Java-AWT (Abstract Windowing Toolkit) z.B. erzeugt Objekte vom Typ *MouseEvent*, wenn ein Benutzer mit der Maus auf der Oberfläche Aktionen durchführt (z.B. Maustaste klicken, Maus bewegen). Beim Modell der JavaBeans wird von einer Bean ein *PropertyChangeEvent*-Objekt erzeugt, wenn ein Attribut der Bean verändert wird. Diese Event-Objekte werden an andere Objekte weitergegeben, die an derartigen Ereignissen interessiert sind. Diese Art von Ereignisbehandlung nennt man *Delegation-Event-Model*. Eine Ereignisquelle bietet Interessenten (Listenern) die Möglichkeit, sich für bestimmte Ereignisse zu registrieren. Tritt dann ein solches Ereignis ein, wird von der Ereignisquelle ein Event-Objekt erzeugt und an alle registrierten Interessenten weitergegeben. Die Ereignisquelle behandelt also das Ereignis nicht selbst, sondern delegiert die Behandlung dieses Events an die Listener-Objekte.

Wie in Abschnitt 9.3 bereits erwähnt wurde, wäre ein Event-Modell für die dynamische Kopplung von Enterprise-Bean-Komponenten wünschenswert. Was bei der Entwicklung von EJB-basierten Anwendungen häufig auch vermisst wird, ist die Möglichkeit, Ereignisse vom Server aus an den Client zu übermitteln. Beispielsweise möchte eine Buchhaltungsanwendung, die parallel auf mehreren Clientrechnern benutzt wird, über das Anlegen oder Löschen von Konten informiert werden, um den Benutzer darauf hinzuweisen oder um ihre Ansichten zu aktualisieren. Für diese Fälle braucht man einen besonderen Eventmechanismus, nämlich einen verteilten. Da die Enterprise-Beans in einem anderen Prozess agieren als die Objekte der Clientanwendung, muss das Ereignis (das Event-Objekt) über Prozessgrenzen hinweg transportiert werden.

Die Version 2.0 der EJB-Spezifikation sieht weder einen lokalen Eventmechanismus für die Kopplung von Enterprise-Beans noch einen verteilten Eventmechanismus für die Kommunikation mit anderen Prozessen vor. Was sie aber vorsieht, ist die Einbindung des Java Message Service.

Wir werden in diesem Abschnitt ein Konzept für verteilte Events vorstellen, bei dem sich Objekte eines Remote-Clients als Interessenten (Listener) anmelden und über das Enterprise-Beans Event-Objekte verschicken können, wann immer ein bestimmtes Ereignis eintritt. Da der Java Message Service auf die asynchrone Kommunikation zwischen Prozessen über Netzwerk spezialisiert ist, bietet er die ideale Basis für eine solche Implementierung. Ein vergleichbarer Mechanismus kann auch Message-Driven-Beans als Event-Listener unterstützen. Session- und Entity-Beans können nicht als Event-Listener verwendet werden.

Wir wollen bei der Umsetzung von folgenden Voraussetzungen ausgehen :

- Die Lösung soll vor allem Session- und Entity-Beans das Verschicken von Events über Prozessgrenzen hinweg ermöglichen. Die Beans unterstützen demnach den Remote-Client-View.

- Das Ereignis soll in Form eines leichtgewichtigen Java-Objekts weitergegeben werden.
- Eine Enterprise-Bean darf nicht in der Lage sein, Events zu empfangen.
- Die Qualität der Nachrichtenübermittlung bei diesem Event-Mechanismus soll niedrig sein, d.h., der Absender eines Events bekommt keine Garantie und keine Bestätigung dafür, dass das Ereignis-Objekt bei den Empfängern angekommen ist.
- Eine gesendete Nachricht wird an alle registrierten Interessenten geschickt, die Reihenfolge der Ereignis-Zustellung ist rein zufällig.
- Der Zustand des Event-Dienstes (die registrierten Listener) soll nach Möglichkeit nicht in der Enterprise-Bean selbst gespeichert werden, da es dadurch zu Konflikten mit dem Lebenszyklusmanagent des EJB-Containers kommen kann.

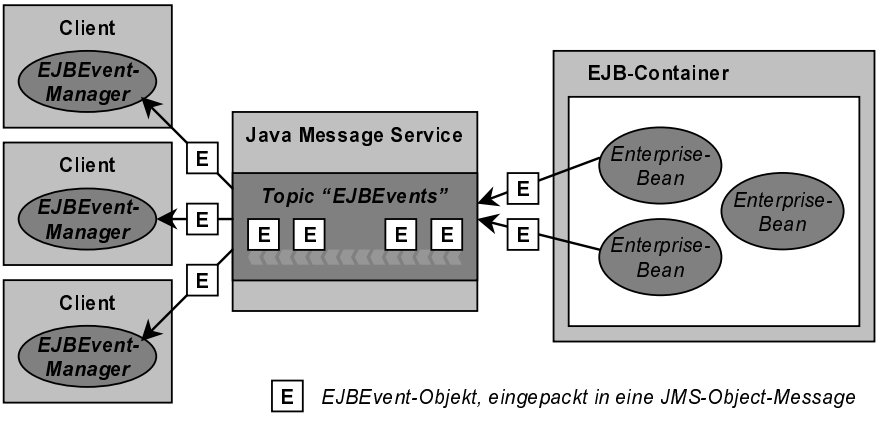

Abbildung 9.12: Verteilter Event-Service via Java Message Service

Abbildung 9.12 stellt die Implementierung schematisch dar. Jedem Clientprozess steht ein *EJBEventManager*-Objekt zur Verfügung. Der Event-Manager bietet über eine bestimmte Schnittstelle (siehe unten) Interessenten die Möglichkeit, sich für Enterprise-Bean-Events zu registrieren. Der Event-Manager wiederum ist als Empfänger des Topics *EJBEvents* beim Java Message Service registriert (vgl. Abschnitt 6.2.6). Möchte eine Enterprise-Bean einen Event auslösen, um den oder die Clients über ein bestimmtes Ereignis zu informieren, verpackt sie ein *EJBEvent*-Objekt in einem Objekt vom Typ *javax.jms.ObjectMessage* (vgl. Abschnitt 6.2.3) und sendet es an den Topic *EJBEvents* (vgl. Abschnitt 6.2.5). Die Enterprise-Bean agiert dabei wie ein normaler (sendender) JMS-Client. Der JMS-Provider sorgt dafür, dass die Nachricht an alle für diesen Topic registrierten Empfänger zugestellt wird. Der Event-Manager empfängt die Nachricht, entnimmt der Nachricht das *EJBEvent*-Objekt und verteilt es an alle bei ihm registrierten Interessenten (siehe Abbildung 9.13).

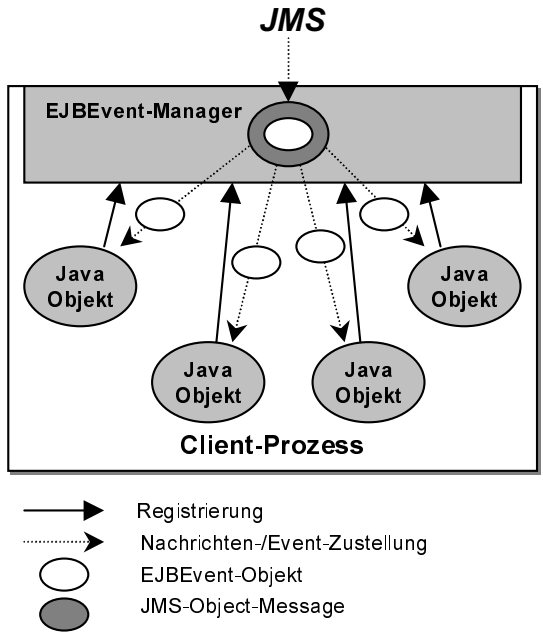

Abbildung 9.13: Prozessinterne Kommunikation mit dem Event-Manager

Aus dem soeben beschriebenen Sachverhalt lassen sich verschiedene Teilnehmer und Implementierungen für den verteilten Event-Service zur Benachrichtigung von Clients ableiten, die wir im Folgenden näher besprechen.

Der Event

```
package ejb.event;

import javax.ejb.Handle;

public class EJBEvent implements java.io.Serializable {

    private int    eventType;
    private Handle eventSource;

    public EJBEvent(int type) {
        this(type, null);
    }

    public EJBEvent(int type, Handle source) {
        eventSource = source;
        eventType   = type;
    }
```

```
    public Handle getEventSource() {
        return eventSource;
    }

    public int getEventType() {
        return eventType;
    }

}
```

Listing 9.9: EJBEvent-Klasse

Die Klasse *EJBEvent* repräsentiert eingetretene Ereignisse. Über das Attribut *eventType* lässt sich das Ereignis identifizieren. Das Attribut *eventSource* erlaubt es, den Auslöser des Events zu identifizieren. Dieses Attribut ist vom Typ *javax.ejb.Handle*. Über das Handle ist der Client in der Lage, die Enterprise-Bean, die den Event ausgelöst hat, anzusprechen. Einzig Message-Driven-Beans besitzen kein Handle, da sie nicht direkt von einem Client angesprochen werden können. Sollen Message-Driven-Beans ebenfalls EJBEvents auslösen können, müssen die Clients darauf vorbereitet sein, dass das Attribut *eventSource* den Wert *null* annehmen kann. Eine Enterprise-Bean könnte auch eine von *EJBEvent* abgeleitete Klasse benutzen, um dem Client ein Ereignis zu senden. Über abgeleitete Klassen lässt sich der Informationsgehalt eines Ereignisses erhöhen. So könnte z.B. eine Event-Klasse *NewAccountEvent* (abgeleitet von *EJBEvent*) über das Anlegen eines neuen Kontos informieren. Zusätzlich könnte die *NewAccountEvent*-Klasse ein Attribut *accountNumber* besitzen, das dem Client die Kontonummer des soeben angelegten Kontos mitteilt.

Der Interessent für Ereignisse (Listener)

```
package ejb.event;

public interface EJBEventListener {

    public void notify(EJBEvent event);

}
```

Listing 9.10: Interface EJBEventListener

Die Klasse eines Objekts, das sich für Enterprise-Bean-Events interessiert, muss dieses Interface implementiert haben, um sich beim Event-Manager als Listener registrieren zu können. Tritt ein Ereignis ein, wird auf einem registrierten Objekt die Methode *notify()* aufgerufen und das Ereignisobjekt als Parameter übergeben.

Der Auslöser von Ereignissen

Ausgelöst werden die Ereignisse von Enterprise-Beans. Sie lösen ein Ereignis aus, indem sie ein Objekt vom Typ *EJBEvent* oder von einem abgeleiteten Typ erzeugen. Die Enterprise-Bean übergibt ihr Handle (oder *null*, falls es sich um eine Message-Driven-Bean handelt) an das Event-Objekt. Je nach verwendeter Event-Klasse übergibt die Enterprise-Bean weitere Informationen an das Event-Objekt. Die Event-Klassen sowie die entsprechenden Werte eines Event-Objekts können von einer Anwendung beliebig festgelegt werden. Falls die Anwendung nur die Basisklasse *EJBEvent* benutzt, sollte eine zentrale Klasse oder ein zentrales Interface benutzt werden, in dem die Werte für die unterschiedlichen Event-Typen in Form von öffentlichen Konstanten definiert werden.

Um der Enterprise-Bean das Auslösen von *EJBEvents* zu erleichtern, könnte man eine Hilfsklasse zur Verfügung stellen, wie sie in Listing 9.11 gezeigt wird.

```
package ejb.event;

import javax.jms.JMSException;
import javax.jms.ObjectMessage;
import javax.jms.Session;
import javax.jms.Topic;
import javax.jms.TopicConnection;
import javax.jms.TopicConnectionFactory;
import javax.jms.TopicPublisher;
import javax.jms.TopicSession;
import javax.naming.NamingException;
import ejb.util.Lookup;

public class EJBEventHelper {

    public static final String FACTORY    =
        "java:/env/jms/DefaultConnectionFactory";
    public static final String TOPIC_NAME =
        "java:/env/jms/EJBEvents";

    private static TopicConnectionFactory topicCF  = null;
    private static Topic                  ejbEvent = null;

    public EJBEventHelper()
        throws NamingException
    {
        topicCF  = (TopicConnectionFactory)Lookup.get(FACTORY);
        ejbEvent = (Topic)Lookup.get(TOPIC_NAME);
    }

    public void fireEvent(EJBEvent event)
        throws JMSException
    {
```

```
        if(event == null) {
            throw new
              IllegalArgumentException("event must not be null!");
        }
        TopicConnection tc   = null;
        TopicSession    ts   = null;
        TopicPublisher  tpub = null;
        try {
            tc = topicCF.createTopicConnection();
            ts = tc.createTopicSession(false,
                                       Session.AUTO_ACKNOWLEDGE);
            tpub = ts.createPublisher(ejbEvent);
            ObjectMessage om = ts.createObjectMessage();
            om.setObject(event);
            tpub.publish(om);
        } finally {
            try  { tpub.close(); } catch(Exception ex) {}
            try  { ts.close(); }    catch(Exception ex) {}
            try  { tc.close(); }    catch(Exception ex) {}
        }
    }

}
```

Listing 9.11: Klasse EJBEventHelper

Eine Enterprise-Bean *BankAccount* könnte die Clients mit folgendem Codefragment über das Anlegen eines neuen Bankkontos informieren:

```
Handle h = theContext.getEJBObject().getHandle();
EJBEvent event = new EJBEvent(EventTypes.NEW_ACCOUNT, h);
EJBEventHelper helper = new EJBEventHelper();
helper.fireEvent(event);
```

Da der Java Message Service asynchron arbeitet, wird die Enterprise-Bean durch das Auslösen eines Ereignisses nicht unnötig lange blockiert. Während die Enterprise-Bean ihre Verarbeitung fortsetzt, kümmert sich der JMS-Provider um die Zustellung der Nachricht und der Event-Manager um die Verteilung des Ereignisses an die Interessenten.

Der Event-Manager

```
package ejb.event;

import java.util.List;
import javax.ejb.EnterpriseBean;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.ObjectMessage;
```

```
import javax.jms.Session;
import javax.jms.Topic;
import javax.jms.TopicConnection;
import javax.jms.TopicConnectionFactory;
import javax.jms.TopicSubscriber;
import javax.jms.TopicSession;
import javax.naming.NamingException;
import ejb.util.Lookup;

public class EJBEventManager implements MessageListener {

    public static final String FACTORY =
        "java:/env/jms/DefaultConnectionFactory";
    public static final String TOPIC_NAME =
        "java:/env/jms/EJBEvents";

    private static EJBEventManager        theInstance = null;
    private static TopicConnectionFactory tcFactory   = null;
    private static Topic                  ejbEvents   = null;

    private TopicConnection tConnection = null;
    private TopicSession    tSession    = null;
    private TopicSubscriber tSub        = null;
    private List            tListeners  = null;

    static {
        try {
            tcFactory = (TopicConnectionFactory)
                              Lookup.get(FACTORY);
            ejbEvents = (Topic)Lookup.get(TOPIC_NAME);
            theInstance = new EJBEventManager();
        } catch(NamingException nex) {
            nex.printStackTrace();
            throw new IllegalStateException(nex.getMessage());
        }
    }

    private EJBEventManager() {
        tListeners = new java.util.ArrayList();
    }

    public synchronized void
           addEJBEventListener(EJBEventListener listener)
    {
        if(listener instanceof EnterpriseBean) {
            throw new
                IllegalArgumentException("beans are not allowed!");
        }
        if(tListeners.isEmpty()) {
            connect();
        }
```

```
        if(!tListeners.contains(listener)) {
            tListeners.add(listener);
        }
    }

    public synchronized void
              removeEJBEventListener(EJBEventListener listener)
    {
        tListeners.remove(listener);
        if(tListeners.isEmpty()) {
            disconnect();
        }
    }

    public static EJBEventManager getInstance() {
        return theInstance;
    }

    private void connect() {
        try {
            tConnection = tcFactory.createTopicConnection();
            tSession    =
                tConnection.createTopicSession(
                    false,
                    Session.AUTO_ACKNOWLEDGE);
            tSub = tSession.createSubscriber(ejbEvents);
            tSub.setMessageListener(this);
            tConnection.start();
        } catch(JMSException jmsex) {
            jmsex.printStackTrace();
            throw new IllegalStateException(jmsex.getMessage());
        }
    }

    private void disconnect() {
        try {
            tConnection.stop();
            tSub.close();
            tSession.close();
            tConnection.close();
        } catch(JMSException jmsex) {
            jmsex.printStackTrace();
            throw new IllegalStateException(jmsex.getMessage());
        }
    }

    public void onMessage(Message msg) {
        EJBEvent event = null;
        try {
            event = (EJBEvent)((ObjectMessage)msg).getObject();
        } catch(ClassCastException ccex) {
```

```
                ccex.printStackTrace();
                System.err.println("expected ObjectMessage!");
                return;
            } catch(JMSException jmsex) {
                jmsex.printStackTrace();
                return;
            }
            EJBEventListener l = null;
            if(event == null) {
                return;
            }
            for(int i = 0; i < tListeners.size(); i++) {
                l = (EJBEventListener)tListeners.get(i);
                l.notify(event);
            }
        }

    }
```

Listing 9.12: Klasse EJBEventManager

Die Klasse *EJBEventManager* ist nach dem Singleton-Pattern (vgl. [Gamma et al., 1996]) implementiert. Das Singleton-Pattern stellt sicher, dass es von dieser Klasse nur eine Instanz in einem Java-Prozess geben kann. Im Fall des Event-Managers ist dadurch sichergestellt, dass sparsam mit den JMS-Ressourcen umgegangen wird. Da der Konstruktor mit dem Attribut *private* versehen ist, kann nur die Klasse selbst Instanzen erzeugen. Die Instanz des Event-Managers wird im statischen Initialisierer der Klasse erzeugt. Dieser wird einmalig ausgeführt, nachdem die Klasse geladen wurde. Dort holt sich der Event-Manager auch eine Referenz auf die Topic-Connection-Factory sowie auf den Topic, über den die Ereignisse verteilt werden.

Die Clients erhalten über die statische Methode *getInstance* Zugriff auf die Instanz des Event-Managers. Sie haben die Möglichkeit, sich über die Methode *addEJBEventListener* als Interessenten für Events zu registrieren. Diese Methode stellt zudem sicher, dass sich keine Enterprise-Beans als Interessenten für Events registrieren können. Würde sich eine Enterprise-Bean als Interessent registrieren können, würde der Event-Manager beim Eingang einer Nachricht eine Methode an der Enterprise-Bean-Instanz aufrufen, ohne dass der EJB-Container dies kontrollieren kann. Dieser Sachverhalt würde einen schweren Verstoß gegen die Spezifikation darstellen. Über die Methode *removeEJBEventListener* können sich Interessenten für Ereignisse deregistrieren. Der Event-Manager registriert sich erst dann beim Topic als Empfänger, wenn sich der erste Interessent registriert hat. Wenn sich der letzte Interessent deregistriert hat, schließt der Event-Manager die Verbindung zum Topic automatisch. Dadurch werden vom JMS-Provider nicht unnötig Nachrichten zugestellt. Wie in Abschnitt 6.2. bereits erwähnt, werden Nachrichten, die über einen Topic verteilt werden, nur an diejenigen Empfänger zugestellt, die zu diesem Zeitpunkt gerade als Empfänger registriert sind.

Löst eine Enterprise-Bean ein Ereignis aus, wird der Event-Manager-Instanz vom JMS-Provider über den Aufruf der Methode *onMessage* eine Nachricht zugestellt. Der Event-Manager entnimmt der Nachricht das Event-Objekt und verteilt es an die bei ihm registrierten Interessenten durch den Aufruf der Methode *notify* am Interface *EJBEventListener*.

Der Client

Listing 9.13 zeigt, wie ein Client den Event-Manager benutzen würde, um Ereignisse, die von Enterprise-Beans ausgelöst werden, zu empfangen.

```
package ejb.event;

public class Client implements EJBEventListener {

    …

    public Client() {
        …
    }

    public void init() {
        …
        EJBEventManager em = EJBEventManager.getInstance();
        em.addEJBEventListener(this);
        …
    }

    public void shutdown() {
        …
        EJBEventManager em = EJBEventManager.getInstance();
        em.removeEJBEventListener(this);
        …
    }

    public void notify(EJBEvent event) {
        switch(event.getEventType()) {
            case EventTypes.NEW_ACCOUNT: …
            case EventTypes.REMOVE_ACCOUNT: …
        }
    }

    …

}
```

Listing 9.13: Klasse EJBEventManager-Client

Der Client registriert sich zum Initialisierungszeitpunkt (im Beispiel in der Methode *init*) als Interessent für Ereignisse. Löst eine Enterprise-Bean ein Ereignis aus, wird es über den JMS-Provider an den Event-Manager zugestellt, der es wiederum an die registrierten Interessenten über den Aufruf der Methode *notify* weitergibt. In der Methode *notify* wertet der Client das Event-Objekt aus und löst entsprechende Aktionen aus. Bei der Beendigung des Clients (im Beispiel in der Methode *shutdown*) deregistriert er sich als Interessent für Ereignisse.

Um den Informationsgehalt eines Ereignisses zu erhöhen, könnten (wie bereits erwähnt) Subklassen der Klasse *EJBEvent* gebildet werden. Ferner könnte die Registrierung kategorisiert werden. Das heißt, Interessenten registrieren sich beim Event-Manager nur für bestimmte Event-Typen. Dazu müssten die Methoden *addEJBEventListener* und *removeEJBEventListener* um das Attribut Event-Klasse erweitert werden. Dann würde ein Interessent nicht mehr alle Ereignisse zugestellt bekommen, sondern nur noch die Ereignisse, für deren Typ er sich wirklich interessiert.

9.6 Internet-Anbindung

Dieser Abschnitt behandelt die Besonderheiten des web-basierten EJB-Clients. Wir werden zwei konzeptuell stark unterschiedliche Ansätze betrachten. Da es sich lediglich um eine spezielle Form der Clientsicht handelt, die bereits ausgiebig dargestellt wurde, werden wir nicht weiter auf die Implementierungsdetails eingehen. Uns erscheint es jedoch im Rahmen dieses Buches wichtig, die Notwendigkeit eines Webclients und die damit verbundenen Konsequenzen zu diskutieren.

Wurden ursprünglich einfache Dienste über Internet angeboten (E-Mail, FTP, HTML-Publishing), wickelt man heute über dieses Medium komplexe Transaktionen ab (z. B. beim Internet-Banking). Sicherheit in diversen Ausprägungen (Zugriffssicherheit, Datenkonsistenz etc.) spielt dabei eine maßgebliche Rolle. Der Entwickler von web-basierten Anwendungen wird mit einer Vielzahl von unterschiedlichen Techniken konfrontiert.

Als Basis für die Kommunikation steht nach wie vor das (im Vergleich zu anderen Protokollen primitive) HTTP-Protokoll (HyperText Transfer Protocol) zur Verfügung, über das die Kommunikation eines Browsers mit einem HTTP-Server abgewickelt wird. Es ist zustandslos und unterstützt nur eine sehr begrenzte Anzahl von Operationen. Ein (Standard-) HTTP-Server hat keine Persistenzschnittstelle, kein Anwendungsframework und keinen Transaktionsmonitor – zurecht, denn HTTP ist nicht als Kommunikationsbasis transaktionsorientierter, web-basierter Anwendungssysteme entwickelt worden, sondern für den Transfer von HyperText-Dateien (sprich Dateien, die HTML-Seiten beinhalten).

Die Techniken für die Implementierung von Anwendungen reichen serverseitig von C-Programmen, Perl-, Python- und Unix-Shell-Skripten (die über das CGI-Interface eines HTTP-Servers angesprochen werden) bis hin zu proprietären HTTP-Server-Erweiterungen (wie z.B. der NSAPI für den Netscape-HTTP-Server oder der ISAPI für den Internet-Information-Server von Microsoft). Dazu kommen Servlets (CGI für Java) und Techniken für die serverseitige, dynamische Generierung von HTML-Seiten (Active-Server-Pages von Microsoft und Java-Server-Pages von SUN). Viele Anbieter von Datenbanken stellen für ihre Produkte HTTP-Schnittstellen zur Verfügung, womit der Datenbankserver direkt als Webserver dienen kann und ebenfalls über proprietäre APIs aus den Datenbeständen dynamisch Webseiten für die Darstellung und Modifikation der entsprechenden Daten generiert werden können. Auf der Clientseite können reines HTML, dynamisches HTML (Javascript), Java-Applets sowie proprietäre Browserschnittstellen und Bibliotheken genutzt werden.

Jede der oben genannten Technologien hat in bestimmten Anwendungsbereichen ihre Daseinsberechtigung. Jedoch liegt genau in dieser Spezialisierung ein maßgebliches Problem. In den Pioniertagen des Internets (die noch nicht ganz vorbei sind) waren viele Webanwendungen ursprünglich als kleine Insellösungen geplant, entwickeln sich aber schnell zu unternehmensweit eingesetzen Anwendungen. Durch diesen Wechsel entstehen stark veränderte Anforderungen an die Anwendung und die gewählte Technologie hat nicht immer die erforderliche Flexibilität. Heute haben viele Webanwendungen eine entscheidende Bedeutung für den Unternehmenserfolg. Die Architektur dieser Anwendungen sollte ihrer Bedeutung gerecht werden.

Als Plattform bietet sich ein Applikationsserver an, der verschiedene Services (Persistenz, Benutzerverwaltung, Transaktionen etc.), verschiedene Kommunikationsprotokolle (HTTP, HTTPS, RMI, IIOP etc.) und einen EJB-Container zur Verfügung stellt. Er bietet all die Fähigkeiten, die heute von einer zuverlässigen Anwendungsplattform erwartet werden, die einem (Standard-) HTTP-Server jedoch fehlen. Zudem wird das Technologie-Portfolio eingeschränkt, was sich positiv auf Wartung und Wiederverwendung auswirkt. Gerade im Internetumfeld sind die Sicherheitsmechanismen der Programmiersprache Java (vgl. dazu auch Kapitel 2) von Vorteil. In den Kapiteln 2 und 3 wurde bereits besprochen, dass Sun Mircrosystems dem durch die Java-2-Plattform, Enterprise Edition, Rechnung trägt.

Häufig werden native Clients eines Anwendungssystems durch Webclients ergänzt (evtl. auch ersetzt), um Anwendungen für das Internet verfügbar zu machen (z.B. um unterwegs über das Internet Firmenanwendungen wie Reisekostenabrechnungen, Arbeitszeiterfassung, Informationen für Angebotserstellung etc. nutzen zu können). Ebenso werden Anwendungen gezielt für die Nutzung über das Internet entwickelt (z.B. E-Commerce-Applikationen). Denkbar ist auch, Anwendungen auf Basis von Webtechniken im Intranet zur Verfügung zu stellen, z.B. für Personal ohne festen oder mit häufig wechselndem Arbeitsplatz. Außerdem können dadurch die Kosten für

Installation und Updates der Clientsoftware gesenkt werden. Oftmals werden über ein Intranet aber auch eingeschränkte, schnell und leicht zu implementierende alternative Sichten auf Anwendungen angeboten (z.B. ein einfacher Zeiterfassungsclient für Personal, das auf Stundenbasis arbeitet, oder Infoterminals für den schnellen Zugriff auf Anwesenheitslisten, Qualitätsstatistiken im Produktionsbereich, Gleitzeitsalden etc.).

Für die Implementierung eines Webclients für Enterprise-JavaBeans-basierte Anwendungen wollen wir zwei Ansätze diskutieren, die sich konzeptuell stark unterscheiden, aber beide Konzepte der Java-2-Plattform, Enterprise Edition sind. Zum einen handelt es sich dabei um einen Applet-basierten Webclient, zum anderen um einen reinen HTML-Client, der durch die Servlet-Technologie unterstützt wird.

9.6.1 Java-Applets

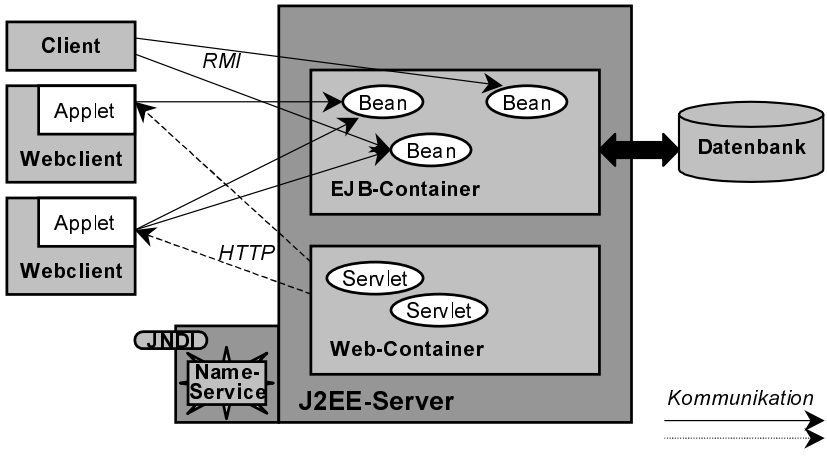

Abbildung 9.14: Internetanbindung mit Applets

Ein Client-Applet, das eingebettet in eine HTML-Seite vom Server über das HTTP-Protokoll an den Client geliefert wird, bietet dem Benutzer eine Schnittstelle, wie er sie bereits von proprietären Clients gewohnt ist (vgl. Abbildung 9.14). Das Look-and-Feel sowie die Benutzerführung eines Applet-basierten Clients entsprechen dem herkömmlicher Anwendungen. Das Applet kommuniziert, sobald es geladen ist, in der Regel direkt mit dem Applikationsserver. Die Implementierung des Applets unterscheidet sich von der eines normalen Clients nicht wesentlich. Beim Applet sollte auf die Größe des erzeugten Codes geachtet werden, um die Downloadzeiten gering zu halten. Ein Applet und ein normaler Client dürften große Teile der Implementierung gleichermaßen benutzen können. Im Extremfall kann das Applet und der native Client identisch sein (ein Applet lässt sich relativ leicht in eine Applikation verwandeln). Problematisch

bei diesem Szenario könnte die Verwendung der Java-2-Plattform sein. Die gängigen Browser unterstützen lediglich JDK 1.1, lassen sich aber durch Plug-Ins auf Java 2 aufrüsten.

9.6.2 HTML und Servlets

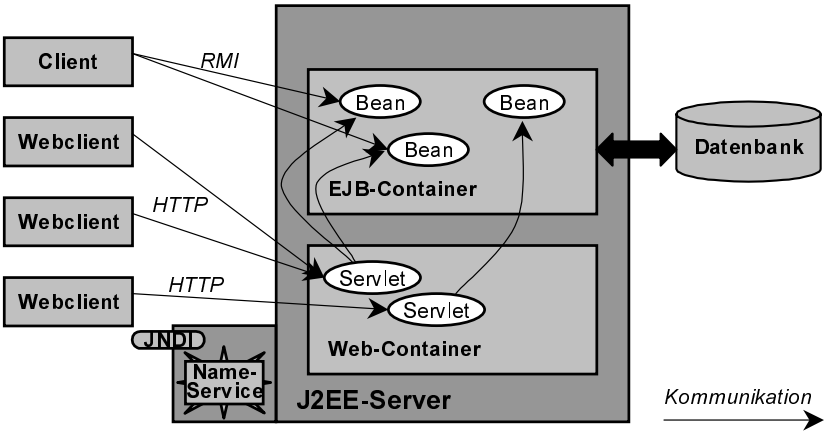

Abbildung 9.15: Internetanbindung mit Servlets

Als Alternative zu einem Applet kann ein reiner HTML-Client eingesetzt werden (vgl. Abbildung 9.15). Serverseitig dient ein Servlet (siehe [Roßbach et al., 1999]) dazu, entsprechend den Anfragen des Clients Enterprise-Bean-Aufrufe durchzuführen und dynamisch HTML-Seiten für die Darstellung der Ergebnisse bzw. die Entgegennahme von Benutzereingaben zu generieren. Dieser Ansatz hat gegenüber der Applet-Lösung den Vorteil, dass keine langen Downloadzeiten für den Applet-Code anfallen. Der verwendete Browser muss außerdem kein Java unterstützen. Damit entfällt auch das Problem, dass viele Benutzer die Java-Unterstützung des Browsers wegen Sicherheitsbedenken deaktivieren.

HTML-Clients haben nicht das typische Look-and-Feel und die Benutzerführung einer herkömmlichen Anwendung (bedingt durch den HTML-typischen Server-Roundtrip, um nach einer Benutzeraktion die Ansicht entsprechend zu aktualisieren). Durch die Verwendung von Bildern kann das Aussehen sehr individuell gestaltet werden. Ein weiterer wesentlicher Unterschied zwischen beiden Ansätzen liegt in der Verwaltung des Zustands einer Anwendung. Das Applet kann – ebenso wie der native Client – einen Zustand bis zum nächsten Aufruf einer serverseitigen Operation halten (z.B. Eingabedaten des Benutzers sammeln, Undo-Listen verwalten, Referenzen auf Enterprise-Beans speichern etc.) und kommuniziert in der Regel direkt mit dem Applikationsserver. Ein reiner HTML-Client kann keinen Zustand speichern. Er ist auf die

Unterstützung des Servlets angewiesen. Die HTML-Seite im Browser enthält nur Darstellungsinformationen und keine Logik (wir wollen von der Verwendung des clientseitigen Javascripts an dieser Stelle absehen). Die Logik wird durch das Servlet beliebig vielen Browsern zur Verfügung gestellt. Dem Servlet fällt dabei die Aufgabe zu,

- die Zustände aller Clients, die es benutzen, parallel zu verwalten,
- Anfragen der Browser entgegenzunehmen,
- die Anfrage einer Session (sprich einem bestimmten Anwendungszustand) zuzuordnen,
- der Anfrage entsprechend Enterprise-Bean-Aufrufe durchführen und
- als Antwort für den Browser eine neue HTML-Seite (die die Ergebnisse des Bean-Aufrufs darstellt) zu generieren.

Die in diesem Abschnitt dargestellten Sachverhalte treffen in gleichem Maße auf die Verwendung von Java Server Pages (JSP) zu (vgl. [Roßbach et al., 1999]). Java Server Pages bauen auf die Servlet-Architektur auf und folgen somit den gleichen Verhaltensgrundsätzen.

9.6.3 Zusammenfassung

Beide Ansätze unterscheiden sich grundlegend und stellen in gewisser Weise Gegenpole dar. Durch den Einsatz anderer Technologien und deren Kombination (z. B. client- oder serverseitiges JavaScript, proprietäre Browserschnittstellen) sind beliebige Abstufungen denkbar. Festzuhalten bleibt an dieser Stelle, dass ein bestehendes Enterprise-JavaBeans-basiertes System relativ leicht durch web-basierte Clients ergänzt werden kann. Der Implementierungsaufwand für den Webclient ist dann relativ gering, wenn bei der Implementierung des Anwendungssystems auf eine konsequente Umsetzung der Dreischichten-Architektur (Three-Tier-Architecture) geachtet wurde (vgl. Kapitel 2). In diesem Fall steht die gesamte Funktionalität und Applikationslogik in Form von Enterprise-Beans zur Verfügung. Lediglich die Viewing-Szenarien der Thin-Clients sind neu zu entwickeln. Für Anwendungen, die ausschließlich für den Internetbereich entwickelt werden (E-Commerce-Anwendungen), ist Enterprise JavaBeans ebenfalls eine interessante Plattform. Durch Enterprise JavaBeans können bei Internetanwendungen relativ leicht hohe Qualität, Sicherheit und Wartbarkeit erreicht werden. Zudem können Entwickler eingesetzt werden, die keine Spezialisten im Bereich internetorientierter Technologien sind (diese werden lediglich für die Entwicklung der Clientszenarien benötigt).

Nicht zu vergessen bleibt abschließend die Tatsache, dass sich durch die Öffnung eines Applikationsservers in Richtung Internet und Intranet die Anforderungen in puncto Sicherheit wesentlich erhöhen. Die verschlüsselte Übertragung zwischen Webclient

und dem Applikationserver, ein Firewallkonzept und diverse andere Sicherheitsmechanismen sind dabei als Ergänzung der in der Enterprise-JavaBeans-Spezifikation beschriebenen Sicherheitsanforderungen in Erwägung zu ziehen.

9.7 Entity-Beans und Details-Objekte

Die Grundlagen von Entity-Beans wurden in Kapitel 5 diskutiert. In diesem Abschnitt wird ein besonderer Aspekt bei der Verwendung von Entity-Beans behandelt.

Aus Performance-Gründen ist es sinnvoll, die Daten einer Entity-Bean über Schnittstellen zu verändern, die Massendatenoperationen zulassen. Dazu werden die Daten einer Entity-Bean in einer Klasse zusammengefasst. Sun Microsystems empfiehlt diese Vorgehensweise in einer Veröffentlichung, dem J2EE-Blueprint (vgl. [J2EE-Blueprint]). Klassen, die die Daten einer Entity-Bean zusammenfassen, werden in der Regel mit dem Zusatz »Details“ oder »Value“ bezeichnet. Als Beispiel hierzu soll die Entity-Bean *Part* dienen, welche ein Zukaufteil eines Produktionsbetriebs modelliert und den Remote-Client-View unterstützt. Ein Zukaufteil hat die Attribute Teilenummer, Teilebeschreibung, Name des Lieferanten und Preis des Zukaufteils. Listing 9.14 zeigt das Home-Interface der Enterprise-Bean *Part*.

```
package ejb.part;

import java.rmi.RemoteException;
import javax.ejb.EJBHome;
import javax.ejb.CreateException;
import javax.ejb.FinderException;

public interface PartHome extends EJBHome {

    public Part create(String partNumber)
        throws CreateException, RemoteException;

    public Part findByPrimaryKey(String partNumber)
        throws FinderException, RemoteException;

}
```

Listing 9.14: Interface PartHome

Ein Zukaufteil wird im System über seine Teilenummer identifiziert. Diese wird beim Erzeugen einer neuen Entität übergeben. Die Teilenummer dient auch als Primärschlüssel. Über die Teilenummer lassen sich bestimmte Teile wiederfinden. Listing 9.15 zeigt das Remote-Interface der *Part*-Bean.

```
package ejb.part;

import java.rmi.RemoteException;
import javax.ejb.EJBObject;

public interface Part extends EJBObject {

    public void setPartDetails(PartDetails pd)
        throws RemoteException;

    public PartDetails getPartDetails()
        throws RemoteException;

}
```

Listing 9.15: Interface Part

Das Remote-Interface weist statt einer Vielzahl von get-/set-Methoden für jedes Attribut der Entity-Bean nur genau eine get- und genau eine set-Methode auf. Als Parameter bzw. Rückgabewert dient ein Objekt vom Typ *PartDetails*, welches alle Attribute der Entity-Bean enthält. Alle Attribute der Enterprise-Bean werden gebündelt übertragen, was zu einer höheren Effizienz führt, da für die Übertragung der Attribute nur *ein* Netzwerkzugriff erforderlich ist. Listing 9.16 zeigt die Implementierung der Klasse *PartDetails*.

```
package ejb.part;

public class PartDetails implements java.io.Serializable {

    String partNumber;
    String partDescription;
    String supplierName;
    float  price;

    public PartDetails() {
    }

    public String getPartNumber() {
        return partNumber;
    }

    public void setPartDescription(String desc) {
        partDescription = desc;
    }

    public String getPartDescritption() {
        return partDescription;
    }

    public void setSupplierName(String name) {
        supplierName = name;
```

```
    }

    public String getSupplierName() {
        return supplierName;
    }

    public void setPrice(float p) {
        price = p;
    }

    public float getPrice() {
        return price;
    }

}
```

Listing 9.16: Klasse PartDetails

Dem Client werden die Attribute der Enterprise-Bean gesammelt in einem *PartDetails*-Objekt übermittelt und er ruft die entsprechenden getter-/setter-Methoden am *PartDetails*-Objekt statt an der Enterprise-Bean auf. Er kann die Attribute lokal verändern und dann gesammelt zurück an die Enterprise-Bean übertragen, um die Veränderung vorzunehmen. Üblicherweise werden in den get-/set-Methoden eines Details-Objekts bereits einfache Plausibilitätsprüfungen vorgenommen. Beispielsweise darf in manchen Fällen nicht *null* als Parameter übergeben werden oder Integerwerte dürfen bestimmte Wertebereiche nicht über- oder unterschreiten. Diese Vorgehensweise hat den Vorteil, dass auf Fehler noch vor dem Übertragen der Daten an die Enterprise-Bean hingewiesen werden kann. Oftmals ist es jedoch schwierig, die Grenze zwischen Plausibiliätsprüfungen und Business-Logik zu ziehen, d.h., es ist schwer zu entscheiden, welche Implementierungen in das Details-Objekt und welche in die Klasse der Enterprise-Bean gehört.

Listing 9.17 zeigt die Implementierung der Enterprise-Bean-Klasse.

```
package ejb.part;

import javax.ejb.*;

public abstract class PartBean implements EntityBean {

    private EntityContext theContext;
    private PartDetails   theDetails;

    /** Creates new PartBean */
    public PartBean() {}

    //Die create-Methode des Home-Interface

    public String ejbCreate(String partNumber)
```

```
    throws CreateException
{
    setPartNumber(partNumber);
    theDetails = new PartDetails();
    theDetails.partNumber = partNumber;
    return null;
}

public void ejbPostCreate(String partNumber)
    throws CreateException
{}

//Abstrakte getter-/setter-Methoden

public abstract void setPartNumber(String num);
public abstract String getPartNumber();
public abstract void setPartDescription(String desc);
public abstract String getPartDescription();
public abstract void setSupplierName(String name);
public abstract String getSupplierName();
public abstract void setPrice(float p);
public abstract float getPrice();

//Die Methode des Remote-Interface

public void setPartDetails(PartDetails pd) {
    setPartDescription(pd.getPartDescritption());
    setSupplierName(pd.getSupplierName());
    setPrice(pd.getPrice());
    theDetails = pd;
}

public PartDetails getPartDetails() {
    return theDetails;
}

//Die Methoden des javax.ejb.EntityBean-Interface

public void setEntityContext(EntityContext ctx) {
    theContext = ctx;
}

public void unsetEntityContext() {
    theContext = null;
}

public void ejbRemove()
    throws RemoveException
{}
```

```
    public void ejbActivate() {
    }

    public void ejbPassivate() {
    }

    public void ejbLoad() {
        if(theDetails == null) {
            theDetails = new PartDetails();
        }
        theDetails.partNumber = this.getPartNumber();
        theDetails.partDescription =
                              this.getPartDescription();
        theDetails.supplierName = this.getSupplierName();
        theDetails.price = this.getPrice();
    }

    public void ejbStore() {
    }
}
```

Listing 9.17: Klasse PartBean

Die Enterprise-Bean hält eine Referenz auf ein *PartDetails*-Objekt. Dieses wird zurückgegeben, wenn der Client die Attribute der Enterprise-Bean über die Methode *getPartDetails* anfordert. Will der Client die Attribute der Enterprise-Bean verändern, wird die Referenz mit dem neuen Details-Objekt überschrieben. Beim erstmaligen Erzeugen der Enterprise-Bean enthält das Details-Objekt nur die Teilenummer. Der Client versorgt die Enterprise-Bean nach und nach mit den restlichen Daten. Jedesmal wenn der Client die Attribute der Enterprise-Bean ändert, werden die Werte aus dem Details-Objekt entnommen und an die get-/set-Methoden weitergegeben, so dass der Persistence-Manager für die Speicherung der veränderten Werte sorgen kann. Wird eine existierende *Part*-Bean geladen, wird ein neues *PartDetails*-Objekt erzeugt und mit den entsprechenden Werten versehen.

Die eben beschriebene Vorgehensweise wird durch das J2EE-Blueprint vorgeschlagen. Bei dieser Vorgehensweise gibt es eine wichtige Besonderheit im Zusammenhang mit dem Remote-Client-View, die wir im weiteren Verlauf dieses Abschnitts diskutieren wollen.

Nehmen wir an, in der Datenbank existiert folgendes Zukaufteil:

Teilenummer	Beschreibung	Name Lieferant	Preis
0815	Abdeckung	A-Lieferant	1,50 Dollar

Tabelle 9.2: Ursprüngliche Daten der Part-Bean

Nehmen wir weiter an, es gibt eine Anwendung, mit der Zukaufteile verwaltet werden können. Zwei Benutzer wollen über diese Anwendung das Zukaufteil 0815 verändern. Abbildung 9.16 veranschaulicht diesen Sachverhalt.

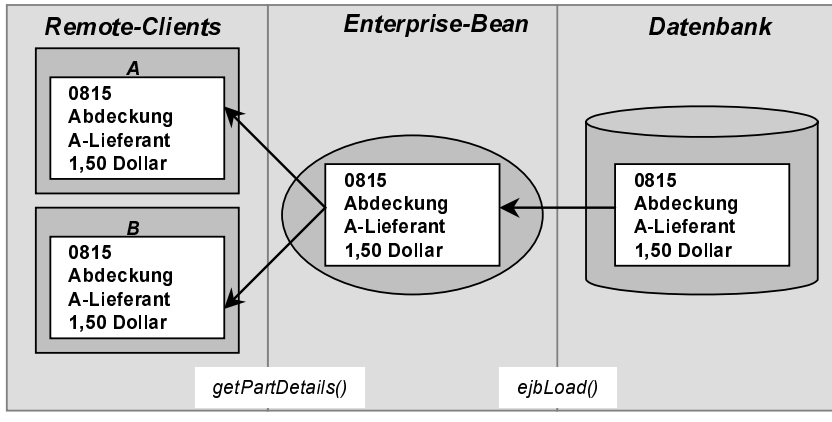

Abbildung 9.16: Verwendung eines Details-Objekts

Die Enterprise-Bean erhält eine Kopie der Daten aus der Datenbank, die Remote-Clients erhalten jeweils ihre eigene Kopie der Daten in Form eines Details-Objekts. Angenommen, Client A verändert den Lieferanten in *C-Lieferant* und speichert die Daten. Das veränderte Details-Objekt wird an die Enterprise-Bean übertragen, die Attribute der Enterprise-Bean werden angepasst und der Persistence-Manager speichert die Daten in der Datenbank. Anschließend verändert Client B den Preis von 1,50 Dollar auf *1,60 Dollar* und speichert die Daten ebenfalls. Auch in diesem Fall wird das veränderte Details-Objekt des Clients B an die Enterprise-Bean übertragen, die Attribute der Enterprise-Bean werden angepasst und anschließend vom Persistence-Manager in der Datenbank gespeichert. Als Zustand der Daten in der Datenbank würde man Folgendes erwarten:

Teilenummer	Beschreibung	Name Lieferant	Preis
0815	Abdeckung	C-Lieferant	1,60 Dollar

Tabelle 9.3: Zu erwartende Daten der Part-Bean nach der Änderung

Tatsächlich wird man jedoch folgenden Zustand vorfinden:

Teilenummer	Beschreibung	Name Lieferant	Preis
0815	Abdeckung	**A-Lieferant**	1,60 Dollar

Tabelle 9.4: Tatsächliche Daten der Part-Bean nach der Änderung

Client B hat die Änderungen von Client A überschrieben, weil beide ihre eigene Kopie des Details-Objekts hatten und Client B von der vorherigen Änderung von Client A nichts wusste. Eine solche Situation kann entstehen, wenn Client A ein Benutzer aus der Abteilung Einkauf ist und nur den Lieferanten ändern darf. Client B könnte ein Mitarbeiter aus der Rechnungsprüfung sein, der im System nur Preise ändern darf. Hätte die Enterprise-Bean einzelne get-/set-Methoden, wäre der Zustand in der Datenbank so, wie man ihn erwartet hätte. Client A hätte lediglich den Lieferant und Client B hätte lediglich den Preis verändert. Wegen der Verwendung von Details-Objekten kann ein Client aber nur alle Attribute einer Enterprise-Bean auf einmal verändern. Aus diesem Grund stellt sich das oben geschilderte Verhalten ein. Ob dieses Verhalten akzeptabel ist, hängt vom Anwendungsfall ab. Der weitere Verlauf dieses Abschnitts wird Alternativen aufzeigen, um dieses Verhalten zu verhindern.

Die Klasse PartDetails könnte so programmiert werden, dass sie sich merkt, welche Änderungen der Client vornimmt. Dazu müsste zu jedem Attribut ein Merker eingeführt werden (z.B. in Form einer boolschen Variable für jedes Attribut oder durch ein Bitfeld, bei dem jedes Bit den Änderungszustand eines bestimmten Attributs repräsentiert). Werden die Attribute der Enterprise-Bean neu gesetzt, werden nur die Attribute überschrieben, die sich im Details-Objekt tatsächlich geändert haben. Bei dieser Lösung muss darauf geachtet werden, dass die Merker nach einer erfolgreichen Änderung wieder zurückgesetzt werden. Diese Lösung ist relativ einfach zu implementieren. Die Integrität der Daten ist bei dieser Lösung in Gefahr, wenn die Merker im Details-Objekt nicht oder nicht zum richtigen Zeitpunkt zurückgesetzt werden.

Eine alternative Lösung könnte so aussehen, dass die Enterprise-Bean *Part* über einen Mechanismus, wie er in Abschnitt 9.5 vorgestellt wurde, einen Event auslöst, wenn sich ihre Daten ändern. Alle momentan aktiven Clients würden den Event empfangen und wüssten, dass ihre lokale Kopie des *PartDetails*-Objekts ungültig geworden ist. Eine Änderung der Daten ohne vorherige Aktualisierung des Details-Objekts würde bedeuten, dass eine vorangegangene Änderung eventuell überschrieben wird. Diese Vorgehensweise wäre leicht zu implementieren, kann jedoch im Extremfall zu einem hohen Netzwerkverkehr zwischen dem Server und dem Client führen. Das gilt insbesondere dann, wenn dieser Mechanismus generell bei Entity-Beans verwendet wird, um Konflikte bei der Datenspeicherung zu vermeiden. Dieser Mechanismus ist außerdem unwirksam, wenn der Client den Event ignoriert oder den Event aus irgendeinem Grund noch nicht bekommen hat.

Eine andere Lösung wäre, in die Klasse *PartDetails* einen Zeitstempel aufzunehmen. Wir wollen auf diese Alternative im Detail eingehen und werden dazu eine neue Klasse, *TsPartDetails*, verwenden. Listing 9.18 zeigt die Implementierung der Klasse *TsPartDetails*. Die Unterschiede zur Klasse *PartDetails* sind durch Hervorhebungen gekennzeichnet.

```
package ejb.part;

public class TsPartDetails implements java.io.Serializable {

    String partNumber;
    String partDescription;
    String supplierName;
    float  price;
    private long timestamp = 0;

    TsPartDetails() {
    }

    public boolean isOutDated(TsPartDetails details) {
        return details.timestamp < this.timestamp;
    }

    void updateTimestamp(javax.ejb.EntityContext ctx,
                         long tt)
    {
        if(ctx == null) {
            throw new IllegalStateException("ctx == null");
        }
        timestamp = tt;
    }

    public String getPartNumber() {
        return partNumber;
    }

    public void setPartDescription(String desc) {
        partDescription = desc;
    }

    public String getPartDescritption() {
        return partDescription;
    }

    public void setSupplierName(String name) {
        supplierName = name;
    }

    public String getSupplierName() {
        return supplierName;
    }

    public void setPrice(float p) {
        price = p;
    }

    public float getPrice() {
        return price;
```

```
    }

    public String toString() {
      StringBuffer sb = new StringBuffer("[[TsPartDetails]");
      sb.append("partNumber=").append(partNumber)
        .append(";");
      sb.append("partDescription=").append(partDescription)
        .append(";");
      sb.append("supplierName=").append(supplierName)
        .append(";");
      sb.append("price=").append(price).append(";");
      sb.append("]");
      return sb.toString();
    }

}
```

Listing 9.18: Klasse TsPartDetails

Die Methode *isOutdated* ermöglicht es, zwei *TsPartDetails*-Objekte zu vergleichen und festzustellen, ob das übergebene Objekt älter ist. Diese Methode wird die Enterprise-Bean später verwenden, um festzustellen, ob das übergebene Details-Objekt gegenüber dem von ihr referierten Objekt veraltet ist. Die Methode *updateTimestamp* wird später von der Enterprise-Bean benutzt, um nach einer Veränderung der Attribute den Zeitstempel des Details-Objekts zu aktualisieren. An die Methode muss eine Instanz des Entity-Bean-Kontextes übergeben werden, um sicherzustellen, dass nur eine Entity-Bean diese Methode aufrufen kann. Ein Client kommt auf keinem legalen Weg an ein Objekt vom Typ *javax.ejb.EntityContext*. Außerdem wird der Zeitstempel als Variable vom Typ *long* übergeben. Die Variable enthält die Zeit in Millisekunden.

Die Implementierung der von *java.lang.Object* geerbten Methode *toString* wird an dieser Stelle eingeführt, ist jedoch erst in Abschnitt 9.8.2 relevant.

Listing 9.19 zeigt das veränderte Remote-Interface der Enterprise-Bean *Part*.

```
package ejb.part;

import java.rmi.RemoteException;
import javax.ejb.EJBObject;

public interface Part extends EJBObject {

    public TsPartDetails setPartDetails(TsPartDetails pd)
        throws RemoteException, OutOfDateException;

    public TsPartDetails getPartDetails()
        throws RemoteException;

}
```

Listing 9.19: Geändertes Interface Part

Der Rückgabewert der Methode *setPartDetails* ist nun nicht mehr *void*, sondern vom Typ *TsPartDetails*. Das ist notwendig, da aus Integritätsgründen nur eine Entity-Bean den Zeitstempel des Details-Objekts aktualisieren kann. Der Client übergibt beim Aufruf der *setPartDetails*-Methode eine Kopie des Details-Objekts (bei RMI werden Objekte generell *call-by-value* übertragen). Der Zeitstempel des Details-Objekts, das von der Enterprise-Bean referiert wird, wird zwar aktualisiert, nicht aber der Zeitstempel des Details-Objekts, das der Client referiert. Damit er nach einem erfolgreichen Aufruf der Methode *setPartDetails* nicht sofort die Methode *getPartDetails* aufrufen muss, um wieder in den Besitz eines aktuellen *TsPartDetails*-Objekts zu gelangen, wird das aktualisierte Details-Objekt gleich beim Aufruf der Methode *setPartDetails* zurückgegeben. Dadurch wird ein Methodenaufruf an der Enterprise-Bean gespart.

Damit der Mechanismus sicher funktioniert, muss der Zeitstempel der letzten Änderung mit den Daten der Enterprise-Bean in der Datenbank gespeichert werden. Manche EJB-Container benutzen mehrere Instanzen einer Entity-Bean-Identität, um z. B. gleichzeitig eingehende Methodenaufrufe, die jeweils in einer eigenen Transaktionen ablaufen, an einem Zukaufteil mit z. B. der Nummer 0815 parallel durchführen zu können.

Listing 9.20 zeigt die neue Klasse *PartBean*. Die geänderten Codestellen sind durch Hervorhebungen gekennzeichnet.

```
package ejb.part;

import javax.ejb.*;

public abstract class PartBean implements EntityBean {

    private EntityContext theContext = null;
    private TsPartDetails theDetails = null;

    public PartBean() {}

    //Die create-Methode des Home-Interface

    public String ejbCreate(String partNumber)
        throws CreateException
    {
        this.setPartNumber(partNumber);
        theDetails = new TsPartDetails();
        theDetails.partNumber = partNumber;
        theDetails.partDescription = "";
        theDetails.supplierName    = "";
        theDetails.price           = 0;
        long tt = System.currentTimeMillis();
        this.setLastModified(tt);
        theDetails.updateTimestamp(theContext, tt);
        return null;
```

```
}

public void ejbPostCreate(String partNumber)
    throws CreateException
{}

//Abstrakte getter-/setter-Methoden

public abstract void setPartNumber(String num);
public abstract String getPartNumber();
public abstract void setPartDescription(String desc);
public abstract String getPartDescription();
public abstract void setSupplierName(String name);
public abstract String getSupplierName();
public abstract void setPrice(float p);
public abstract float getPrice();
public abstract long getLastModified();
public abstract void setLastModified(long tt);

//Die Methode des Remote-Interface

public TsPartDetails setPartDetails(TsPartDetails pd)
    throws OutOfDateException
{
    if(theDetails.isOutDated(pd)) {
        throw new OutOfDateException();
    }
    this.setPartDescription(pd.getPartDescritption());
    this.setSupplierName(pd.getSupplierName());
    this.setPrice(pd.getPrice());
    long tt = System.currentTimeMillis();
    this.setLastModified(tt);
    theDetails = pd;
    theDetails.updateTimestamp(theContext, tt);
    return theDetails;
}

public TsPartDetails getPartDetails() {
    return theDetails;
}

//Die Methoden des javax.ejb.EntityBean-Interface

public void setEntityContext(EntityContext ctx) {
    theContext = ctx;
}

public void unsetEntityContext() {
    theContext = null;
}
```

```
    public void ejbRemove()
        throws RemoveException
    { }

    public void ejbActivate() { }

    public void ejbPassivate() { }

    public void ejbLoad() {
        if(theDetails == null) {
            theDetails = new TsPartDetails();
        }
        theDetails.partNumber = this.getPartNumber();
        theDetails.partDescription =
                                this.getPartDescription();
        theDetails.supplierName = this.getSupplierName();
        theDetails.price = this.getPrice();
        long tt = this.getLastModified();
        theDetails.updateTimestamp(theContext, tt);
    }

    public void ejbStore() { }

}
```

Listing 9.20: Geänderte Klasse Part-Bean

In der Methode *setPartDetails* überprüft die Enterprise-Bean zuerst, ob das übergebene Details-Objekt veraltet ist. Falls das übergebene Objekt veraltet ist, löst die Enterprise-Bean eine Ausnahme vom Typ *OutOfDateException* aus. Andernfalls werden die Daten der Entity-Bean und der Zeitstempel des Details-Objekts sowie der Zeitstempel der Part-Bean aktualisiert. Abschließend wird das aktualisierte Details-Objekt an den Client zurückgegeben. Der Persistence-Manager sorgt dafür, dass nach dem Methodenaufruf der Zeitstempel zusammen mit den anderen Daten der Enterprise-Bean persistent gemacht wird. Die Methode *ejbCreate* initialisiert den Zeitstempel, die Methode *ejbLoad* setzt den Zeitstempel entsprechend des persistenten Zustands.

Der Zeitstempel ist kein öffentliches Datum der Part-Bean. Aus Effizienzgründen wird daher eine Variable vom Typ *long* statt eines Datum-Objekts verwendet. Die *long*-Variable enthält den Zeitpunkt der letzten Änderung in Millisekunden.

Listing 9.21 zeigt den Deployment-Deskriptor der Part-Bean.

```
<?xml version="1.0" ?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems,
  Inc.//DTD Enterprise JavaBeans 2.0//EN"
 "http://java.sun.com/j2ee/dtds/ejb-jar_2_0.dtd">
<ejb-jar>
  <description>
```

```
      Dieser Deployment-Deskriptor enthält Informationen
      über die Entity-Bean Part.
    </description>
    <enterprise-beans>
      <entity>
        <ejb-name>Part</ejb-name>
        <home>ejb.part.PartHome</home>
        <remote>ejb.part.Part</remote>
        <ejb-class>ejb.part.PartBean</ejb-class>
        <persistence-type>Container</persistence-type>
        <prim-key-class>java.lang.String</prim-key-class>
        <reentrant>False</reentrant>
        <cmp-version>2.x</cmp-version>
        <abstract-schema-name>PartBean</abstract-schema-name>
        <cmp-field>
          <description>Die Teilenummer</description>
          <field-name>partNumber</field-name>
        </cmp-field>
        <cmp-field>
          <description>Die Teilebeschreibung</description>
          <field-name>partDescription</field-name>
        </cmp-field>
        <cmp-field>
          <description>Der Teilelieferant</description>
          <field-name>supplierName</field-name>
        </cmp-field>
        <cmp-field>
          <description>Der Teilepreis</description>
          <field-name>price</field-name>
        </cmp-field>
        <cmp-field>
          <description>Zeitstempel der letzten
                       Änderung</description>
          <field-name>lastModified</field-name>
        </cmp-field>
        <primkey-field>partNumber</primkey-field>
      </entity>
    </enterprise-beans>
    <assembly-descriptor>
      <container-transaction>
        <method>
          <ejb-name>Part</ejb-name>
          <method-name>setPartDetails</method-name>
        </method>
        <trans-attribute>Required</trans-attribute>
      </container-transaction>
    </assembly-descriptor>
  </ejb-jar>
```

Listing 9.21: Deployment-Deskriptor der Part-Bean

Durch diese Lösung wird verhindert, dass sich Clients ihre Änderungen gegenseitig überschreiben. Falls der EJB-Container mehrere Instanzen einer Enterprise-Bean-Identität für die parallele Verarbeitung eines Methodenaufrufs in verschiedenen Transaktionen einsetzt, synchronisiert er die verschiedenen Instanzen durch den Aufruf der Methoden *ejbLoad* und *ejbStore*. Da der Zeitstempel persistent ist, greift dieser Mechanismus auch für diesen Fall.

Die in Listing 9.20 gezeigte Implementierung kann zu Problemen führen, wenn mehrere Applikationsserver im Einsatz sind und deren Systemzeit nicht übereinstimmt. Dieses Problem kann umgangen werden, wenn man die Vergabe des Zeitstempels an die Datenbank delegiert, statt die Vergabe in den Code der Enterprise-Bean aufzunehmen. Da in der Regel nur ein Datenbankserver zum Einsatz kommt, kann es nicht zu Abweichungen der Uhrzeit kommen. Eine andere Möglichkeit ist der Einsatz eines sogenannten Zeit-Servers, mit dem sich die Applikationsserver synchronisieren. Dadurch ist sichergestellt, dass die Uhrzeit auf allen Server-Rechnern übereinstimmt.

In bestimmten Situationen kann der Mechanismus mit dem Zeitstempel dazu führen, dass ein Client nicht dazu kommt, seine Änderungen zu speichern, da ihm andere Clients immer zuvorkommen. Außerdem muss ein Client eine relativ aufwändige Fehlerbehandlung für den Fall einer Ausnahme vom Typ *OutOfDateException* implementieren. Er muss sich ein akutelles Details-Objekt besorgen und versuchen, die aktualisierten Daten mit den Eingaben des Benutzers zu mischen. Ist das nicht möglich, werden die Änderungen des Benutzers verloren gehen und er wird die Eingabe wiederholen müssen. Um dieses Verhalten zu vermeiden, könnte die Vorgehensweise mit dem Zeitstempel um einen Locking-Mechanismus erweitert werden.

Wenn ein Client eine Enterprise-Bean lockt, ist sie für alle anderen Clients für schreibende Zugriffe gesperrt. Selbst wenn ein Client im Besitz eines aktuellen Details-Objekts ist, kann er die Daten nicht aktualisieren, wenn ein anderer Client die Enterprise-Bean gelockt hat. Die Implementierung eines derartigen Locking-Mechanismus ist auf viele Arten denkbar. Die Enterprise-Bean könnte den Zustand des Locks selber verwalten oder man könnte für das Locking von Enterprise-Beans einen zentralen Dienst zur Verfügung stellen. Der Lock kann persistent (d.h., der Lock bleibt auch nach einem Neustart des Servers erhalten) oder transient (d.h., der Lock ist durch einen Neustart des Servers aufgehoben) sein. Einen transienten Locking-Mechanismus im Code der Enterprise-Bean zu realisieren, ist sicherlich die einfachste Variante. In einem solchen Fall ist jedoch Vorsicht geboten, wenn diese Enterprise-Bean in zwei verschiedenen EJB-Container-Instanzen zum Einsatz kommt. Das könnte dann der Fall sein, wenn zur Verbesserung der Performance ein so genannter Server-Cluster eingerichtet wird. Statt eines EJB-Containers bzw. Applikationsservers werden mehrere identisch konfigurierte EJB-Container bzw. Applikationsserver eingesetzt, die sich die Summe der Clientanfragen teilen. Ein Applikationsserver bzw. EJB-Container kann eben nur eine bestimmte Anzahl von Client-Anfragen gleichzeitig bearbeiten. In einem solchen

Fall ist ein transienter Locking-Mechanismus, der im Code der Enterprise-Bean realisiert ist, unwirksam. Der Client will im Fall einer Entity-Bean nicht die Enterprise-Bean an sich locken. Was er will, ist der exklusive Zugriff auf das Zulieferteil mit der Teilenummer 0815. Und das kann im Fall eines Server-Clusters eben mehrmals vorhanden sein. Auch in dem Fall, dass der EJB-Container mehrere Instanzen einer Entity-Bean-Identität benutzt, um Clientanfragen in getrennten Transaktionen parallel bearbeiten zu können, ist ein transienter Locking-Mechanismus, der im Code der Enterprise-Bean realisiert wird, unwirksam. Für diese Fälle müsste ein aufwändigerer Locking-Mechanismus eingesetzt werden.

Wir werden wegen der Komplexität dieses Themas nicht weiter ins Detail gehen. Es sollte jedoch klar geworden sein, was mit einer Locking-Strategie erreicht werden kann und welche Konsequenzen getragen werden müssen.

Verwendet die Enterprise-Bean den Local- statt den Remote-Client-View, ergibt sich ein neues Problem. Beim Local-Client-View wird RMI komplett außen vor gelassen, da sich die (lokalen) Clients und die Enterprise-Bean im selben Prozess befinden. Die Semantik der Übergabe von Aufrufparametern und Rückgabewerten ist in diesem Fall nicht mehr *call-by-value*, sondern *call-by-reference*. Das in Abbildung 9.16 dargestellte Szenario würde unter Einsatz des Local-Client-View bei unverändertem Code der Part-Bean anders aussehen. Es gäbe nicht drei Instanzen des Details-Objektes (Eine je Remote-Client und eine, die von der Enterprise-Bean referiert wird), sondern es gäbe nur eine Instanz. Diese eine Instanz würde von den (lokalen) Clients und von der Enterprise-Bean referiert. Ändert ein Client das Details-Objekt, so wirkt sich diese Änderung auch auf die privaten Daten der Enterprise-Bean aus, ohne dass ein Aufruf am Bean-Interface getätigt wurde. Auch der EJB-Container bekommt von der Änderung nichts mit. Das hätte vor allem die Konsequenz, dass die Enterprise-Bean unter Umständen Daten ausliefert, die nicht mit dem momentanen persistenten Zustand übereinstimmen. Um dieses Verhalten zu vermeiden, müsste die Enterprise-Bean übergebene Details-Objekte klonen, bevor sie sie in einer internen Referenz speichert bzw. als Rückgabewert an den Client ausgibt (Siehe Listing 9.22). Damit wäre die Semantik des Methodenaufrufs von *setPartDetails* und *getPartDetails* im Local-Client-View die gleiche wie beim Remote-Client-View. Das Kopieren der Details-Objekte erfolgt in diesem Fall nicht implizit durch die Verwendung von Java-RMI, sondern explizit durch die Enterprise-Bean.

```
...

    public void setPartDetails(PartDetails pd)
    {
        this.setPartDescription(pd.getPartDescritption());
        this.setSupplierName(pd.getSupplierName());
        this.setPrice(pd.getPrice());
        this.theDetails = pd.clone();
```

```
    }

    public PartDetails getPartDetails() {
        return this.theDetails.clone();
    }

...
```

Listing 9.22: Klonen in set-/getPartDetails

Um das Klonen der Details-Objekte zu umgehen, könnte die Details-Klasse auch so entworfen werden, dass sie keine *set*-Methoden mehr hat (siehe Listing 9.23). Der lokale Client müsste, um die Daten der Enterprise-Bean zu verändern, erst ein neues Details-Objekt erzeugen. Das neu erzeugte Details-Objekt übergibt er dann der Enterprise-Bean, um die Daten der Enterprise-Bean zu verändern. Somit hat der lokale Client keinen Zugang mehr zu den privaten Daten der Enterprise-Bean. Das Problem, dass sich zwei Clients (lokal oder remote) gegenseitig ihre Änderungen überschreiben können, wird durch die *read-only* PartDetails-Klasse nicht gelöst. Sie löst lediglich ein neues, durch die *call-by-reference*-Semantik entstandenes Problem im Zusammenhang mit dem Local-Client-View.

```
package ejb.part;

public class PartDetails implements java.io.Serializable {

    String partNumber;
    String partDescription;
    String supplierName;
    float  price;

    public PartDetails(String partNumber,
                       String partDescription,
                       String supplierName,
                       float  price)
    {
        this.partNumber = partNumber;
        this.partDescription = partDescription;
        this.supplierName = supplierName;
        this.price = price;
    }

    public String getPartNumber() {
        return partNumber;
    }

    public String getPartDescritption() {
        return partDescription;
    }
```

```
    public String getSupplierName() {
        return supplierName;
    }

    public float getPrice() {
        return price;
    }

}
```

Listing 9.23: PartDetails read-only

Wenn man sich dafür entscheidet, Details-Objekte zu Zwecken der optimierten Datenübertragung zwischen Client und Enterprise-Bean einzusetzen, sollte man sich auch der Konsequenzen bewusst sein. Ob dann das in diesem Kapitel dargelegte Verhalten akzeptiert werden kann oder nicht, hängt von den Anforderungen an die Anwendung ab. Falls dieses Verhalten nicht akzeptiert werden kann, gibt es mehrere Möglichkeiten, dieses Verhalten zu verhindern. Wie weit man bei den Maßnahmen geht, d.h., ob man das Details-Objekt mit einem Zeitstempel versieht, wie ausgereift die Fehlerbehandlungsalgorithmen im Falle einer *OutOfDateException* sind, ob man zusätzlich eine Locking-Strategie einführt, ob man transiente oder persistente Locks benutzt, ob ein zentraler Locking-Service benötigt wird etc., hängt ebenfalls von den Anforderungen an die Anwendung ab. Bei der Entscheidung für eine bestimmte Vorgehensweise sollte man immer die Verhältnismäßigkeit zwischen der Komplexität der eingesetzten Lösung und dem gewonnenen Nutzen im Auge behalten.

Abschließend möchten wir noch auf die Ausführungen zum *Optimistic Locking Pattern* in Kapitel 10.3 in [IBM, 2000] hinweisen.

9.8 Qualitätssicherung

Eine Komponente erhebt für sich den Anspruch der Wiederverwendbarkeit und kann in verschiedenen Anwendungen unter verschiedenen Bedingungen eingesetzt werden. Dabei gibt sie nur ihre Schnittstelle preis und verbirgt ihr Inneres. Ferner soll sie mit anderen Komponenten zu Superkomponenten oder Teilanwendungen aggregiert werden können. Eine Komponente kann diesem Anspruch nur dann gerecht werden, wenn die Implementierung von entsprechend hoher Qualtität ist. Ein bewährtes Werkzeug bei der Qualitätssicherung für Software im Allgemeinen sind automatisierte Tests. Durch Tests können Fehler in der Implementierung entdeckt werden. Mit jedem behobenen Fehler verbessert sich die Qualität der Software. Automatisierte Tests können ohne weiteres auch für die Qualitätssicherung von Komponenten eingesetzt werden.

Um einen durch Tests festgestellten Fehler zu finden, gibt es verschiedene Vorgehensweisen. Eine weit verbreitete Vorgehensweise ist der Einsatz von einem so genannten *Debugger*. Ein Debugger ermöglicht es dem Entwickler, sich auf einen Prozess aufzuschalten und die Ausführung des Prozesses zu überwachen und zu kontrollieren. Während der Ausführung lässt sich der Zustand von Variablen abfragen und bei manchen Debuggern sogar verändern. Eine andere, nicht weniger weit verbreitete Vorgehensweise ist der Einsatz von *Logging*. Unter Logging versteht man im Allgemeinen die Ausgabe von Meldungen auf dem Bildschirm oder in eine Datei. Welche Meldungen ausgegeben werden, bestimmt der Entwickler, da er die Ausgaben implementiert. Üblicherweise werden Informationen ausgegeben, mit denen der Verlauf der Anwendung sowie der Zustand von Variablen rekonstruiert werden kann. Beim Logging hat der Entwickler im Wesentlichen die gleichen Informationen zur Verfügung wie beim Einsatz von Debugging-Software. Er ist jedoch beim Logging von der Qualität der Logging-Ausgaben im Code der Anwendung abhängig. Außerdem hat der Entwickler beim Verfolgen des Programmablaufs über Logging-Ausgaben keine Möglichkeit, in das Geschehen einzugreifen. Debugging-Software erlaubt es dem Entwickler, die Ausführung des Programms anzuhalten, die Ausführung Schritt für Schritt abzuarbeiten oder in der Ausführung an eine bestimmte Stelle im Code zu springen.

Der Einsatz von Debuggern wird bei Enterprise JavaBeans dadurch erschwert, dass es sich hier um verteilte Anwendungen handelt. Der Debugger muss sich auf einen entfernten Prozess aufschalten. Bei der Ausführungsumgebung der Enterprise-Beans, dem Applikationsserver, handelt es sich um einen sehr schwergewichtigen Prozess, in dem viele Threads gleichzeitig aktiv sind. Die Analyse der Abläufe innerhalb eines solchen schwergewichtigen Prozesses ist sehr zeitaufwändig und mühsam und erfordert sehr viel Sachverstand.

In diesem Kapitel wollen wir zunächst ein kleines Framework zum Testen von Enterprise-Beans entwickeln. Damit werden wir Testfälle für die Entity-Bean BankAccount aus Kapitel 3 und die Entity-Bean Part aus Abschnitt 9.7 dieses Kapitels schreiben. Die Besonderheit des Test-Frameworks liegt darin, dass für jeden Testlauf ein Ergebnisprotokoll im HTML-Format generiert wird. Damit kann die Entwicklungshistorie einer Enterprise-Bean dokumentiert und die Funktionstüchtigkeit der Komponente belegt werden. Das Testframework ist klein, leicht zu handhaben und einfach an individuelle Bedürfnisse anpassbar. Zum anderen zeigen wir den Einsatz von Logging in Enterprise-Beans. Mit Hilfe von Logging lassen sich in den Tests oder später im produktiven Einsatz aufgetretene Fehler lokalisieren. Logging und automatische Tests leisten so gemeinsam einen wesentlichen Beitrag zur Qualitätssicherung von Enterprise-Beans.

9.8.1 Tests

Software muss sich permanent Änderungen unterziehen lassen. Dies trifft auch auf Komponenten zu. Die Anforderungen an Software und damit auch an Komponenten wachsen mit der Erfahrung, die man unter ihrem Einsatz gewinnt. Änderungen an einer Software bzw. einer Komponente vorzunehmen birgt die Gefahr, Fehler zu begehen. Durch das Vorhandensein von Tests kann sofort nach der Codeänderung sichergestellt werden, dass die Komponente noch so arbeitet, wie sie das vor der Änderung getan hat (vorausgesetzt, die Tests arbeiten korrekt).

Bei der Entwicklung von Anwendungen werden häufig Komponenten von Dritten eingesetzt. Die Entwickler verlassen sich dabei auf ein bestimmtes Verhalten der verwendeten Komponente. Ändert sich das Verhalten der verwendeten Komponente in einer neuen Version plötzlich, kann dies die Funktionstüchtigkeit der Anwendung gefährden. Auch für Komponenten von Dritten ist es sinnvoll, Tests zu programmieren. Diese Tests stellen sicher, dass die verwendete Komponente über mehrere Versionen hinweg das Verhalten zeigt, das von ihr erwartet wird. Bevor eine neue Version dieser Komponente zum Einsatz kommt, kann man durch Tests bereits im Vorfeld feststellen, ob sich das Verhalten der Komponente geändert hat. Abgesehen davon können Komponenten von Drittanbietern auch fehlerhaft sein und die Funktionstüchtigkeit des Systems beeinträchtigen.

Es gibt im Wesentlichen zwei verschiedene Arten von Tests, *Blackbox*-Tests und *Whitebox*-Tests. Blackbox-Tests testen die Funktionstüchtigkeit einer Komponente, indem sie feststellen, ob ihre Schnittstelle hält, was sie verspricht. Der interne Zustand der Komponenten wird nicht gestestet. Es wird auch kein Wissen über den Aufbau der Komponente für die Auswahl der Testfälle herangezogen. Speziell bei Enterprise-Beans würde das bedeuten, dass im Fall eines Blackbox-Tests eine Komponente nur über die Sicht des Clients getestet wird. Whitebox-Tests testen die Funktionsfähigkeit einer Komponente, indem sie den internen Zustand der Komponente während der Ausführung überprüfen und speziell auf die Implementierung abgestimmte Testfälle verwenden. In der Regel verwendet man bei Softwaretests eine Mischung aus beiden Elementen. Whitebox-Tests sind bei Enterprise-Beans nur schwer durchzuführen. Eine Enterprise-Bean braucht immer eine Laufzeitumgebung. Für Whitebox-Tests müsste man diese Laufzeitumgebung simulieren, um das Verhalten und den Zustand einer Entperprise-Bean zu beobachten, da sie sonst vollständig vom EJB-Container abgeschirmt ist und dieser keinen direkten Zugriff auf die Instanz der Enterprise-Bean erlaubt. Eine solche Vorgehensweise ist sehr aufwändig in der Implementierung. Wir werden uns in diesem Kapitel auf Blackbox-Tests beschränken. Später werden wir in diesem Kapitel noch auf eine Möglichkeit hinweisen, wie man Whitebox-Tests an Enterprise-Beans mit etwas weniger Aufwand durchführen kann.

Das Testframework für Blackbox-Tests an Enterprise-Beans setzt sich im Wesentlichen aus zwei Klassen und drei Ausnahmen (Exceptions) zusammen. Eine der beiden Klassen bildet die Grundlage für die Testfälle. Die Klasse heißt *EJBTestCase* und ist in Listing 9.24 dargestellt.

```
package ejb.test;

import java.util.Properties;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public abstract class EJBTestCase {

    private Properties theProps;
    private Context    theContext;

    public EJBTestCase() {
    }

    void setProperties(Properties p)
        throws NamingException
    {
        if(p == null) {
            throw new
                IllegalArgumentException("null is not allowed!");
        }
        theProps   = p;
        theContext = new InitialContext(theProps);
    }

    public Object lookup(String name)
        throws NamingException
    {
        this.assertNotNull("name", name);
        return theContext.lookup(name);
    }

    public Object narrow(String name, Class c)
        throws NamingException
    {
        this.assertNotNull("name", name);
        this.assertNotNull("class", c);
        Object o = theContext.lookup(name);
        return javax.rmi.PortableRemoteObject.narrow(o, c);
    }

    public void fail(String msg) {
        throw new TestFailException(msg);
    }
```

```
    public void assertEquals(Object obj1, Object obj2) {
        assertEquals("values do not match", obj1, obj2);
    }

    public void assertEquals(float f1, float f2) {
        assertEquals("values do not match", f1, f2);
    }

    public void assertEquals(int i1, int i2) {
        assertEquals("values do not match", i1, i2);
    }

    public void assertEquals(String msg, Object o1, Object o2) {
        if(!o1.equals(o2)) {
            throw new
                AssertionException(msg + ": " +o1 + " != " + o2);
        }
    }

    public void assertEquals(String msg, float f1, float f2) {
        if(f1 != f2) {
            throw new
                AssertionException(msg + ": " +f1 + " != " + f2);
        }
    }

    public void assertEquals(String msg, int i1, int i2) {
        if(i1 != i2) {
            throw new
                AssertionException(msg + ": " +i1 + " != " + i2);
        }
    }

    public void assertNotNull(String name, Object obj) {
        if(obj == null) {
            throw new AssertionException(name + " is null");
        }
    }

    public Properties getProperties() {
        return theProps;
    }

    public abstract void prepareTest()
        throws Exception;

    public abstract void finalizeTest()
        throws Exception;

}
```

Listing 9.24: Klasse EJBTestCase

EJBTestCase ist eine abstrakte Klasse und ist die Basisklasse aller Testfälle. Sie wird mit einem Properties-Objekt initialisiert, das in erster Linie dazu benutzt wird, einen NamingContext zu erzeugen. Abgeleitete Klassen können über die Methode *lookup* bzw. *narrow* diesen NamingContext indirekt benutzen, um sich Referenzen auf Enterprise-Beans zu besorgen. Das Properties-Objekt wird aus einer Konfigurationsdatei erzeugt. Über dieses Properties-Objekt wäre es also auch möglich, einen Testfall von außen zu parametrisieren. Die abgeleitete Klasse muss die Methoden *prepareTest* und *finalizeTest* überschreiben. Die eigentlichen Tests muss die abgeleitete Klasse per Konvention in Methoden implementieren, deren Namen mit *test* beginnen. Diese Methoden müssen *public* deklariert sein und dürfen keine Übergabeparameter haben. Die Methode *prepareTest* wird aufgerufen, bevor die erste Testmethode aufgerufen wird und dient zur Initialisierung des Testfalls. Danach werden die einzelnen Testmethoden aufgerufen. Abschließend wird die Methode *finalizeTest* aufgerufen. Sie dient zur Durchführung von Aufräumarbeiten. Den Lesern, die mit dem Testframework *Junit* vertraut sind (vgl. [JUNIT]), dürfte diese Vorgehensweise bekannt sein.

Die Klasse, die für die Durchführung der Tests sorgt, heisst *EJBTest* und wird in Listing 9.25 dargestellt. Aus Gründen des Umfangs werden wir nur die zentralen Methoden der Klasse in voller Länge zeigen, da nur sie für das Verständnis der Vorgehensweise notwendig sind. Der vollständige Source-Code kann von der auf dem Buchumschlag angegebenen Quelle heruntergeladen werden.

```
package ejb.test;

import java.io.*;
import java.util.*;
import java.text.DateFormat;
import java.lang.reflect.Method;

public final class EJBTest {

    ...
    //Member - Variablen
    private static final String TC  = "test.class.";
    private Properties   theProps;
    private Object[]     theClasses;
    ...

    public EJBTest() {
        ...
        //Initialisierung der Member-Variablen
        ...
    }

    public void init(String propertyFile) {
        Properties p;
        ...
```

```
        //Einlesen der property-Datei (siehe unten),
        //speichern der Werte in der Variable p und
        //Weitergabe an init(Properties)
        ...
        init(p);
    }

    public void init(Properties p) {
        if(p == null) {
        throw new IllegalArgumentException("null …");
        }
        theProps = p;
        ArrayList al = new ArrayList();
        Enumeration e = theProps.propertyNames();
        String name;
        String cname;
        Class  c;
        while(e.hasMoreElements()) {
            name = (String)e.nextElement();
            if(name.startsWith(TC)) {
                cname = theProps.getProperty(name);
                try {
                    c = Class.forName(cname);
                    al.add(c);
                } catch(Exception ex) {
                    al.add(cname);
                }
            }
        }
        theClasses = al.toArray();
        initOutputBuffer();
    }

    private void initOutputBuffer() {
        ...
        //Initialisieren des Ausgabepuffers
        //für den Testbericht, der am Ende aller
        //Tests im HTML-Format erstellt wird.
        ...
    }

    private void reportBeginSection
                        (String name, Throwable t)
    {
        ...
        //Ausgabe von Meldungen, die zu Beginn eines
        //Tests anfallen im HTML-Format.
        ...
    }

    private void reportTestCase
```

```
                (String name, long time, Throwable t)
{
    ...
    //Ausgabe von Meldungen, die während einem Test
    //anfallen im HTML-Format
    ...
}

private void reportEndSection(String name) {
    ...
    //Ausgabe von Meldungen, die am Ende eines Tests
    //anfallen im HTML-Format
    ...
}

private void closeOutputBuffer() {
    ...
    //Schliessen des Ausgabepuffers nach Durchführung
    //aller Tests
    ...
}

private String formatThrowable(Throwable t) {
    ...
    //Formatieren einer Ausnahme für die Ausgabe
    //im HTML-Format
    ...
}

private String computeFileName(String path) {
    ...
    //Berechnen des Dateinamens für das Testprotokoll
    ...
}

private String format(int num) {
    ...
    //Formatieren von Integern für die Ausgabe im
    //HTML-Format
    ...
}

public void runTests() {
    Class       cl;
    EJBTestCase tc;
    for(int i = 0; i < theClasses.length; i++) {
        if(theClasses[i] instanceof String) {
            try {
                cl = Class.forName
                            ((String)theClasses[i]);
            } catch(Exception ex) {
```

```
                reportBeginSection
                        ((String)theClasses[i], ex);
                continue;
            }
        } else {
            cl = (Class)theClasses[i];
        }
        try {
            tc = (EJBTestCase)cl.newInstance();
            tc.setProperties(theProps);
        } catch(Exception ex) {
            reportBeginSection(cl.getName(), ex);
            continue;
        }
        reportBeginSection(cl.getName(), null);
        runTest(tc);
        reportEndSection(cl.getName());
    }
    closeOutputBuffer();
}

private void runTest(EJBTestCase tc) {
    Class    c  = tc.getClass();
    Method[] ms = c.getMethods();
    String   name;
    Class[]  params;
    try {
        tc.prepareTest();
    } catch(Exception ex) {
        reportTestCase("prepareTest", 0, ex);
        return;
    }
    for(int i = 0; i < ms.length; i++) {
        name   = ms[i].getName();
        params = ms[i].getParameterTypes();
        if(!(name.startsWith(MP) &&
             params.length == 0))
        {
            continue;
        }
        try {
            long t1 = System.currentTimeMillis();
            ms[i].invoke(tc, params);
            long t2 = System.currentTimeMillis();
            reportTestCase(name, (t2 - t1), null);
        } catch(Exception ex) {
            reportTestCase(name, 0, ex);
        }
    }
    try {
        tc.finalizeTest();
```

```
            } catch(Exception ex) {
                reportTestCase("finalizeTest", 0, ex);
            }
        }

        public static void main(String[] args) {
            EJBTest et = new EJBTest();
            if(args.length == 1) {
                et.init(args[0]);
            } else {
                et.init(new Properties());
            }
            et.runTests();
        }

    }
```

Listing 9.25: Klasse EJBTest

Die Klasse *EJBTest* wird durch eine Konfigurationsdatei initialisiert. Listing 9.26 zeigt diese Konfigurationsdatei. Sie kann ebenfalls von der auf dem Umschlag angegebenen Quelle heruntergeladen werden.

```
#
# JNDI Einstellungen
#
# Die JNDI-Factory-Klasse
java.naming.factory.initial=weblogic.jndi.WLInitialContextFactory
# Die JNDI-URL
java.naming.provider.url=t3://localhost:7001

#
# Datenbank Einstellungen für Testfälle,
# die direkt auf die Datenbank zugreifen
#
# Die Klasse des Datenbanktreibers
jdbc.driver=org.postgresql.Driver
# Die Datenbank-URL
jdbc.url=jdbc:postgresql://localhost/postgres
# Der Benutzername für den Datenbank-Login
jdbc.user=postgres
# Das zugehoerige Passwort
jdbc.pwd=postgres

#
# Die Klassen, die die Tests enthalten
#
test.class.0=ejb.testcases.TestPartBean
test.class.1=ejb.testcases.TestBankAccount
test.class.2=ejb.testcases.TestLogger
test.class.3=ejb.testcases.TestCounterBean
```

```
test.class.4=ejb.testcases.TestExchangeSLBean
test.class.5=ejb.testcases.TestExchangeSFBean
test.class.6=ejb.testcases.TestMigrationBean
test.class.7=ejb.testcases.TestSupplychain

#
# Das Verzeichnis, in welches die Reports
# geschrieben werden sollen
#
test.output.dir=./
```

Listing 9.26: Properties für die Klasse EJBTest

Über die *main*-Methode der Klasse *EJBTest* wird die Methode *init* aufgerufen. Dort wird die Konfigurationsdatei geladen und gelesen. Die Namen der Klassen, die einen Test implementieren (sie werden in die Konfigurationsdatei mit dem Schlüssel *test.class.** eingetragen), werden separat gespeichert. Im Beispiel sind das unter anderen die Klassen *ejb.test.TestPartBean* und *ejb.test.TestBankAccount* (die Implementierung der Klassen folgt weiter unten). Nach der Initialisierung werden die Tests über die Methode *runTests* durchgeführt. Dazu wird für jede Testklasse eine Instanz erzeugt. Die Instanz wird durch den Aufruf der Methode *setProperties* initialisiert. In der Methode *runTest* wird der jeweilige Test dann ausgeführt. Dazu wird der Testfall durch den Aufruf der Methode *prepareTest* vorbereitet. Über den Introspection-Mechanismus der Programmiersprache Java werden alle Methoden an diesem Objekt aufgerufen, die *public* deklariert sind, deren Name mit *test* beginnt und die keine Übergabeparameter erwarten. Diese Methoden implementieren per Konvention die Tests für die jeweiligen Enterprise-Beans. Nachdem alle Test-Methoden aufgerufen wurden, wird der Test mit dem Aufruf der Methode *finalizeTest* abgeschlossen. Alle Informationen über den Test (Name, Ausführungsdauer, aufgetretene Fehler etc.) werden in die Protokolldatei geschrieben. Für jeden Testlauf wird einen neue Protokolldatei angelegt. Der Name der Datei enthält Datum und Uhrzeit. Ein Beispiel für eine solche Protokolldatei folgt weiter unten in diesem Abschnitt.

Nachdem nun die Grundlagen gelegt sind, können wir die Tests für die Enterprise-Beans schreiben. Listing 9.27 zeigt den Test für die Enterprise-Bean *Part* aus Abschnitt 9.7 dieses Kapitels.

```
package ejb.testcases;

import ejb.part.*;
import ejb.test.*;
import javax.ejb.*;

public class TestPartBean extends EJBTestCase {

    PartHome partHome;
    Part p1;
```

```
Part p2;
Part p3;

public TestPartBean() {
    super();
}

public void prepareTest()
    throws Exception
{
    partHome = (PartHome)narrow("Part", PartHome.class);
}

public void testCreate()
    throws Exception
{
    final String one = "11111";
    p1 = partHome.create(one);
    assertNotNull("Part", p1);
}

public void testDelete()
    throws Exception
{
    final String two = "22222";
    p2 = partHome.create(two);
    assertNotNull("Part", p2);
    p2.remove();
    try {
        p2 = partHome.findByPrimaryKey(two);
        fail("expected FinderException, part "
             + two + " should not exist");
    } catch(FinderException fex) {
        //expected
    }
}

public void testUpdate()
    throws Exception
{
    final String three = "33333";
    p3 = partHome.create(three);
    assertNotNull("Part", p3);
    TsPartDetails pd = p3.getPartDetails();
    pd.setPartDescription("Test Part");
    pd.setSupplierName("Test Supplier");
    pd.setPrice(120);
    TsPartDetails pd1 = p3.setPartDetails(pd);
    assertEquals(pd.getPartNumber(),
                 pd1.getPartNumber());
    assertEquals(pd.getPartDescritption(),
```

```
                    pd1.getPartDescritption());
            assertEquals(pd.getSupplierName(),
                    pd1.getSupplierName());
            assertEquals(pd.getPrice(),
                    pd1.getPrice());
            try {
                p3.setPartDetails(pd);
                fail("expected OutOfDateException");
            } catch(OutOfDateException ex) {
                //expected
            }
        }

        public void finalizeTest()
            throws Exception
        {
            p1.remove();
            p3.remove();
        }

    }
```

Listing 9.27: Klasse TestPartBean

Der Testfall überprüft die drei wesentlichen Funktionen der Enterprise-Bean *Part*. Das sind das Erzeugen, das Verändern und das Löschen eines Zukaufsteils. Während des Testlaufs wird durch die Klasse *EJBTest* zunächst die Methode *prepareTest* aufgerufen. Dort beschafft sich der Test eine Referenz auf das Home-Interface der Part-Bean. Danach werden durch den Introspection-Mechanismus der Klasse EJBTest die drei Testmethoden, *testCreate*, *testDelete* und *testUpdate* der Klasse *TestPartBean* aufgerufen. Die Reihenfolge des Aufrufs ist dabei zufällig. Um den Test abzuschließen, wird die Methode *finalizeTest* aufgerufen.

In der Methode *testCreate* wird die Methode *create* des Home-Interface getestet. Die Methode *assertNotNull* der Basisklasse überprüft, ob das übergebene Objekt den Wert *null* hat. Falls ja, wird eine Ausnahme vom Typ *AssertionException* ausgelöst und der Test wird abgebrochen. Die Ausnahme wird von *EJBTest* aufgefangen und in der Ausgabedatei protokolliert.

In der Methode *testDelete* wird eine soeben erzeugte Part-Bean wieder gelöscht. Nach dem Löschen wird versucht, genau diese Enterprise-Bean über die Methode *findByPrimaryKey* wiederzufinden. Korrekterweise muss die *findByPrimaryKey*-Methode eine Ausnahme vom Typ *FinderException* auslösen, da diese Enterprise-Bean nicht mehr existiert. Falls sie das nicht tut, ist der Test fehlgeschlagen. Die Methode *fail* der Basisklasse löst eine Ausnahme vom Typ *TestFailException* aus. Auch diese Ausnahme wird von *EJBTest* abgefangen und in der Ausgabedatei protokolliert.

Die Methode *testUpdate* erzeugt eine Part-Bean und verändert ihre Werte. Danach stellt sie sicher, dass die Bean auch wirklich die veränderten Werte zurückgibt. Die Methode *assertEquals* vergleicht zwei Objekte oder zwei Werte. Sind sie nicht gleich, löst diese Methode eine Ausnahme vom Typ *AssertionException* aus und der Test wird abgebrochen. Auch in diesem Fall würde die Ausnahme von *EJBTest* abgefangen und in der Ausgabedatei protokolliert. Außerdem testet die Methode *testUpdate* den Timestamp-Mechanismus, der in Abschnitt 9.7 dieses Kapitels mit Hilfe der Part-Bean vorgestellt wurde. Wird die Methode *setPartDetails* mit einem veralteten Details-Objekt aufgerufen, löst diese Methode eine Ausnahme vom Typ *OutOfDateException* aus. Tut sie das nicht, wird der Testfall durch den Aufruf der Methode *fail* (und dem impliziten Werfen einer Ausnahme vom Typ *TestFailException*) abgebrochen.

Wie oben bereits erwähnt, handelt es sich hierbei um einen Blackbox-Test. Es werden nur die Methoden getestet, die über die öffentliche Schnittstelle der Enterprise-Bean verfügbar sind. Der Test bezieht nicht die internen Zustände der Enterprise-Bean während der Methodenaufrufe mit ein. In diesem Beispiel handelt es sich um sehr einfache Tests, die lediglich sicherstellen, dass die Grundfunktionen der Komponente nutzbar sind. Solche Tests können beliebig ausführlich und beliebig komplex gestaltet werden.

Der Test für die Enterprise-Bean *BankAccount* ist nach genau dem gleichen Schema aufgebaut und testet die gleichen Aspekte der Enterprise-Bean. Deswegen verzichten wir auf ein Listing dieses Tests. Der Quellcode kann aber von der auf dem Buchumschlag angegebenen Quelle heruntergeladen werden.

Um einen Testlauf durchzuführen, müssen die beiden Enterprise-Beans in einem EJB-Container installiert werden. Der EJB-Container bzw. der Applikationsserver muss gestartet werden, da die Enterprise-Beans eine Laufzeitumgebung brauchen. Danach kann der Testlauf durch den Aufruf der Methode *main* in der Klasse *EJBTest* durchgeführt werden. Als Argument wird der Name der Konfigurationsdatei übergeben (Listing 9.26 zeigt ein Beispiel). Das Ergebnis des Testlaufs ist eine Protokolldatei im HTML-Format. Abbildung 9.17 zeigt das Ergebnis eines Testlaufs für die Testfälle der Part- und BankAccount-Bean.

Aus dem Testprotokoll geht hervor, dass alle Tests erfolgreich waren. Fehlgeschlagene Tests werden in einem rot statt grün markierten Feld der Tabelle dargestellt. Statt des Wortes *success* wird die Ausnahme ausgegeben, die den Testabbruch verursacht hat.

Nach jeder Änderung an einer der beiden Komponenten sollte über die Tests sichergestellt werden, dass der Schnittstellenvertrag der Komponenten immer noch erfüllt wird.

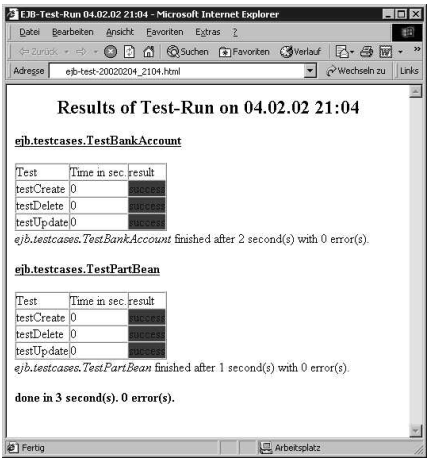

Abbildung 9.17: Testergebnis für TestBankAccount und TestPartBean

9.8.2 Logging

Logging ist eine relativ einfache Methode, um Fehler in Software zu finden. Man benutzt Logging, um Fehler einzugrenzen und wenn keine geeignete Debugger-Software zur Verfügung steht. Wenn bei einem Kunden ein Fehler in der Software aufgetreten ist, kann er meistens über die Log-Datei rekonstruiert werden. Selbst wenn man geeignete Debugger-Software hat, ist es oft nicht möglich, sie beim Kunden einzusetzen.

Logging wird oft mit den Argumenten kritisiert, dass es den Code aufbläht und die Leserlichkeit des Codes beeinträchtig. Ein anderes, häufig gegen Logging vorgebrachtes Argument betrifft die Beeinträchtigung der Performanz. Die Befürworter von Logging argumentieren, dass Logging-Code die Lesbarkeit des übrigen Codes verbessert, da die Logging-Statements dokumentierenden Charakter haben. Außerdem sei die Anwendung von Logging oft einfacher und weniger zeitraubend als die Handhabung komplizierter Debugging-Software. Die Einbußen in der Performanz seien sehr gering und gerade bei verteilten Anwendungen leicht in Kauf zu nehmen. In diesem Segment ist Logging oft die einzige Möglichkeit, um Fehler zu finden und zu analysieren.

Viele Applikationsserver bieten Logging-Services an, die eine Enterprise-Bean benutzen kann. Diese Logging-Services können mehr oder weniger flexibel einsetzbar sein. Auf jeden Fall haben sie den Nachteil, dass ihre Schnittstelle proprietär ist. Benutzt eine Enterprise-Bean einen solchen proprietären Logging-Service eines Applikationsservers, kann man sie in keinem anderen Applikationsserver mehr einsetzen, ohne den Code der Enterprise-Bean zu ändern. Die Wiederverwendbarkeit einer Enterprise-Bean ist dadurch stark eingeschränkt. Sun Microsystems plant im Übrigen, einen Logging-Dienst in die Laufzeitumgebung von Java zu integrieren. Solange dieser nicht zur

Verfügung steht, wäre der Einsatz eines Logging-Tools eines Drittanbieters eine Alternative. Soll die Enterprise-Bean in einem anderen Applikationsserver zum Einsatz kommen, muss ihr das Logging-Tool dieses Anbieters zur Verfügung gestellt werden. Im Deployment-Deskriptor der Enterprise-Bean kann auf diese Abhängigkeit hingewiesen werden. Ein Logging-Tool sollte nach Möglichkeit frei zur Verfügung stehen.

Ein solches Tool existiert in Form der *log4j*-Bibliothek des Apache-Projekts. Dieses Projekt (vgl. [APACHE]) ist ein über das Internet organisiertes und koordiniertes Entwicklungsprojekt zur Erstellung und Pflege von Software, die jedem kostenlos zur Verfügung steht. Diese Art von Software bezechnet man auch als Open-Source-Software, da auch der Quellcode der Software frei zur Verfügung gestellt wird. *Log4j* (Logging for Java) ist ein Teilprojekt, das ein Logging-Tool für die Programmiersprache Java zur Verfügung stellt. Die Flexibilität von *log4j* ist sehr hoch und die Benutzung relativ einfach. *Log4j* besitzt unter anderem die Fähigkeit, die Logging-Ausgaben auf verschiedene Medien zu leiten (z.B. in Dateien, auf denBildschirm und auf andere Server). *Log4j* unterstützt verschiedene Logging-Ebenen und hierarchische Logging-Kategorien. Eine ausführliche Beschreibung des Leistungsumfangs von *log4j* ist unter [LOG4J] zu finden.

Durch die Ausgabe von Meldungen über *log4j*, sei es auf den Bildschirm oder in eine Datei, kann das Verhalten einer Enterprise-Bean beobachtet werden. Für Entwickler, die sich erstmalig mit dem Komponentenmodell der Enterprise JavaBeans beschäftigen, ist der Einsatz von Logging eine Möglichkeit, über die Logging-Ausgaben der Enterprise-Bean das Verhalten des EJB-Containers kennen zu lernen. Der primäre Zweck von Logging ist jedoch das Auffinden von Fehlern.

Um den Einsatz von *log4j* zu demonstrieren, wollen wir die Enterprise-Bean Part aus Abschnitt 9.7 dieses Kapitels mit Logging-Ausgaben versehen. Unter Zuhilfenahme des Blackbox-Tests aus dem vorangegangenen Abschnitt werden wir dann im zweiten Schritt eine Logging-Datei erzeugen. Listing 9.28 zeigt die geänderte Klasse *PartBean*. Die für das Logging relevanten Teile sind durch Hervorhebungen gekennzeichnet. Um den internen Zustand der Enterprise-Bean aufzeichnen zu können, wurde die Methode *toString* in der Klasse *TsPartDetails* implementiert (vgl. Listing 9.18 in Abschnitt 9.7).

```
package ejb.part;

import javax.ejb.*;
import org.apache.log4j.Category;
import org.apache.log4j.BasicConfigurator;
import org.apache.log4j.FileAppender;
import org.apache.log4j.Layout;
import org.apache.log4j.PatternLayout;
import org.apache.log4j.Priority;

public abstract class PartBean implements EntityBean {

    private EntityContext theContext = null;
    private TsPartDetails theDetails = null;
```

```
private Category log = null;

public PartBean() {
    final String name = "PartBean";
    BasicConfigurator.configure();
    Category.getRoot().removeAllAppenders();
    if((log = Category.exists(name)) == null) {
        log = Category.getInstance(name);
        log.setPriority(Priority.DEBUG);
        Layout l = new PatternLayout(
                        PatternLayout.TTCC_CONVERSION_PATTERN);
        try {
          log.addAppender(new FileAppender(l, name + ".log"));
        } catch(java.io.IOException ioex) {
           throw new IllegalStateException(ioex.getMessage());
        }
    }
    log.info("initialized instance succesfully");
}

//Die create-Methode des Home-Interface

public String ejbCreate(String partNumber)
    throws CreateException
{
    log.info("entering ejbCreate (" + partNumber + ")");
    this.setPartNumber(partNumber);
    theDetails = new TsPartDetails();
    theDetails.partNumber = partNumber;
    theDetails.partDescription = "";
    theDetails.supplierName    = "";
    theDetails.price           = 0;
    long tt = System.currentTimeMillis();
    this.setLastModified(tt);
    theDetails.updateTimestamp(theContext, tt);
    log.debug("Created new part with part-no:" + partNumber);
    log.info("leaving ejbCreate");
    return null;
}

public void ejbPostCreate(String partNumber)
    throws CreateException
{
    log.debug("ejbPostCreate (" + partNumber + ")");
}

//Abstrakte getter-/setter-Methoden

public abstract void setPartNumber(String num);
public abstract String getPartNumber();
```

```
public abstract void setPartDescription(String desc);
public abstract String getPartDescription();
public abstract void setSupplierName(String name);
public abstract String getSupplierName();
public abstract void setPrice(float p);
public abstract float getPrice();
public abstract long getLastModified();
public abstract void setLastModified(long tt);

//Die Methode des Remote-Interfaces

public TsPartDetails setPartDetails(TsPartDetails pd)
    throws OutOfDateException
{
    log.info("entering setPartDetails " + pd);
    if(theDetails.isOutDated(pd)) {
        log.warn("out of date part-details-object");
        log.info("leaving setPartDetails");
        throw new OutOfDateException();
    }
    this.setPartDescription(pd.getPartDescritption());
    this.setSupplierName(pd.getSupplierName());
    this.setPrice(pd.getPrice());
    long tt = System.currentTimeMillis();
    this.setLastModified(tt);
    theDetails = pd;
    theDetails.updateTimestamp(theContext, tt);
    log.debug("part-data updated " + theDetails);
    log.info("leaving setPartDetails");
    return theDetails;
}

//public PartDetails getPartDetails() {
public TsPartDetails getPartDetails() {
    log.debug("getPartDetails :" + theDetails);
    return theDetails;
}

//Die Methoden des javax.ejb.EntityBean-Interface

public void setEntityContext(EntityContext ctx) {
    log.debug("setEntityContext " + ctx);
    theContext = ctx;
}

public void unsetEntityContext() {
    log.debug("unsetEntityContext");
    theContext = null;
}

public void ejbRemove()
```

```
        throws RemoveException
    {
        log.debug("ejbRemove");
    }

    public void ejbActivate() {
        log.debug("ejbActivate");
    }

    public void ejbPassivate() {
        log.debug("ejbPassivate");
    }

    public void ejbLoad() {
        log.info("entering ejbLoad");
        if(theDetails == null) {
            theDetails = new TsPartDetails();
        }
        theDetails.partNumber = this.getPartNumber();
        theDetails.partDescription = this.getPartDescription();
        theDetails.supplierName = this.getSupplierName();
        theDetails.price = this.getPrice();
        long tt = this.getLastModified();
        theDetails.updateTimestamp(theContext, tt);
        log.debug("data successfully loaded:" + theDetails);
        log.info("leaving ejbLoad");
    }

    public void ejbStore() {
        log.debug("ejbStore");
    }

}
```

Listing 9.28: Klasse PartBean mit Logging-Ausgaben

Im Konstruktor der Enteprise-Bean werden die *log4j*-Klassen durch den Aufruf der Methode *configure* an der Klasse *BasicConfigurator* initialisiert. Diese Methode kann ohne weiteres mehrmals hintereinander aufgerufen werden. Anschließend wird eine eigene Logging-Kategorie für die Enterprise-Bean *Part* mit dem Namen der Klasse angelegt, falls sie nicht schon vorhanden ist (eine andere Instanz dieser Bean-Klasse könnte diese Kategorie bereits erzeugt haben). Das Interface *Category* ist die zentrale Schnittstelle der *log4j*-Bibliothek. Alle Logging-Ausgaben werden über dieses Interface gemacht.

Die Logging-Ebene der Kategorie für die Part-Bean wird über die Methode *setPriority* auf *DEBUG* gesetzt. *Log4j* definiert die Logging-Ebenen *DEBUG*, *INFO*, *WARN*, *ERROR* und *FATAL*, wobei *DEBUG* die geringfügigste und *FATAL* die höchste Priorität hat. Für jede Priorität steht am Interface *Category* eine eigene Methode zur Verfügung (*debug*, *info*, *warn*, *error* und *fatal*). Um eine Logging-Ausgabe mit der Priorität *WARN* zu machen, muss die Methode *warn* am Interface *Category* aufgerufen werden. Ist zur Lauf-

zeit die Logging-Ebene auf *ERROR* eingestellt, erzeugt ein Aufruf der Methode *warn* keine Ausgabe. In unserem Fall erzeugt jede der Logging-Methoden eine Ausgabe, da die Logging-Ebene auf die niedrigste Priorität eingestellt ist. Im Beispiel ist die Einstellung der Einfachheit halber hardcodiert. In der Regel wird man die Einstellung der Logging-Ebene aus der Konfiguration lesen. Man könnte sogar eine Methode am Remote-Interface der Enterprise-Bean anbieten, um die Logging-Ebene von der Clientseite aus herauf- oder herabsetzen zu können. Durch das Herauf- oder Herabsetzen der Logging-Ebene kann die Anzahl der Logging-Ausgaben zur Laufzeit verändert werden. Die Enterprise-Bean könnte auch mit einer Logik versehen werden, die die Logging-Ebene beim Eintreten bestimmter Ereignisse automatisch herauf- oder heruntersetzt.

Um die Logging-Ausgaben in eine Datei zu schreiben, wird der Kategorie ein Objekt vom Typ *FileAppender* hinzugefügt. So genannte Appender sind Objekte, die verschiedene Ausgabekanäle repräsentieren. Eine Kategorie kann mehrere Appender haben. Die Appender können dynamisch zu einer Kategorie hinzugefügt oder gelöscht werden. Der Datei-Appender wird mit dem Dateinamen der Ausgabedatei und mit einem Objekt, welches das Layout der Ausgaben definiert, initialisiert. Für das Layout wird im Beispiel ein Standardlayout benutzt, das weiter unten am Beispiel einer Logging-Datei erläutert wird. Die Logging-Ausgaben aller Instanzen der Klasse *PartBean* werden damit in die gleiche Ausgabedatei mit dem Namen *PartBean.log* geschrieben.

Über die Verwendung des *FileAppender* kann man an dieser Stelle sicherlich diskutieren. Einer Enterprise-Bean ist es nämlich nicht erlaubt, auf das Dateisystem zuzugreifen (siehe Kapitel 3.8). Allerdings greift die Enterprise-Bean nicht direkt auf das Dateisystem zu, sondern indirekt über die *log4j*-Bibliothek. Sollten beim Schreiben der Datei Fehler auftreten, werden diese nicht an die Enterprise-Bean weitergereicht. *Log4j* fängt die Fehler ab und behandelt sie intern. Die Wahrscheinlichkeit, dass die Enterprise-Bean beim Schreiben des Logging-Eintrages blockiert wird (z. B. wegen Synchronisationsproblemen mit dem Dateisystem) ist ebenfalls sehr gering, da die Ausgabe ins Dateisystem durch *log4j* gepuffert erfolgt. Proprietäre Logging-Dienste eines Applikationsservers verwenden in der Regel auch das Dateisystem als Ablage für Logging-Ausgaben. Sollten dennoch Bedenken gegen den Einsatz des *FileAppender* bestehen, kann als Alternative z. B. der *SocketAppender* benutzt werden, der die Logging-Ausgaben über eine Netzwerkverbindung an einen anderen Server sendet, wo sie dann gespeichert werden. Eine weitere Möglichkeit ist die Entwicklung eines eigenen Appenders, der die Logging-Ausgaben beispielsweise in die Datenbank schreibt. Die notwendigen Details für alternative Appender können der *log4j*-Dokumentation entnommen werden (vgl. [LOG4J].

Als letzte Aktion im Konstruktor der *PartBean* erfolgt eine Logging-Ausgabe mit der Priorität *INFO* über die erfolgreiche Initialisierung der Instanz. In den übrigen Methoden wird die im Konstruktor erzeugte Kategorie dazu verwendet, Logging-Ausgaben mit verschiedener Priorität über den Programmablauf und den internen Zustand der Komponente zu machen.

Da es im Deployment-Deskriptor keine Stelle gibt, an der auf die Verwendung von Bibliotheken von Drittanbietern hingewiesen werden kann, empfiehlt es sich, in der Beschreibung der Enterprise-Bean auf die Verwendung von *log4j* hinzuweisen. Da der Deployment-Deskriptor die zentrale Dokumentationsinstanz der Komponente ist, sollte diese Information auf jeden Fall dort untergebracht werden. Sinnvoll ist es auch, die Quelle zu nennen, von der der Deployer (der für die Installation der Enterprise-Bean verantwortlich ist) die Bibliothek beziehen kann. Hinweise zur Installation der Bibliothek sind ebenfalls hilfreich. Im Fall von *log4j* muss lediglich die Archivdatei *log4j.jar* im *Classpath* des EJB-Containers zu finden sein. Alternativ kann *log4j.jar* mit in das jeweilige *ejb.jar* gepackt werden.

Zur Veranschaulichung wurde mit Hilfe des Blackbox-Tests für die Part-Bean aus dem vorhergehenden Abschnitt eine Logging-Datei erzeugt. Listing 9.29 zeigt den Inhalt dieser Datei. Die angedeuteten Kommentare beschreiben die Zuordnung der Logging-Ausgaben zu den Methoden und Aktionen des Testfalls.

```
//TestPartBean.testCreate
//Erzeugen Part 11111
631 [ExecuteThread-14] INFO PartBean  -
    initialized instance succesfully
631 [ExecuteThread-14] DEBUG PartBean  -
    setEntityContext EntityEJBContextImpl@795cce
631 [ExecuteThread-14] INFO PartBean  -
    entering ejbCreate (11111)
631 [ExecuteThread-14] DEBUG PartBean  -
    Created new part with part-no:11111
641 [ExecuteThread-14] INFO PartBean  - leaving ejbCreate
671 [ExecuteThread-14] DEBUG PartBean  -
    ejbPostCreate (11111)
671 [ExecuteThread-14] DEBUG PartBean  - ejbStore
//TestPartBean.testDelete
//Erzeugen Part 22222
731 [ExecuteThread-10] INFO PartBean  -
    initialized instance succesfully
731 [ExecuteThread-10] DEBUG PartBean  -
    setEntityContext EntityEJBContextImpl@5507d3
731 [ExecuteThread-10] INFO PartBean  -
    entering ejbCreate (22222)
731 [ExecuteThread-10] DEBUG PartBean  -
    Created new part with part-no:22222
731 [ExecuteThread-10] INFO PartBean  - leaving ejbCreate
761 [ExecuteThread-10] DEBUG PartBean  -
    ejbPostCreate (22222)
761 [ExecuteThread-10] DEBUG PartBean  - ejbStore
//Löschen Part 22222
781 [ExecuteThread-14] INFO PartBean  - entering ejbLoad
811 [ExecuteThread-14] DEBUG PartBean  -
    data successfully loaded:
```

```
    [[TsPartDetails]partNumber=22222;partDescription
    =null;supplierName=null;price=0.0;]
811 [ExecuteThread-14] INFO PartBean  - leaving ejbLoad
811 [ExecuteThread-14] DEBUG PartBean  - ejbRemove
//TestPartBean.testUpdate
//Erzeugen Part 33333
901 [ExecuteThread-14] INFO PartBean  -
    initialized instance succesfully
901 [ExecuteThread-14] DEBUG PartBean  -
    setEntityContext EntityEJBContextImpl@79781
901 [ExecuteThread-14] INFO PartBean  -
    entering ejbCreate (33333)
901 [ExecuteThread-14] DEBUG PartBean  -
    Created new part with part-no:33333
901 [ExecuteThread-14] INFO PartBean  - leaving ejbCreate
911 [ExecuteThread-14] DEBUG PartBean  -
    ejbPostCreate (33333)
911 [ExecuteThread-14] DEBUG PartBean  - ejbStore
//Aufruf von PartBean.getPartDetails
921 [ExecuteThread-10] INFO PartBean  - entering ejbLoad
921 [ExecuteThread-10] DEBUG PartBean  -
    data successfully loaded:
    [[TsPartDetails]partNumber=33333;partDescription=
    null;supplierName=null;price=0.0;]
921 [ExecuteThread-10] INFO PartBean  - leaving ejbLoad
921 [ExecuteThread-10] DEBUG PartBean  - getPartDetails :
    [[TsPartDetails]partNumber=33333;partDescription=
    null;supplierName=null;price=0.0;]
921 [ExecuteThread-10] DEBUG PartBean  - ejbStore
//Aufruf von PartBean.setPartDetails
941 [ExecuteThread-14] INFO PartBean  - entering ejbLoad
941 [ExecuteThread-14] DEBUG PartBean  -
    data successfully loaded:
    [[TsPartDetails]partNumber=33333;partDescription=
    null;supplierName=null;price=0.0;]
941 [ExecuteThread-14] INFO PartBean  - leaving ejbLoad
941 [ExecuteThread-14] INFO PartBean  -
    entering setPartDetails
941 [ExecuteThread-14] DEBUG PartBean  - part-data updated
    [[TsPartDetails]partNumber=33333;partDescription=
    Test Part;supplierName=Test Supplier;price=120.0;]
941 [ExecuteThread-14] INFO PartBean  -
    leaving setPartDetails
941 [ExecuteThread-14] DEBUG PartBean  - ejbStore
//Aufruf von PartBean.setPartDetails mit veraltetem
//TsPartDetails-Objekt
1001 [ExecuteThread-10] INFO PartBean  - entering ejbLoad
1001 [ExecuteThread-10] DEBUG PartBean  -
    data successfully loaded:
    [[TsPartDetails]partNumber=33333;partDescription=
    Test Part;supplierName=Test Supplier;price=120.0;]
```

```
1001 [ExecuteThread-10] INFO PartBean  - leaving ejbLoad
1001 [ExecuteThread-10] INFO PartBean  -
     entering setPartDetails
1001 [ExecuteThread-10] WARN PartBean  -
     out of date part-details-object
1001 [ExecuteThread-10] INFO PartBean  -
     leaving setPartDetails
1001 [ExecuteThread-10] DEBUG PartBean  - ejbStore
//TestPartBean.finalizeTest
//Löschen Part 11111
1011 [ExecuteThread-10] INFO PartBean  - entering ejbLoad
1011 [ExecuteThread-10] DEBUG PartBean  -
     data successfully loaded:
     [[TsPartDetails]partNumber=11111;partDescription=
     null;supplierName=null;price=0.0;]
1011 [ExecuteThread-10] INFO PartBean  - leaving ejbLoad
1011 [ExecuteThread-10] DEBUG PartBean  - ejbRemove
//Löschen Part 33333
1021 [ExecuteThread-14] INFO PartBean  - entering ejbLoad
1031 [ExecuteThread-14] DEBUG PartBean  -
     data successfully loaded:
     [[TsPartDetails]partNumber=33333;partDescription=
     Test Part;supplierName=Test Supplier;price=120.0;]
1031 [ExecuteThread-14] INFO PartBean  - leaving ejbLoad
1031 [ExecuteThread-14] DEBUG PartBean  - ejbRemove
```

Listing 9.29: Logging-Ausgabe der Klasse PartBean

Das Format der Logging-Ausgabe kann frei bestimmt werden. Die Part-Bean benutzt ein vorgegebenes Format. In der ersten Spalte wird die seit dem Start der Anwendung vergangene Zeit in Millisekunden ausgegeben. Die zweite Spalte zeigt den Namen des Threads an, der die Anfrage des Clients bearbeitet hat. In der dritten Spalte wird die Ebene des Logging (*INFO*, *DEBUG*, *WARN*, *ERROR* oder *FATAL*) und in der vierten Spalte der Name der Logging-Kategorie (im Fall der Part-Bean ist der Name *PartBean*) ausgegeben. Abschließend wird der Text ausgegeben, der in der jeweiligen Logging-Methode übergeben wird.

Aus dem in Listing 9.29 gezeigten Beispiel geht hervor, dass über Logging-Ausgaben der Ablauf innerhalb einer Enterprise-Bean verfolgt werden kann. Tritt ein Fehler auf, kann der Fehler sofort lokalisiert werden. Fehler können entweder durch die Logging-Ebene WARN, ERROR oder FATAL bzw. durch eine in der Logging-Datei protokollierte Ausnahme entdeckt werden. Wird der interne Zustand der Komponente und der Programmablauf im Vorfeld mitprotokolliert, kann die Fehlerquelle relativ schnell gefunden werden. In unserem Beispiel kann man anhand des WARN-Eintrags sofort sehen, dass ein Client versucht hat, die Daten der Enterprise-Bean mit einem ungültigen Details-Objekt zu aktualisieren. Die Voraussetzung dafür ist, dass der Entwickler

der Komponente entsprechend ausssagefähige Logging-Ausgaben programmiert. Tritt z.B. eine Ausnahme auf, die im Code zwar gefangen, in den Logging-Ausgaben aber nicht protokolliert wird, kann der Fehler durch Logging nicht gefunden werden.

Mit Logging kann auch das Verhalten des EJB-Containers studiert werden. Aus den gewonnenen Erkenntnissen können Maßnahmen für Optimierungen der Komponenten gewonnen werden. Dabei sollte man jedoch keine Abhängigkeiten zu einem bestimmten EJB-Container eingehen.

Whitebox-Tests könnte man bei Enterprise-Beans durchführen, indem man die durch die Blackbox-Tests erzeugten Log-Dateien analysiert. Die Analyse von Log-Dateien erfordert relativ hohen Programmieraufwand. Außerdem muss eine Vorgehensweise für Logging-Ausgaben definiert und auch eingehalten werden. Da dieser Aufwand aber nur einmalig anfällt und dadurch die Qualität der Tests wesentlich verbessert wird, kann er sich durchaus lohnen. Die Einträge der Logging-Datei aus Listing 9.29 beispielsweise sind sehr homogen und würden sich gut für eine lexikalische Analyse eignen. Der Klasse *EJBTest* könnte z.B. über die Konfigurationsdatei bekannt gemacht werden, in welchem Verzeichnis die Log-Dateien der Enterprise-Beans abgelegt werden und welche Log-Datei zu welchem Testfall gehört. Außerdem könnte die Klasse *EJBTestCase* um die Fähigkeit erweitert werden, diese Log-Dateien nach einer bestimmten Vorgehensweise zu analysieren (Parsen). Man könnte sogar so weit gehen, dass man Objekte aus den Einträgen in der Logging-Datei rekonstruiert und der abgeleiteten Klasse zur Analyse übergibt. Zum Beispiel dürfte es die Information *[[TsPartDetails] partNumber = 33333; partDescription = Test Part; supplierName = Test Supplier; price = 120.0;]* aus der Logging-Datei ohne weiteres zulassen, das Objekt vom Typ *TsPartDetails* zu rekonstruieren. Die abgeleitete Klasse könnte dann für den jeweiligen Testfall die Auswertung vornehmen und beurteilen, ob der durch die Log-Datei während des Blackbox-Tests aufgezeichnete interne Zustand der Komponente korrekt war.

Interessant ist es auch, mittels Blackbox- und Whitebox-Tests zu untersuchen, ob sich eine Komponente in verschiedenen Containern gleich verhält bzw. ob das Verhalten des EJB-Containers dem in der Spezifikation beschriebenen Verhalten entspricht. Indirekt hat man so auch die Möglichkeit, den EJB-Container zu testen.

Automatische Tests und Logging sind Mechanismen, die die Qualität einer Komponente tendenziell verbessern. Sind die Tests an sich fehlerhaft oder nachlässig programmiert oder sind die Logging-Ausgaben lückenhaft oder wenig aussagekräftig, ist der Beitrag zu einer besseren Qualität der Komponente fragwürdig. Um Tests und Logging zu einem sinnvollen Instrument zu machen, müssen die Entwickler der Komponenten und der Tests viel Disziplin und Sorgfalt an den Tag legen.

Literaturverzeichnis

[Griffel, 1998] Griffel, Frank: Componentware – Konzepte und Techniken eines Softwareparadigmas. dpunkt-Verlag, 1998.

[Szyperski, 1998] Szyperski, Clemens: Component Software. Addison-Wesley Longman, 1998.

[Orfali et al., 1996] Orfali, R., Harkey, D.: Client / Server Programming with Java and CORBA. John Whiley & Sons, 1996.

[Oaks., 1998] Oaks, Scott: Java Security, O'Reilly & Associates, 1998.

[I-Kinetics, 1997] ComponentWare®: Component Software for the Enterprise, April 1997. *www.i-kinetics.com/wp/cwvision/CWVision.html*.

[Sun Microsystems, 1997] Java Beans Version 1.01, 1997. *http://www.javasoft.com/beans/spec.html*.

[Sun Microsystems, 1999] Enterprise JavaBeans Version 1.1 Public Release, 1999. *http://www.javasoft.com/ejb/spec.html*

[Sun Microsystems, 2001] Enterprise JavaBeans Version 2.0 Public Release, 2001. *http://www.javasoft.com/ejb/spec.html*.

[EJB Developer Guide, 1999] Enterprise Java Beans Developers Guide, Beta-Version, September 1999, Sun Microsystems.

[J2EE, 2000] The Java 2 Platform, Enterprise Edition (J2EE), *http://java.sun.com/j2ee.*

[J2EE-APM, 1999] The J2EE Application Programming Model, Version 1.0, September 1999, Sun Microsystems.

[JMS, 1999] Java Messaging Service, Version 1.0.2, November 1999, Sun Microsystems.

[Gamma et al., 1996] Gamma, Erich, Helm, Richard, Johnson, Ralph, Vlisides John: Designpatterns, Addison-Wesley Longman, 1996.

[IBM, 1997] International Technical Support Organisation (ITSO): Factoring Java Beans in the Enterprise, Dezember 1997, IBM Corporation.

[IBM, 1999] International Technical Support Organisation (ITSO): Developing Enterprise Java Beans with Visual Age for Java, März 1999, IBM Corporation.

[IBM, 2000] International Technical Support Organisation (ITSO): Design and Implement Servlets, JSPs, and EJBs for IBM WebSphere Application Server, August 2000, IBM Coorporation

[Roßbach et al., 1999] Roßbach, Peter, Schreiber, Hendrik: Java Server und Servlets. Addison-Wesley Longman, 1999.

[Behme et al., 1998] Behme, Henning, Mintert, Stefan: XML in der Praxis. Addison-Wesley Longman, 1998.

[Monson-Haefel, 1999] Monson-Haefel, Richard: Enterprise JavaBeans. O'Reilly & Associates, 1999.

[JDBC Sun Microsystems, 1997] Sun Microsystems Inc., 1998, JDBC 2.0 API, *http://www.javasoft.com/products/jdbc*

[Sun Microsystems, JTA, 1999] Sun Microsystems Inc., Java Transaction API (JTA), Sun Microsystems Inc. 1999, Copyright © 1997-1999 by Sun Microsystems, 901 San Antonio Rd., Palo Alto, CA 94303

[Sun Microsystems, JTS, 1999] Sun Microsystems Inc., Java Transaction Service (JTS), Sun Microsystems Inc. 1999, Copyright © 1997-1999 by Sun Microsystems, 901 San Antonio Rd., Palo Alto, CA 94303

[Lang 1995] Lang, Lockemann: Datenbankeinsatz. Springer-Verlag 1995.

[Valesky 1999] Valesky, Tomas: Enterprise JavaBeans, Addison Wesley Longman, 1999.

[Kredel et al., 1999] Kredel, Heinz, Yoshida, Akitoshi: Thread- und Netzwerkprogrammierung mit Java. dpunkt-Verlag, 1999.

[Eckel, 1998] Eckel, Bruce: Thinking in Java. Prentice-Hall PTR, 1998.

[APACHE] *http://www.apache.org*

[LOG4J] *http://www.apache.org/projects/log4j*

[WEBLOGIC] *http://www.weblogic.com*

[Jamie/Perone 2000] Jamie Jaworski, Paul Perrone, Java Security Handbook, Sams

[Scott 1999] Scott Oaks, Java Security, O'Reilly & Associates

[JCE] Java Cryptography Extendsion, Sun Microsystems, Seite: 2
http://www.java.sun.com/products/jce/index.html

[JAAS] Java Authentication and Authorization Service, Sun Microsystems, Seite: 2
http://java.sun.com/products/jaas/

[JSSE] Java Secure Socket Extension,Seite: 3
Sun Microsystems, *http://www.developer.java.sun.com/developer/earlyAccess/jsse/index.html*

Index

E

T