Enterprise JavaBeans: Architektur, Konzepte und Implementation

Christoph Korn (mi7790)

31. März 2001

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Über diesen Text

Dieser Text ist im Rahmen einer Studienarbeit im Fach Medieninformatik an der Fachhochschule Wedel im Wintersemester 2000/2001 entstanden. Diese Studienarbeit stellt die architektonischen Konzepte von Enterprise JavaBeans vor, verdeutlicht diese anhand einer einfachen Beispielanwendung und führt die Vor- und Nachteile von EJB-basierten Anwendungen auf.

Die vorliegende Arbeit basiert auf der derzeit aktuellen Enterprise JavaBeans Spezifikation v.1.1 von Sun Microsystems ([SUN99]). Die Spezifikation 2.0 ([SUN00]) findet in dieser Arbeit nur am Rande Beachtung, da sie zurzeit noch nicht endgültig verabschiedet ist. Die wesendlichen Änderungen zum vorläufigen Entwurf der Version 2.0 sind in Kapitel 7.1 zusammengefasst.

1.2 Überblick über Enterprise JavaBeans

1.2.1 Definition

Sun Microsystems definiert Enterprise JavaBeans (EJBs) folgendermaßen:

The Enterprise JavaBeans architecture is a component architecture for the development and deployment of component-based distributed business applications. Applications written using the Enterprise JavaBeans architecture are scalable, transactional, and multi-user secure. These applications may be written once, and then deployed on any server platform that supports the Enterprise JavaBeans specification. [SUN99]

Sun sieht Enterprise JavaBeans also als Technologie um unternehmenskritische,

komponentenbasierte verteilte Client/Server-Anwendungen, umzusetzen. Ein starkes Gewicht wird hierbei auf die Portierbarkeit[1] dieser von Sun *Beans* genannten Komponenten gelegt.

Die Enterprise JavaBean-Architektur ist von Sun Microsystems lediglich spezifiziert, und Schnittstellen sowie Aufgaben einzelner Komponenten definiert worden. Es stellt ähnlich wie Java Servlets, Java Server Pages etc. nur einen Standard und kein Produkt dar. Die Umsetzung dieses Standards wird von den Herstellern von Java 2 Enterprise Edition (J2EE) konformen Servern gefordert.

1.2.2 Enterprise JavaBeans als Teil von J2EE

Enterprise JavaBeans ist Teil der Java 2 Enterprise Edition von Sun Microsystems. Diese stellt eine Erweiterung Javas (Java Standard Edition) um Technologien und Schnittstellen zu Diensten dar, die für die Entwicklung von Unternehmensanwendungen benötigt werden. J2EE umfasst neben Enterprise Java Beans z.B. folgende Technologien: Java Database Connectivity (JDBC), Java Servlets, Java Server Pages (JSP), Java Messaging Service (JMS), Transaktionen (JTA und JTS), Java Naming and Directory Interface (JNDI).

1.2.3 Enterprise JavaBeans als Middleware-Technologie

In einer Drei-Schichten-Architektur bzw. Mehr-Schichten-Architektur, wie sie derzeit aus softwaretechnischer Sicht für Client/Server-Anwendungen zu favorisieren ist, positioniert sich die Enterprise JavaBean-Technologie im Bereich der Middleware-Anwendungen. Diese Schicht soll nach dem Drei- bzw. Mehr-Schichten-Modell die Anwendungslogik kapseln. Die Datenhaltung wird auf unteren Schichten meist von Datenbankmanagementsystemen geleistet, während die Darstellung und das Benutzerinterface von Clientanwendungen auf übergeordneten Schichten übernommen werden. Abbildung 1.1 verdeutlicht das Konzept einer Mehr-Schichten-Anwendung.

1.2.4 Zweck von Enterprise JavaBeans

Durch Enterprise JavaBeans hat Sun Microsystems ein Framework für ein Komponentenmodell entwickelt, das es Entwicklern erlaubt, Softwarekomponenten zu erstellen, die sich auf eine Anwendungslogik konzentrieren und möglichst stark von konkreten Einsatzgebieten und -umgebungen getrennt sind. So ist es möglich, Geschäftsprozesse in diesen Komponenten abzubilden, die sich in unterschiedlichen Umgebungen einsetzen lassen. Die Entwickler einer EJB-basierten Anwendung können sich so auf ihre Spezialgebiete (z.B. die Programmierung einer spezifischen Anwendungslogik)

[1] 'Write Once Run Everywhere'

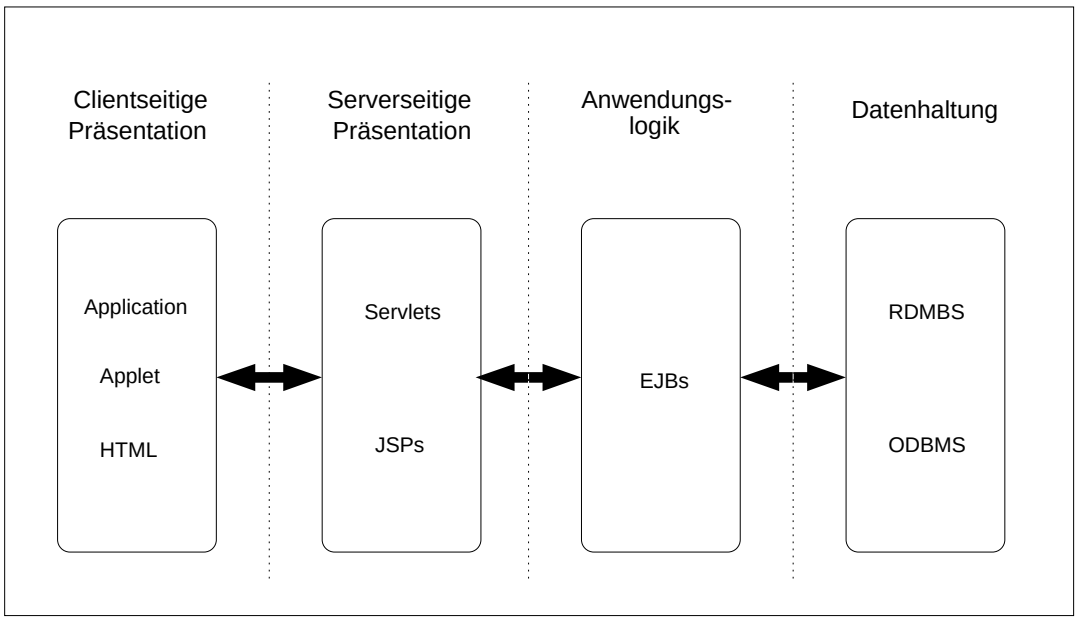

Abbildung 1.1: J2EE-Mehrschichten-Architektur

konzentrieren und sich auf definierte Schnittstellen z.B. zu Diensten oder anderen Komponenten verlassen.

Eine Beispielanwendung für EJBs wäre z.B. eine automatische Lagerverwaltung. Bei dieser Art von Anwendung gibt es eine Reihe von Komponenten, die konkrete Aufgaben abbilden müssen. Hierzu zählt z.B. eine Komponente, die bei sinkendem Lagerbestand neue Artikel bestellt. Eine andere Komponente verwaltet die Ausgabe von Artikeln und das Erstellen entsprechender Rechnungen. All diese Komponenten lassen sich in jeder Lagerverwaltung erkennen. Das konkrete Einsatzgebiet und die Umgebung (z.B. Sicherheitsrichtlinien, Größe und Art des Systems, zu Grunde liegende Datenbank etc.) ändern sich jedoch von Fall zu Fall. Die EJB-Spezifikation beschreibt eine Umgebung, die es erlaubt, solche Komponenten zu entwickeln und später in verschiedensten Einsatzgebieten zu benutzen.

Hierbei handelt es sich nicht um einen neuen Ansatz. Software sollte generell so konzipiert sein, dass sie leicht erweiterbar, skalierbar und ohne großen Aufwand in wechselnde Anwendungsumgebungen portierbar ist. Durch die Spezifikation von Enterprise JavaBeans wird jedoch ein Standard geschaffen, der konkrete Schnittstellen und eine Laufzeitumgebung für Komponenten schafft und erzwingt. Hierdurch wird erreicht, dass die Entwickler von Komponenten diese mit weitestgehend ähnlichen Schnittstellen und Anforderungen erstellen müssen. Die Benutzer dieser Komponenten können sich auf bestimmte Schnittstellen verlassen und haben so die Möglichkeit ggf. Komponenten verschiedenster Hersteller möglichst einfach gegeneinander auszutauschen oder miteinander zu kombinieren.

1.3 Einordnung in ein technologisches Umfeld

Zur Abgrenzung der Enterprise JavaBean-Architektur wird diese im folgenden kurz einigen ähnlichen Technologien gegenübergestellt.

1.3.1 Enterprise JavaBeans und CORBA

CORBA ist diejenige Technologie, die Enterprise JavaBeans am ähnlichsten ist. Beide Architekturen definieren ein Framework für Komponenten, mit dem sich verteilte objektorientierte Anwendungen erstellen lassen. Weiterhin bieten beide Technologien den Komponenten Dienste, wie z.B. einen Transaktionsmanager, einen Persistenzmanager oder einen Sicherheitsdienst über definierte Schnittstellen an, die es dem Entwickler ermöglichen, sich auf die eigentliche Anwendungslogik zu konzentrieren.

Der Hauptunterschied zwischen Enterprise JavaBeans und CORBA ist, dass CORBA nicht für eine bestimmte Programmiersprache definiert worden ist und es mit Hilfe einer programmiersprachen-unabhängigen Schnittstellenbeschreibungssprache (Interface Definition Language (IDL)) ermöglicht, Komponenten, die in verschiedenen Programmiersprachen entwickelt worden, in einer Anwendung zu kombinieren. CORBA wurde von einem Konsortium unterschiedlichster Firmen (Object Management Group) definiert und weiterentwickelt.

Enterprise JavaBeans hingegen ist von Sun Microsystems als Teil der J2EE und somit nur für Java definiert worden. Es ist jedoch möglich, Enterprise JavaBeans über CORBAs IIOP an einen CORBA-ORB anzubinden, und so als Objekte in einer CORBA-basierten Anwendung zu nutzen.

1.3.2 Enterprise JavaBeans und JavaBeans

Sun nennt Softwarekomponenten, die in Java geschrieben worden, *'Beans'*. Somit lassen die Namen *Enterprise Java Beans* und *JavaBeans* darauf schließen, dass beide Technologien einen Standard für Softwarekomponenten darstellen. Trotz der ähnlichen Namen, die oft zu Verwechselungen führen, bestehen jedoch grundlegende Unterschiede:

JavaBeans sind relativ kleine Softwarekomponenten, die meist ein graphisches Userinterface besitzen und sich visuell komponieren lassen. Eine JavaBean könnte z.B. ein Datei-Dialog sein, der sich in verschiedensten Umgebungen einsetzen lässt.

Enterprise Java Beans hingegen werden konzipiert, um Geschäftsanwendungen zu erstellen. Sie dürfen keine graphischen Benutzerinterfaces besitzen und kapseln meist komplexe Anwendungslogiken.

1.3.3 Enterprise JavaBeans und RMI

Java RMI (Remote Method Invocation) definiert eine Möglichkeit, verteilte objektorientierte Anwendungen zu erstellen. Methodenaufrufe an Objekte, die sich in einer anderen Virtual Machine (ggf. auf einem entfernten Server) befinden, werden von einem RMI-Stub-Objekt entgegen genommen und an ein RMI-Skeleton-Objekt auf einem über ein Netzwerk erreichbaren Server weiter geleitet um dort bearbeitet zu werden.

RMI stellt im Gegensatz zu Enterprise JavaBeans kein Framework für Komponenten dar. Die Enterprise JavaBean-Technologie benutzt RMI jedoch für die Kommunikation zwischen einem Client und den Enterprise Beans oder zwischen den Enterprise Beans untereinander.

Kapitel 2

Architektur

Dieses Kapitel stellt die Konzepte und die Architektur, auf der Enterprise JavaBeans basieren, kurz vor. In den späteren Kapiteln wird dann auf Details der einzelnen Komponenten und die Implementierung eingegangen. Alle in diesem und den folgenden Kapiteln gemachten Angaben basieren auf [SUN99]. Einen Überblick über die EJB-Architektur gibt Abbildung 2.1.

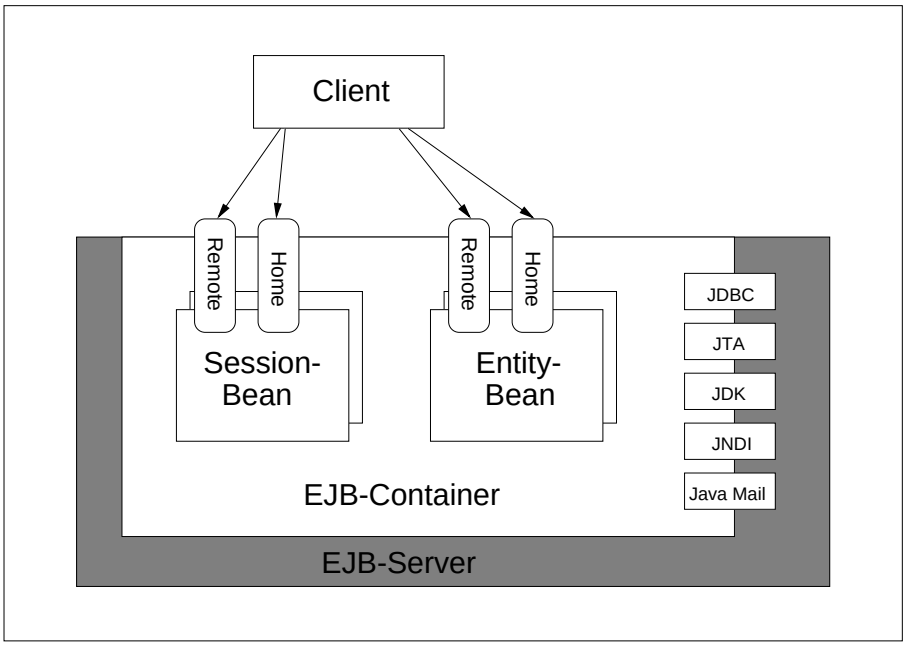

Abbildung 2.1: Überblick über die EJB-Architektur (aus [DENN00])

2.1 EJB Umgebung

Die EJB-Spezifikation sieht eine Laufzeitumgebung für die Enterprise Beans vor, die diesen Komponenten Dienste über definierte Schnittstellen zur Verfügung stellt

und die EJBs verwaltet. Zu dieser Laufzeitumgebung gehören ein J2EE-konformer Server und ein in diesen Server eingebetteter EJB-Container.

2.1.1 Der Server

Wie oben bereits erwähnt, hat Sun Microsystems Enterprise JavaBeans als Teil des J2EE-Standards konzipiert. Einem J2EE-konformen Server fällt die Aufgabe zu, eine Umgebung für mindestens einen EJB- und einen Servlet-Container zu schaffen und bestimmte Dienste (z.B. eine ServletEngine, eine JSP-Engine) zu implementieren. Für Anwendungen, die nur auf EJBs basieren, ist ein reiner EJB-Server (nur mit einem EJB-Container) ausreichend.

Der Server kapselt die betriebssystemnahen und -spezifischen Funktionen und stellt sie den Containern über geeignete Schnittstellen zur Verfügung.

Die Schnittstelle zwischen dem Server und den Containern wurde von Sun nicht spezifiziert, was es in der Praxis nahezu unmöglich macht verschiedene Container in unterschiedlichen Servern zu betreiben oder sogar den Container selbst zu entwickeln und in verschiedenen Servern einzusetzen. Aus diesem Grund sind der Hersteller des Servers und der des Containers fast immer identisch und im allgemeinen Sprachgebrauch wird oft nicht zwischen Server und Container unterschieden.

Weitere Aufgaben des Servers sind z.B. die Garantie von Ausfallsicherheit, ggf. die Lastverteilung zwischen mehreren Servern, das Bereitstellen eines Namens- und Verzeichnisdienstes, das Pooling von Ressourcen und das Thread- bzw. Prozessmanagement für den Einsatz mehrerer Container.

Ein J2EE-Server wird in der Praxis häufig in Kombination mit einem Webserver und anderen Diensten als Application-Server angeboten.

2.1.2 Der EJB-Container

Ein EJB-Container ist Teil des oben beschriebenen J2EE-konformen Servers und stellt die eigentliche Laufzeitumgebung für die EJBs dar. Zum einen bietet der Container den Beans verschiedene Dienste über definierte Schnittstellen an, zum anderen hat er eine Reihe von Aufgaben, wie z.B. die Kontrolle der Lebenszyklen der Beans, Zugriffskontrolle, Instanzenpooling. Die Dienste und Aufgaben werden im folgenden kurz erläutert.

Dienste und Schnittstellen des Containers

Die Dienste, die der Container den Beans zur Verfügung stellt, basieren auf den Technologien, die J2EE bietet. Die folgenden Schnittstellen zu Diensten müssen den

Beans von einem EJB 1.1 konformen Container mindestens angeboten werden:

- Das API des JDK 1.1.x bzw. der Java 2 Plattform
- Das EJB 1.1 API
- Das API des JNDI 1.2 (Java Naming and Directory Interface)
- Das JDBC 2.0 API (Java Database Connectivity)
- Das JTA 1.0.1 (Java Transaction API)
- Das Java Mail 1.1 API

Es steht den Entwicklern von EJB-Containern frei, weitere Dienste anzubieten. So implementieren bereits jetzt viele Anbieter von Containern das von der Version 2.0 der EJB-Spezifikation geforderte JMS-API (Java Messaging Service).

Die Nutzung von Diensten, die nicht als Muss von der EJB 1.1 Spezifikation vorgesehen sind, gilt jedoch als problematisch, da die Beans, die diese Dienste nutzen, auf einen bestimmten Container eines Herstellers angewiesen sind und so nicht problemlos in anderen Containern eingesetzt werden können.

Zugriffskontrolle und Sicherheit

Der Container bildet die Schnittstelle zwischen der Clientanwendung und den Beans. Der Zugriff eines Clients[1] auf eine Bean wird durch ihn kontrolliert.

Zur Lastverteilung kommen oft mehrere Container auf mehreren Systemen zum Einsatz. Da der Zugriff des Clients auf eine Bean mit Hilfe des JNDI gesteuert wird und so vom Container kontrolliert wird, kann eine Skalierung des Systems oder eine Lastverteilung durch ein Verschieben einer Bean in einen anderen Container ohne große Anpassungen erzielt werden. Lediglich im Namens- und Verzeichnisdienst muss in diesem Fall ein entsprechender Eintrag geändert werden. Hierdurch ist der Zugriff des Clients auf die Bean jederzeit transparent und die Bean ortsunabhängig.

Das Sicherheitsmanagement wird ebenso vom Container übernommen, der kontrolliert ob ein entsprechender Client auf bestimmte Methoden der Bean (oder die Bean selbst) Zugriff hat. Eine Bean selbst sollte keine Sicherheitsmechanismen implementieren, um möglichst in vielen Umgebungen mit wechselnden Sicherheitsstrategien eingesetzt werden zu können. Die Sicherheitsrichtlinien werden erst beim Zusammenstellen von mehreren Beans zu einer EJB-Anwendung vom Application Assembler (s. Kapitel 2.6.2) und später vom Systemadministrator festgelegt.

[1] Bei einem Client kann es sich sowohl um eine Clientanwendung, die ein Benutzerinterface bietet, als auch um eine andere Enterprise Bean handeln.

Weiterhin serialisiert der Container parallele Clientzugriffe auf die Methoden der Enterprise Beans, sodass diese keine Mechanismen zum Threadmanagement implementieren müssen.

Instanzenpooling und Kontrolle des Lebenszyklus

Da die Zugriffe des Clients immer über den Container geschehen, hat dieser die Möglichkeit Ressourcen zu sparen, indem er den Zugriff auf die EJBs für den Client vollkommen transparent optimiert. Diese Optimierungen umfassen zum einen ein Instanzenpooling, zum Anderen die Kontrolle des Lebenszyklus einer Bean.

Das Instanzenpooling basiert auf der Möglichkeit, die Identität einer Bean (ihre individuellen Daten) von ihrer Instanz (dem eigentlichen Objekt) zu lösen. D.h. der Container hält einen Pool von Beans ohne Identität. Will nun ein Client auf eine Bean zugreifen, so wird der Beaninstanz eine Identität zugewiesen, was bedeutet, dass erst jetzt die Bean mit ihren individuellen Daten initialisiert wird. Braucht der Client die Bean nicht mehr, wird nicht die Instanz gelöscht, sondern nur ihre Identität. Anschließend wird die Bean wieder dem Pool zu verwendender Beans zugeführt. Dieses Instanzenpooling schont Ressourcen, da es ein teures Erzeugen und Löschen von Objekten minimiert. Ein Instanzenpooling ist nur für Entity-Beans (siehe Kapitel 2.2.2) vorgesehen.

Die Kontrolle des Lebenszyklus umfasst das oben genannte Zuweisen einer Identität, das Löschen dieser Identität sowie die Auslagerung der Bean in einem permanenten Speicher. Dies ist nötig, da EJBs für unternehmenskritische Großanwendungen konzipiert sind, in der eine Vielzahl von Daten (Objekten) vorgehalten werden müssen. Es ist oft nicht möglich, alle Daten (Objekte) gleichzeitig im Arbeitsspeicher zu halten. Wird eine Bean längere Zeit nicht genutzt, kann der Container diese passivieren (in einen permanenten Speicher auslagern), um der Beaninstanz eine andere Identität zuzuweisen und ggf. zu einem späteren Zeitpunkt die Daten der Bean aus dem permanenten Speicher zu lesen sowie evtl. einer anderen Instanz zuzuweisen (aktivieren). Beim Passivieren einer Bean wird diese zunächst serialisiert und dann meist in einer Datenbank oder in einem Sekundärspeicher abgelegt. Die Strategie, nach der der Container die Lebenszyklen der Beans verwaltet, wird von Sun nicht vorgeschrieben und trägt wesendlich zur Performanz des Containers bei.

Persistenz

Im Fall der containergesteuerten Persistenz (siehe Kapitel 2.2.2) hat der Container weiterhin die Aufgabe die Beans automatisch in einem permanenten Speicher (z.B. eine relationale Datenbank) abzulegen, sodass diese über mehrere Systemneustarts erhalten bleiben. So werden die Beans persistent gehalten und ein Programmierer einer solchen Bean muss keinen Code schreiben, der die Daten einer Bean speichert, da diese Aufgabe der Container übernehmen muss.

Transaktionen

Im Fall der Transaktionssteuerung durch den Container hat dieser die Aufgabe, das Transaktionsmanagement zu erfüllen. Wie und ob der Container die Transaktionssteuerung übernehmen soll, wird ihm über den Deployment-Descriptor (siehe Kapitel 2.3.5) mitgeteilt. Mehr zum Thema Transaktionen findet sich in Kapitel 5.

2.2 Typen von EJBs

Die EJB-Spezifikation 1.1 sieht die folgenden zwei Arten von Enterprise JavaBeans vor.

- Session-Beans
- Entity-Beans

Session-Beans bilden eine Anwendungslogik ab, wo hingegen Entity-Beans die Anwendungsdaten darstellen. Im folgenden werden die beiden Bean-Typen kurz näher vorgestellt.

2.2.1 Session-Beans

Im Allgemeinen greift ein Client auf Methoden der Session-Beans zu, um Aktionen auf dem Server auszuführen. In den Methoden der Session-Beans sind also oft komplexe Operationen auf Daten auf dem Server zusammengefasst. Im Idealfall greifen Clients nur über die Methoden, die ihnen die Session-Beans bieten, auf die Daten auf dem Server (die durch die Entity-Beans abgebildet werden) zu. Session-Beans sind im Gegensatz zu Entity-Beans relativ kurzlebig, da sie immer nur genau einem Client zur Verfügung stehen und nur für die Dauer einer Clientsitzung existieren.

Eine Lagerverwaltung könnte beispielsweise als Session-Bean implementiert werden. Über die Methoden der Lagerverwaltungsbean würden Clients dann auf das Lager zugreifen können.

Arten von Session-Beans

Session-Beans lassen sich in

- zustandslose Session-Beans und
- zustandsbehaftete Session-Beans

unterteilen. Der Unterschied zwischen diesen beiden Arten von Session-Beans liegt darin, dass zustandsbehaftete Session-Beans ihren internen Zustand über mehrere Anfragen von demselben Client speichern. Der Client kann so den internen Zustand (bestimmte Daten) einer Session-Bean verändern und später auf diese Daten zugreifen. Dies ist u.a. nötig, um das Verhalten von Operationen abhängig von der vorangegangene Kommunikation mit dem Client zu steuern.

Zustandslose Session-Beans hingegen speichern keinen internen Zustand. Clients kommunizieren nur über Methodenaufrufe mit der Bean, die den internen Zustand der Bean nicht verändern. Hierdurch ist es in der Praxis möglich, dass sich mehrere Clients eine Instanz einer zustandslosen Session-Bean für sie vollkommen transparent 'teilen' können.

2.2.2 Entity-Beans

Wie oben bereits angedeutet sollen Entity-Beans die Daten einer Anwendung abbilden. Im obigen Beispiel einer Lagerhaltung könnten die Entity-Beans die einzelnen Produkte im Lager abbilden.

Entity-Beans sind im Gegensatz zu Session-Beans persistent, d.h. die Daten, die in diesen Beans gespeichert sind, bleiben über mehrere Clientsitzungen und Serverneustarts erhalten. Die Persistenz von Entity-Beans wird oft durch eine Speicherung in (zumeist relationalen) Datenbanken realisiert. Ein weiterer Unterschied zu Session-Beans ist der, dass dem Client niemals eine private Instanz einer Entity-Bean zur Verfügung steht. Die Entity-Beans und deren Daten stehen - sieht man einmal von den implementieren Sicherheitsrichtlinien ab - jedem Client zur Verfügung.

Erst seit Version 1.1 der EJB-Spezifikation müssen Container zwingend Entity-Beans unterstützen. Zuvor galten Entity-Beans als optionaler Bestandteil der EJB-Technologie.

Typen von Entity-Beans

Entity-Beans lassen sich bezüglich der Art, wie die Daten der Beans persistent gehalten werden, unterteilen. Man unterscheidet folgende beiden Arten von Entity-Beans:

- Entity-Beans mit Bean Managed Persistence (BMP)
- Entity-Beans mit Contianer Managed Persistence (CMP)

Entity-Beans mit BMP obliegt selbst die Aufgabe ihre Daten in einen permanenten Speicher abzulegen. In diesem Falle werden sie vom Container lediglich durch einen Methodenaufruf aufgefordert, ihre Daten zu sichern, bevor z.B. diese Entity-Bean passiviert wird.

Im Falle von CMP hat der Container die Aufgabe, die Daten, die in der Entity-Bean gespeichert sind, in z.B. einer relationalen Datenbank abzulegen. Dieses Vorgehen bietet gegenüber der BMP den Vorteil der völligen Unabhängigkeit der Bean von ihrer Umgebung und dem entsprechenden Speicher. In der Praxis ist CMP aber oft nur für Beans geeignet, die eine eins-zu-eins Abbildung einer Datenbanktabelle darstellen, da die meisten EJB-Container nicht über entsprechend komplexe Mechanismen für objektrelationales Datenbankmapping verfügen, um Relationen abbilden zu können.

2.3 Bestandteile einer EJB

Eine Enterprise JavaBean setzt sich aus mehreren Dateien zusammen, deren Inhalt und Funktionen im folgenden vorgestellt werden sollen. Alle diese Bestandteile werden in einem JAR-File zusammengefasst und so in dem EJB-Container installiert (Deployment). Die Bestandteile einer EJB sind:

- das Home-Interface
- das Remote-Interface
- die Bean-Klasse
- der Deployment-Descriptor
- die Primärschlüssel-Klasse (nur bei Entity-Beans)

2.3.1 Das Home-Interface

Das Home-Interface einer Enterprise JavaBean deklariert alle Methoden die den Lebenszyklus einer Bean betreffen und die der Client zum Erzeugen, Löschen und Finden einer Bean aufrufen kann. Das Home-Interface muss von *javax.ejb.EJBHome* abgeleitet sein. Das Interface *javax.ejb.EJBHome* hat folgenden Aufbau:

```
public interface javax.ejb.EJBHome extends java.rmi.Remote {

   public EJBMetaData getEJBMetaData()
      throws java.rmi.RemoteException;

   public HomeHandle getHomeHandle()
      throws java.rmi.RemoteException;

   public void remove(Handle handle)
      throws java.rmi.RemoteException,
```

```
        javax.ejb.RemoveException;

    public void remove(Object primaryKey)
        throws java.rmi.RemoteException,
        javax.ejb.RemoveException;

}
```

Das Interface EJBHome und somit auch jedes Home-Interface ist von *java.rmi.Remote* abgeleitet und somit ein Interface eines RMI-Remote-Objektes, auf das über ein Netzwerk per RMI zugegriffen werden kann.

Die Methode *getEJBMetaData()* liefert ein Objekt vom Typ *EJBMetaData* zurück, das Informationen über die Bean selbst enthält. Z.B. enthält das EJBMetaData-Objekt Daten über den Typ der Bean (Session-Bean oder Entity-Bean). In der Praxis ist diese Information meist nur für die Entwickler von Container-Tools von Interesse.

Mit Hilfe der Methode *getHomeHandle()* hat der Client Zugriff auf ein Handle, das das Home-Objekt (s. Kapitel 2.4) der Bean eindeutig identdefiziert (auch im Falle von Session-Beans), sodass er dieses speichern und zu einem späteren Zeitpunkt wider auf die Bean zugreifen kann.

Die Methoden *remove()* löscht ein EJB-Objekt. Wie oben bereits erwähnt, bewirkt dies z.T. lediglich ein Löschen der Identitat, nicht aber der Instanz einer Bean. Die Methode *remove()* kann auf zwei Arten aufgerufen werden. Zum einen mit Hilfe eines Handles auf die Bean als Parameter, zum anderen mit einem PrimaryKey-Objekt. Der Aufruf mittels dieses Primärschlüssel-Objektes macht nur für Entity-Beans Sinn, da nur diese Beans einen Primärschlüssel besitzen.

Jedes Home-Interface muss weiterhin mindestens eine *create()-Methode* deklarieren. Mit Hilfe der *create()-Methode* wird es einem Client ermöglicht, eine Bean zu erzeugen. Es kann mehrere solcher Erzeugungsmethoden mit unterschiedlichen Signaturen geben, die anhand der übergebenen Parameter eine Bean unterschiedlich initialisieren. Da die Signatur der *create()-Methode* nicht bekannt und beanspezifisch ist, wird sie nicht im Interface *EJBHome* deklariert. Diese Methode liefet eine Referenz auf ein Remote-Objekt (s. Kapitel 2.4) zurück.

Ebenso wenig im Interface *EJBHome* deklariert ist die Methode *findByPrimaryKey()*, die mit Hilfe eines Primarschlüssel-Objektes eine Instanz einer Bean findet und zurückliefert.

2.3.2 Das Remote-Interface

Das Remote-Interface definiert die Methoden einer Bean, die ihre eigentliche Funktionalität abbilden. Für einen Client bietet das Remote-Interface die einzige Möglich-

keit, auf die Methoden (und somit die Anwendungslogik) der Bean zuzugreifen.

Jedes Remote-Interface muss das Interface *javax.ejb.EJBObject* erweitern, das wiederum wie das Interface *EJBHome* von *java.rmi.Remote* abgeleitet ist und somit ein Interfaces eines RMI-Remote-Objektes darstellt. Das Interface *EJBObject* hat folgenden Aufbau:

```
public interface javax.ejb.EJBObject extends java.rmi.Remote {

   public javax.ejb.EJBHome getEJBHome()
      throws java.rmi.RemoteException;

   public java.lang.Object getPrimaryKey()
      throws java.rmi.RemoteException;

   public void remove()
      throws java.rmi.RemoteException,
      javax.ejb.RemoveException;

   public javax.ejb.Handle getHandle()
      throws java.rmi.RemoteException;

   public boolean isIdentical(javax.ejb.EJBObject)
      throws java.rmi.RemoteException;

}
```

Die Methode *getEJBHome()* liefert das Home-Objekt (siehe Kapitel 2.4) der Bean zurück.

Über die Methode *getPrimaryKey()* bekommt der Client Zugriff auf das Primärschlüssel-Objekt einer (Entity-) Bean. Bei Session-Beans wird diese Methode leer implementiert und führt bei Aufruf zu einer RemoteException.

Die Methode *remove()* löscht eine Bean.

Die Methode *getHandle()* liefert ein Handle auf ein Remote-Objekt (s. Kapitel 2.4), analog zu der entsprechenden Methode des Home-Interfaces.

Die Methode *IsIdentical()* vergleicht zwei EJB-Objekte. Im Fall von zwei Referenzen auf zwei zustandslose Session-Beans der selben Klasse, liefert die Methode *true*.

In dem Remote-Interface müssen nur noch die Methoden, die die eigentliche Anwendungslogik der Bean widerspiegeln deklariert werden um dann in der Bean-Klasse implementiert zu werden.

2.3.3 Die Bean-Klasse

In der Bean-Klasse werden alle Methoden, die im Home- und Remote-Interface deklariert wurden, implementiert. Abhängig vom Bean-Typ und dessen Ausprägung werden jedoch ggf. manche Methoden leer oder nicht implementiert, da der Container automatisch eine entsprechende Funktionalität bereit stellt. Dies ist z.B. bei Entity-Beans mit CMP bei den Methoden, die die Identitätenverwaltung und die Persistenz betreffen, sowie bei den Finder-Methoden der Fall.

Die Bean-Klasse implementiert in Abhängigkeit ihres EJB-Typs nur das Interface *javax.ejb.EntityBean* (Entity-Beans) bzw. das Interface *javax.ejb.SessionBean* (Session-Beans). Diese Interfaces sind von dem Interface *javax.ejb.EnterpriseBean* und dieses wiederum von *java.io.Serializable* abgeleitet, wodurch ein Passivieren und Aktivieren der Bean ermöglicht wird.

Das Home-Interface und das Remote-Interface werden von der Bean-Klasse nicht implementiert. Es müssen jedoch alle Methoden dieser Interfaces mit ihrer exakten Signatur von der Bean-Klasse implementiert werden. Der Grund für dieses Vorgehen wird in Kapitel 2.4 erläutert.

2.3.4 Die Primärschlüssel-Klasse

Eine Primärschlüssel-Klasse wird nur für Entity-Beans benötigt. Über ein Objekt dieser Klasse lässt sich eine Entity-Bean eindeutig identdefizieren und suchen.

2.3.5 Der Deployment-Descriptor

Der Deployment-Descriptor ist eine XML-Datei, die Informationen über die Bean bzw. mehrere Beans enthält und wie diese Bean einzusetzen sind. Weiterhin enthält er Informationen wie mehrere Beans zu einer Anwendung zu kombinieren sind. Ein Deployment-Descriptor wird von dem Entwickler einer Bean bzw. einer EJB-Anwendung erzeugt und liefert dem Container und dem Bean-Deployer (siehe Kapitel 2.6.3) Informationen, wie die Bean zu behandeln ist. Im Deployment-Descriptor werden unter anderem folgende Merkmale einer Bean bzw. der EJB-Anwendung beschrieben:

- der JNDI-Name der Bean und eine Beschreibung ihrer Funktionalität
- Abhängigkeiten der Bean von Ressourcen und anderen Beans
- der Bean-Typ (Session-Bean oder Entity-Bean)
- die Klassen des Home- und Remote-Interfaces

- das Transaktionsmanagement
- die Art der Persistenz (BMP oder CMP (nur bei Entity-Beans))
- die Sicherheitsanforderungen

Die Aufgaben des Deployment-Descriptors sind in [SUN99] definiert. Die DTD findet sich unter http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd .

Weiterhin können im Deployment-Descriptor Konfigurationsparameter für die Bean festgelegt werden, die ihr mit Hilfe des Namens- und Verzeichnisdienstes zur Verfügung gestellt werden. Hierdurch ist es möglich, eine Bean ohne Editieren eines weiteren Konfigurationsfiles zu parametrisieren und an eine bestimmte Umgebung anzupassen.

2.4 Home-Objekt und Remote-Objekt

2.4.1 Aufgabe des Home- und Remote-Objekts

Damit der Container seine oben genannten Aufgaben erfüllen und den Zugriff auf die Beans kontrollieren kann, erstellt er bei Installation der Beans im Container zwei Proxy-Objekte[2], die die Clientaufrufe entgegen nehmen und unter Kontrolle des Containers an die Bean weiterleiten. Die Methoden einer Bean werden also niemals direkt von einem Client aufgerufen.

Der Container erstellt die folgenden zwei Objekte, die die Aufrufe der Bean-Methoden kapseln:

- das Home-Objekt
- das Remote-Objekt[3]

Diese Objekte werden vom Container automatisch mit Hilfe des Home- bzw. des Remote-Interfaces erzeugt und implementieren das jeweilige Interface.

Das erklärt, warum die Bean-Klasse weder das Home-, noch das Remote-Interface erweitern darf [4] (im Sinne des Java-Schlüsselwortes *implements*), aber die Methoden, die in diesen beiden Interfaces deklariert sind, von der Bean-Klasse trotzdem implementiert werden müssen (im Sinne des Erfüllens der Schnittstellen dieser beiden

[2] ähnlich dem Proxy-Strukturmuster (vergl. [GAMM96]), mit dem Unterschied einer fehlenden gemeinsamen Oberklasse.

[3] Um Zweideutigkeiten mit einem RMI-Remote-Objekt zu vermeiden auch *EJBObject* genannt.

[4] Das Wort *implementiert* ist an dieser Stelle passender, wird aber auf Grund seiner Zweideutigkeit in diesem Kontext nicht verwendet.

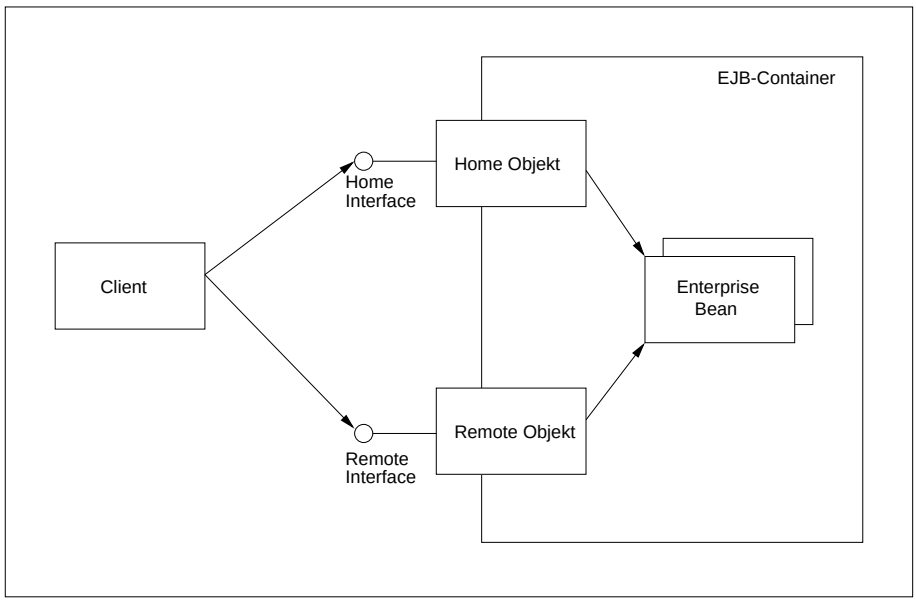

Abbildung 2.2: Zugriff auf die Bean nur über das Home- und das Remote-Objekt (aus [ROMA99])

Interfaces): Das Home- und das Remote-Objekt sind RMI-Remote-Objekte, auf die über ein Netzwerk zugegriffen werden kann. Die Zugriffe auf die eigentliche Bean[5] dürfen jedoch nicht über RMI geschehen, da sonst der Container keine Kontrolle über die Beans hat. Deswegen kapseln das Home- und Remote-Objekt die eigentliche Enterprise Bean und leiten Zugriffe auf die Anwendungslogik an diese (unter Kontrolle des Containers) weiter.

2.4.2 Clientzugriff über das Home- und Remote-Objekt

Der Zugriff eines Clients auf eine Enterprise JavaBean erfolgt nach folgendem Prinzip:

1. Der Client erhält vom Namens- und Verzeichnisdienst das Home-Objekt mit der im Home-Interface definierten Schnittstelle.
2. Der Client ruft auf diesem Objekt eine Fabrik-Methode (vergl. [GAMM96]) zum Erzeugen oder Finden einer Bean auf.
3. Das Home-Objekt gibt die Referenz auf ein Remote-Objekt (mit der Schnittstelle, die im Remote-Interface definiert ist) an den Client zurück.
4. Der Client ruft Methoden des Remote-Objekts auf, das diese an die entsprechenden Methoden der Enterprise Bean weiterleitet.

[5] Die Bean Klasse implementiert nicht das Interface *java.rmi.Remote* .

Die Bean ist also vollständig von Zugriffen des Clients gekapselt, und der Client interagiert für ihn transparent lediglich mit Proxy-Objekten, die es dem Container ermöglichen, die Zugriffe auf die Beans zu kontrollieren und seine ihm zugeteilten Aufgaben zu erfüllen.

2.5 Einschränkungen

Enterprise JavaBeans unterliegen einigen Einschränkungen, die sich auf Grund der Spezifikation und dem geplanten Einsatzgebiet ergeben. Die Enterprise JavaBean-Architektur ist von Sun Microsystems als Komponentenarchitektur mit klar definierten Schnittstellen und Zuständigkeiten konzipiert worden. EJBs sollen für systemnahe Funktionen nur die Schnittstellen, die ihnen der Container mit seiner Laufzeitumgebung liefert, benutzen. Durch diese Einschränkungen sollen Konflikte zwischen dem Container und der Bean vermieden werden. Ein paar der Einschränkungen, denen eine EJB unterliegt sind im folgenden zusammengestellt:

- Eine Bean darf keine Ein- oder Ausgaben über Klassen aus dem Package *java.io* tätigen.
- Sie darf keine Netzwerkverbindungen aufbauen bzw. Netzwerk-Sockets verwenden.
- Sie darf keine statischen Variablen benutzen (Statische Konstanten sind erlaubt).
- Eine EJB darf keine Klassen des AWT zur Ein- und Ausgabe nutzen.
- Sie darf weder Threads starten oder stoppen, noch Mechanismen zur Threadsynchronisation anwenden.
- Sie darf nicht über das JNI auf native Bibliotheken zugreifen.
- Eine Enterprise Bean darf nie *this* als Parameter eines Methodenaufrufes übergeben oder als Rückgabeparameter liefern, um einem Client nie die Möglichkeit zu geben direkt auf sie zuzugreifen.
- Sie darf keine eigenes Sicherheitsmanagement mit Hilfe der Klassen des Packages *java.security* implementieren oder Sicherheitspolicys ändern.

In der Praxis dürften diese Einschränkungen nicht von besonderer Bedeutung sein, da die meisten Restriktionen durch Dienste des Containers ausgeglichen werden oder dem von Sun vorgesehenen Anwendungsgebieten für EJB-Applikationen vollkommen widersprechen.

Lediglich die Vermeidung von statischen Variablen verhindert eine Bean als Singleton (vergl. [GAMM96]) zu implementieren.

2.6 Rollenverteilung

Bei der Entwicklung einer EJB-Anwendung arbeiten eine Vielzahl von Personen oder Gruppen zusammen, die alle unterschiedliche Aufgaben und Spezialgebiete haben. Sun definiert deswegen in seiner EJB-Spezifikation Rollen, die die Aufgaben der an der Entwicklung einer EJB-Anwendung beteiligten Gruppen widerspiegeln.

2.6.1 Enterprise Bean Provider

Der Bean Provider erstellt alle Komponenten einer Enterprise JavaBean und fasst diese in einem EJB-JAR-File zusammen. Typischerweise hat der Bean Provider fundierte Kenntnisse in der Entwicklung von Anwendungslogik zu einem bestimmten Thema.

Die von dem Bean Provider erstellten EJBs stellen Komponenten dar und sollten unabhängig von einer späteren Anwendungsumgebung und den in der Umgebung vorhandenen Ressourcen sein. Die EJB-Spezifikation bietet eine geeignete Schnittstelle, um Beans unabhängig von der Anwendungsumgebung erstellen und in verschiedensten Anwendungen einsetzten zu können. So implementiert der Bean Provider, wie oben bereits erwähnt, z.B. keine Mechanismen zum Sicherheitsmanagement.

Der Bean Provider erstellt einen Deployment-Descriptor, in dem er die Schnittstelle und die Eigenschaften (z.B. den Persistenzmechanismus) der Bean beschreibt.

2.6.2 Application Assembler

Der Application Assembler stellt eine oder mehrere EJBs des Bean Providers zu einer Anwendung zusammen und fügt ggf. weitere Anwendungsbestandeile (z.B. Servlets, JSP etc.) hinzu. Der Application Assembler erweitert die entsprechenden Deployment-Deskriptoren und fügt diese dann zu einem Deployment-Descriptor der Anwendung zusammen, damit der Bean Deployer weiß, wie er die Beans in dem EJB-Container zu installieren hat. Weiterhin definiert der Application Assembler im Deployment-Descriptor Sicherheitsprofile und die Sicherheitsrichtlinien für den Zugriff auf die Methoden der Beans, sowie die Abhängigkeiten der Beans untereinander oder von Ressourcen.

2.6.3 Bean Deployer

Der Bean Deployer ist für die Installation der von dem Application Assembler erstellten EJB-Anwendung in einem speziellen Container zuständig. Er hat typischerweise fundiertes Wissen über die Umgebung und das System, in der eine Anwendung lau-

fen wird (z.B. über den Server und den Container sowie die Tools die mit dem Container geliefert werden).

Zu den Aufgaben des Bean Deployers gehört, die Abhängigkeiten, die im Deployment-Descriptor beschrieben sind, aufzulösen und so die für den Betrieb der EJB nötigen Rahmenbedingungen zu schaffen. Hierzu zählen z.B. das Bereitstellen von Ressourcen und das Anlegen von Datenbanktabellen.

2.6.4 EJB-Server Provider

Der Server Provider erstellt den EJB-Server und ist typischerweise ein Experte auf dem Gebiet der verteilten Anwendungen und betriebssystemnahen Entwicklung. Da die Schnittstelle zwischen Server und Container von Sun nicht spezifiziert wurde, ist in der Praxis der Server Provider meist gleich dem Container Provider.

2.6.5 EJB-Container Provider

Der Container Provider implementiert den von der EJB-Spezifikation definierten EJB-Container. Dabei ist er verpflichtet, die in der Spezifikation genannten Mindestbedingungen für Container (Dienste, Schnittstellen und Aufgaben) zu erfüllen. In der Praxis liefert der Container Provider oft Tools zum Installieren der Beans, zum Erstellen des Deployment-Descriptors und zur Überwachung und Wartung des Systems.

2.6.6 Systemadministrator

Dem Systemadministrator fällt die Aufgabe zu, eine bestehende Anwendung zu überwachen und die Umgebung für einen EJB-Server zu schaffen. Dazu gehört z.B. die Bereitstellung und Überwachung des Netzwerkes und die Pflege von Userdaten.

Kapitel 3

Session-Beans

Wie in Kapitel 2 bereits kurz erwähnt, kapseln Session-Beans die Anwendungslogik und bilden für den Client die Schnittstelle für Operationen auf Daten der EJB-Anwendung. In diesem Kapitel werden die Konzepte von Session-Beans vertieft und auf die Implementation von Session-Beans und den Clientzugriff auf diese Beans eingegangen.

3.1 Session-Bean Konzepte

3.1.1 Conversational-State

In Kapitel 2.2.1 wurde bereits der Unterschied zwischen zustandslosen und zustandsbehafteten Session-Beans erläutert. Für die Implementation von Session-Beans bedeutet dieser Unterschied, dass zustandsbehaftete Session-Beans einen eigenen Conversational-State besitzen müssen, in dem die Daten, die über mehrere Methodenaufrufe des Clients erhalten bleiben sollen, gespeichert werden. Hierzu gehören z.B. offene Datenbankverbindungen, reservierte Ressourcen und zuvor berechnete Werte.

Zustandsbehaftete Session-Beans stehen während ihrer gesamten Existenz genau einem Client exklusiv zur Verfügung. Der Container muss Clientaufrufe an zustandsbehaftete Session-Beans immer an dieselbe Bean weiterleiten, da nur diese den entsprechenden Conversional-State speichert. Aufrufe an zustandslose Session-Beans können von jeder Session-Bean Instanz bearbeitet werden, da diese nichts über die vorangegangene Kommunikation mit dem Client wissen muss. Geht man davon aus, dass zwischen den Methodenaufrufen eines Clients relativ viel Zeit vergeht (da der Client z.B. auf Benutzereingaben wartet), kann der Container eine weitaus kleineren Pool von zustandslosen Session-Beans vorhalten, als Clients parallel mit dem Server verbunden sind. Bei zustandsbehafteten Session-Beans ist dies nicht möglich und pro Client-Session existiert genau eine Session-Bean.

3.1.2 Passivieren und Aktivieren

Um die Belastung des Arbeitsspeichers mit zurzeit nicht benötigten zustandsbehafteten Session-Beans (deren Client z.B. seit längerer Zeit auf Eingaben wartet) gering zu halten, besteht für den Container die Möglichkeit, die Session-Beans aus dem Arbeitsspeicher in einen Sekundärspeicher auszulagern. Dieser Vorgang wird Passivieren genannt. Um eine Session-Bean passivieren zu können, muss diese serialisierbar sein (d.h. das Interface *java.io.Serializable* implementieren). Vor dem Passivieren einer Bean ruft der Container die Methode *ejbPassivate()* einer Session-Bean auf, in der z.B. offene Datenbankverbindungen geschlossen werden können. Will ein Client wieder auf eine Methode der ihm zugeordneten zustandsbehafteten Session-Bean zugreifen, so de-serialisiert der Container die entsprechende Session-Bean und ruft deren Mehtode *ejbActivate()* auf, in der z.B. benötigte Datenbankverbindungen wieder geöffnet werden können. Danach steht die Beaninstanz wieder für Clientanfragen zur Verfügung.

Das Serialisieren von Objekten in Java ist eine sehr teure Operation. Daher sind Anwendungen, die viele Clientanfragen parallel bedienen und mit Hilfe von zustandslosen Session-Beans umgesetzt sind, merklich performanter, als Anwendungen, die auf zustandsbehafteten Session-Beans basieren. Aus diesem Grund werden zustandsbehaftete Session-Beans auch als schwergewichtig und zustandsloses Session-Beans als leichtgewichtig bezeichnet.

3.2 Lebenszyklus einer Session-Bean

Wie in Kapitel 2.1.2 bereits erwähnt, verwaltet der Container den Lebenszyklus der Enterprise Beans. Dies umfasst das Erzeugen und Löschen, aber auch das eventuelle Passivieren und Aktivieren einer Session-Bean.

Wie im Folgenden beschrieben, unterscheidet sich der Lebenszyklus einer zustandslosen Session-Bean grundlegend von dem einer zustandsbehafteten Session-Bean.

3.2.1 Lebenszyklus einer zustandslosen Session-Bean

Die Zustände, die eine zustandslose Session-Bean annehmen kann, sind relativ simpel, da dieser Bean-Typ nicht passiviert oder aktiviert werden muss. Man unterscheidet folgende beiden Zustände:

- nicht existent
- pooled ready

Die Bean wird durch den Aufruf der Methoden *newInstance(), setSessionContext(sc)* und *ejbCreate()* vom Container in den Zustand pooled ready überführt. Durch den Aufruf von *ejbRemove()* wird die Bean darüber informiert, dass sie in den Zustand *nicht existent* überführt wird.

Einen Überblick über den Lebenszyklus und die damit verbundenen Zustände einer zustandslosen Session-Bean gibt Abbildung 3.1.

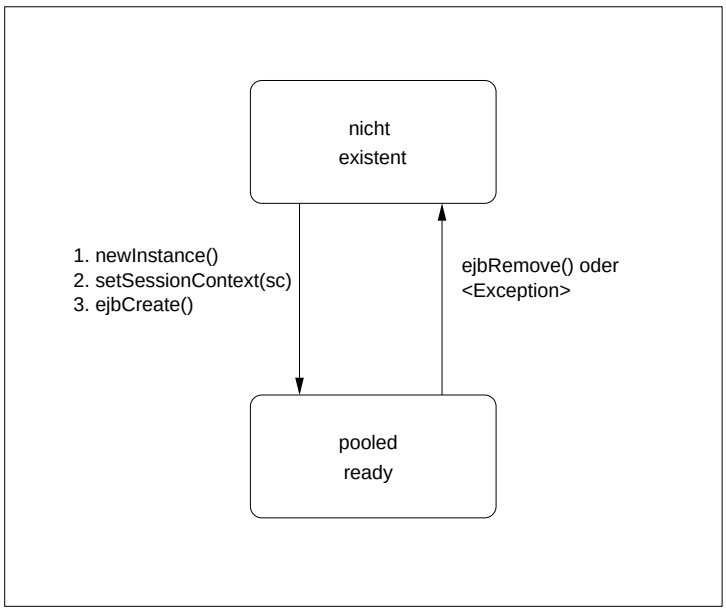

Abbildung 3.1: Lebenszyklus einer zustandslosen Session-Bean

3.2.2 Lebenszyklus einer zustandsbehafteten Session-Bean

Eine zustandsbehaftete Session-Bean kann aufgrund der Tatsache, dass sie passiviert werden kann, zwei zusätzliche Zustände annehmen. Hier unterscheidet man zwischen:

- nicht existent
- ready
- ready in transaction
- passiviert

Im Zustand *ready* steht die Bean für Methodenaufrufe zur Verfügung und kann vom Container jederzeit passiviert werden (d.h. in den Zustand *passiviert* versetzt werden). Während einer Transaktion, die mehrere Methodenaufrufe umfasst, ist die Bean im Zustand *ready in transaction* und kann nicht passiviert werden.

Einen Überblick über den Lebenszyklus und die damit verbundenen Zustände einer zustandsbehafteten Session-Bean gibt Abbildung 3.2.

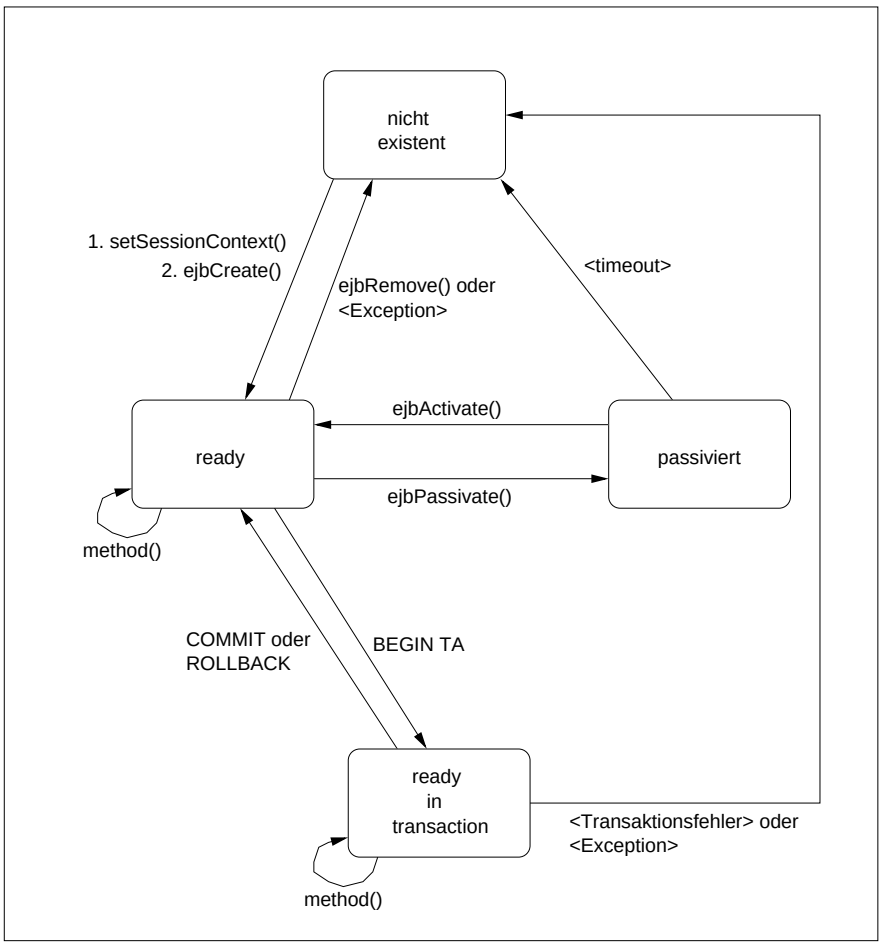

Abbildung 3.2: Lebenszyklus einer zustandsbehafteten Session-Bean

3.2.3 Exceptions

Während des Ausführens von Methoden einer Session-Bean kann es zu Fehlern kommen. Jede auftretende Exception wird an den Client weitergeleitet. Handelt es sich bei dem Fehler um eine RuntimeException, so wird die entsprechende Beaninstanz vom Container gelöscht, um inkonsistente Zustände zu vermeiden.

3.3 Implementation einer Session-Bean

3.3.1 Das Home-Interface einer Session-Bean

Wie in Kapitel 2.3.1 erwähnt, erweitert das Home-Interface einer Session-Bean das Interface *javax.ejb.EJBHome*. Im Home-Interface müssen dann zusätzlich zu den in *javax.ejb.EJBHome* deklarierten Methoden nur noch die verschiedenen *create()*-Methoden der Session-Bean deklariert werden.

Handelt es sich bei der Bean um eine zustandslose Session-Bean, wird nur eine *create()*-Methode ohne Parameter deklariert[1]. Im Home-Interface einer zustandsbehafteten Session-Bean ist es hingegen möglich, mehrere *create()*-Methoden mit unterschiedlichen Signaturen zu deklarieren, um die Bean mit den übergebenen Parametern zu initialisieren.

Alle *create*-Methoden müssen eine *javax.ejb.CreateException* erzeugen können und folglich in ihrer *throws*-Klausel deklarieren. Weiterhin müssen alle Methoden, die dem Client per RMI zur Verfügung stehen, eine *java.rmi.RemoteException* erzeugen können.

Ein Home-Interface einer Session-Bean könnte z.B. folgendermaßen aussehen:

```
package ckorn.studienarbeit.sbean.interfaces;

import java.rmi.*;
import javax.ejb.*;

public interface MyHomeInterface
   extends EJBHome {

   //eine create()-Methode ohne Paramter
   public RemoteInterface create()
      throws RemoteException, CreateException;

}
```

3.3.2 Das Remote-Interface einer Session-Bean

Das Remote-Interface erweitert das Interface *javax.ejb.EJBObject* und ergänzt die darin definierte Schnittstelle um die Methoden, die die eigentliche Anwendungslogik kapseln (vergleiche Kapitel 2.3.2).

Die Methoden der Anwendungslogik müssen, da sie per RMI zur Verfügung gestellt werden, eine *java.rmi.RemoteException* in ihrer *throws*-Klausel deklarieren.

Ein mögliches Remote-Interface einer Session-Bean könnte folgendermaßen aussehen:

```
package ckorn.studienarbeit.sbean.interfaces;
```

[1]Die Übergabe von Initialisierungsparametern an die *create()*-Methode einer zustandslosen Session-Bean macht keinen Sinn, da diese keinen Conversational-State verwaltet.

```
import java.rmi.*;
import javax.ejb.*;

public interface MyRemoteInterface
   extends EJBObject {

   public void method1()
      throws RemoteException;

   public void method2()
      throws RemoteException;

}
```

Das Home- und das Remote-Interface sollten in einem anderen Package, als die Bean-Klasse zusammen gefasst werden, um sie getrennt von der Bean-Klasse vertreiben zu können.

3.3.3 Die Session-Bean-Klasse

In Kapitel 2.3.3 wurde erwähnt, dass eine Enterprise Bean die Methoden ihres Home- und Remote-Interface implementieren muss, ohne diese beiden Interfaces im programmiertechnischen Sinne wirklich zu implementieren[2].

Die Methoden, die die Identitätenverwaltung der Bean betreffen, werden unter hinzufügen des Prefixes *ejb* implementiert. So wird z.B. die *create()*-Methode als *ejbCreate()* implementiert. Die Rückgabewerte dieser Methoden weichen z.T. ebenfalls von denen der im Home- und Remote-Interface deklarierten Methoden ab. So ist die Methode *ejbCreate()* der Bean-Klasse ohne Rückgabewert, die Methode *create()* des Home-Interfaces hingegen hat als Rückgabewert ein Objekt vom Typ des Remote-Interfaces. Dies ist nötig, damit der Container seine Aufgabe den Zugriff auf die Bean zu kapseln, erfüllen kann.

Jede Session-Bean-Klasse muss weiterhin das Interface *javax.ejb.SessionBean* implementieren, das die Methoden, die das Zustandsmanagement (den Lebenszyklus) einer Session-Bean betreffen, deklariert.

Das Interface *javax.ejb.SessionBean* hat folgenden Aufbau:

```
public interface javax.ejb.SessionBean
   extends javax.ejb.EnterpriseBean {

   public void ejbActivate()
```

[2] mittels des Schlüsselwortes *implements*

```
        throws EJBException, java.rmi.RemoteException;

    public void ejbPassivate()
        throws EJBException, java.rmi.RemoteException;

    public void ejbRemove()
        throws EJBException, java.rmi.RemoteException;

    public void setSessionContext(SessionContext ctx)
        throws EJBException, java.rmi.RemoteException;

}
```

Die eigentliche Session-Bean-Klasse muss also folgende Methoden implementieren:

```
package ckorn.studienarbeit.sbean.bean;

import java.rmi.*;
import javax.ejb.*;

public class MySessionBean
    implements javax.ejb.SessionBean {

    /* ---- Methoden des Zustandsmanagements ---- */

    public void setSessionContext(SessionContext ctx)
        throws EJBException, RemoteException {

        // Setzen und speichern des SessionContext-Objektes
    }

    public void ejbPassivate()
        throws EJBException, RemoteException {

        // Vorbereitungen zum Passivieren treffen
        // (nur zustandsbehaftete Session-Beans)
        // Hier werden z.B. offene DB-Verbindungen geschlossen
    }

    public void ejbActivate()
        throws EJBException, RemoteException {

        // Ressourcen wieder reservieren
        // (nur zustandsbehaftete Session-Beans)
        // Hier werden z.B. DB-Verbindungen wieder geöffnet
```

```
    }


    /* ---- Methoden der Identitätenverwaltung ---- */

    public void ejbRemove()
       throws EJBException, RemoteException {

       // Aufräumen
    }

    public void ejbCreate()
       throws EJBException, RemoteException {

       // Initialisierungen
    }

    // ggf. noch weitere ejbCreate()-Methoden mit anderer Signatur


    /* ---- Anwendungslogik ---- */

    public void method1
       throws EJBException, RemoteException {

       // Anwendungslogik
    }

    public void method2
       throws EJBException, RemoteException {

       // Anwendungslogik
    }
}
```

Über die Methode *setSessionContext(SessionContext ctx)* stellt der Container der Session-Bean ein SessionContext-Objekt zur Verfügung. Mit Hilfe dieses Objektes kann die Bean Informationen über ihre Umgebung erlangen und in Interaktion mit dem Container treten. So ist es der Bean mit diesem Objekt z.B. möglich abzufragen, ob bereits ein Rollback in einer laufenden Transaktion gesetzt wurde und so ggf. darauf zu reagieren (z.B. komplexe Berechnungen zu unterlassen). Weiterhin kann die Bean über das SessionContext-Objekt z.B. ein Rollback in einer laufenden Transaktion setzen oder Informationen über den am Client gemeldeten Benutzer erhalten. Der Container setzt vor dem ersten Methodenaufruf eines Clients den Session-Kontext der Session-Bean neu, so dass garantiert ist, dass diese immer über den aktuellen Session-Kontext verfügt.

Die anderen Methoden wurden bereits in den vorangegangenen Kapiteln erläutert.

Jede Methode der Bean-Klasse sollte die Runtime-Exception *javax.ejb.EJBException* erzeugen können, um den Container über Fehler (wie z.B. das fehlgeschlagene Öffnen einer Datenbankverbindung) unterrichten zu können.

3.3.4 Der Deployment-Descriptor

Über den Deployment-Descriptor einer Session-Bean werden dem Container Informationen, wie er diese Bean zu behandeln hat, mitgeteilt. Hierunter fallen z.B. der Typ der Bean (zustandslos oder zustandsbehaftet), die Transaktionsattribute (s. Kapitel 5) sowie der Name der Bean im Namens- und Verzeichnisdienst. Weiterhin werden im Deployment-Descriptor einer Session-Bean die Bean-Klasse und das Home- und Remote-Interface bekannt gemacht.

Ein Delpoyment-Descriptor für obiges Beispiel könnte z.B. folgendermaßen aussehen:

```
<?xml version="1.0" encoding="Cp1252"?>
<ejb-jar>
   <description>Eine einfache Session-Bean</description>
   <display-name>MySessionBean</display-name>
   <enterprise-beans>
      <session>
         <ejb-name>MySessionBean</ejb-name>
         <home>
          ckorn.studienarbeit.sbean.interfaces.MyHomeInterface
         </home>
         <remote>
          ckorn.studienarbeit.sbean.interfaces.MyRemoteInterface
         </remote>
         <ejb-class>
          ckorn.studienarbeit.sbean.bean.MySessionBean
         </ejb-class>
         <session-type>Stateless</session-type>
         <transaction-type>Bean</transaction-type>
      </session>
   </enterprise-beans>
</ejb-jar>
```

3.4 Clientzugriff

Wie oben bereits geschildert, greift ein Client niemals direkt auf eine Session-Bean zu, sondern kommuniziert mit dem vom Container erzeugten Home- bzw. Remote-

Objekt, um auf die durch die Bean gekapselte Anwendungslogik zuzugreifen.

3.4.1 Protokolle für den Clientzugriff

Da es sich bei EJB-Anwendungen im Allgemeinen um verteilte Anwendungen handelt, erfolgt die Kommunikation eines Clients mit den Beans (bzw. dem Home-Objekt und dem Remote-Objekt) über Protokolle, die es erlauben, Objekte über ein Netzwerk zu nutzen. Seit der Spezifikation EJB 1.1 steht einem Client für die Kommunikation mit den EJB-Objekten nicht nur Java RMI, sondern auch IIOP (CORBA) zur Verfügung. Da jedoch das Home- und das Remote-Objekt RMI-Remote-Objekte darstellen, muss ein EJB 1.1 konformer Container einen RMI/IIOP-Wrapper bereitstellen.

Für Kommunikation mit dem Namens- und Verzeichnisdienst stehen demzufolge die beiden Protokolle RMI Registry und COS Naming (CORBA Naming Service), aber auch der Standard LDAP (Lightweight Directory Access Protocol) zur Verfügung.

3.4.2 Der Clientzugriff im Detail

Im folgenden soll schematisch der Zugriff eines Clients auf die Anwendungslogik einer Session-Bean skizziert werden.

Zunächst muss ein Client sich beim Namens- und Verzeichnisdienst mit seinem initialen Kontext (der aktuellen Position innerhalb des durch den Namens- und Verzeichnisdienst abgebildeten Baumes) anmelden. Danach erfragt er über das JNDI beim Namens- und Verzeichnisdienst des EJB-Servers das Home-Objekt der entsprechenden Enterprise Bean. Dieses wird ihm von dem Dienst als *java.lang.Object* zurückgeliefert und muss folglich durch einen Typecast in ein Objekt des entsprechenden Home-Interfaces der jeweiligen Bean konvertiert werden. Da, wie oben erwähnt, seit Version 1.1 der EJB-Spezifikation auch IIOP neben RMI als Netzwerkprotokoll Verwendung findet, muss der Typecast mit Hilfe der statischen Methode *narrow()* der Klasse *javax.rmi.PortableRemoteObject* geschehen.

Über die Methode *create()* des Home-Objektes kann der Client nun eine Referenz auf das Remote-Objekt der Enterprise Bean erzeugen und Methoden des Remote-Objektes zur Kommunikation mit der Bean aufrufen. Wenn das Remote-Objekt von dem Client nicht mehr benötigt wird, muss dies dem Container durch Aufruf der Methode *remove()* bekannt gemacht werden, damit dieser die Instanz der Session-Bean löschen kann.

Eine mögliche Clientanwendung könnte folgendermaßen aussehen[3]:

[3] Auf eine komplexe Fehlerbehandlung wurde aus Gründen der Übersichtlichkeit verzichtet.

```
import java.rmi.*;
import java.util.*;
import java.naming.*;
import ckorn.studienarbeit.sbean.interfaces.*;

public static void main(String[] arg) {
   try {
      // Umegebungsvariablen für JNDI-Zugriff setzen
      System.setProperty("java.naming.factory.initial",
         "org.jnp.interfaces.NamingContextFactory");
      System.setProperty("java.naming.provider.url",
         "localhost:1099");

      // initialen Kontext erzeugen
      InitialContext context = new InitialContext();

      // Home-Objekt anfordern + typecast
      HomeInterface homeObject = (HomeInterface)
         PortableRemoteObject.narrow
         (context.lookup("MySessionBean"));

      // Remote-Objekt erzeugen
      RemoteInterface remoteObject =
         homeObject.create();

      // Kommunikation mit der Bean
      remoteObject.method1();
      remoteObject.method2();
      ...

      // abmelden
      remoteObject.remove();

   }
   catch (Exception e) {
      System.out.println("Fehler: "+e.toString());
   }
}
```

Kapitel 4

Entity-Beans

Wie bereits erwähnt, sollen Entity-Beans die Datenobjekte einer EJB-Anwendung abbilden. Sie stellen Geschäftsobjekte dar, die meist über einen längeren Zeitraum gespeichert werden müssen.

4.1 Entity-Bean Konzepte

4.1.1 Probleme durch parallelen Zugriff

Da Entity-Beans mehreren Clients parallel zur Verfügung stehen, können sich inkonsistente Zustände ergeben. Z.B. kann es vorkommen, dass ein Client die Daten einer Entity-Bean verändert, während ein anderer diese Daten liest oder ebenfalls verändert. Im EJB-Konzept werden diese Probleme zum einen dadurch vermieden, dass Clientzugriffe auf die Beans vom Container serialisiert werden, zum anderen versucht man diese Inkonsistenzen durch Transaktionen (siehe Kapitel 5) zu umgehen.

Die EJB-Spezifikation sieht nach [HALT01] keine ausreichende Locking-Mechanismen vor, Entity-Beans für einen exklusiven Zugriff zu sperren. Daher ist es schwer bei fein granulierten Entity-Beans, die in gegenseitigen Abhängigkeiten stehen, Inkonsistenzen sicher zu vermeiden. Aus diesem Grund und aus den in Kapitel 7.2.1 genannten Performancegewinnen sollten Entity-Beans möglichst grob granuliert sein und möglichst autake Geschäftsobjekte abbilden.

4.1.2 Identität und Instanz

Entity-Beans werden, ähnlich wie Session-Beans, in einen permanenten Speicher ausgelagert, um sie zum einen persistent zu halten, zum anderen um die Anzahl

der gleichzeitig im Arbeitsspeicher vorgehaltenen Objekten zu begrenzen. Wie in Kapitel 2.1.2 erwähnt, hält der Container einen Pool von Entity-Beans bereit, denen bei Bedarf ihre spezifischen Daten (eine sog. Identität) zugeordnet werden kann.

Man unterscheidet also zwischen der Instanz einer Entity-Bean und ihrer Identität. Die Identität einer solchen Bean besteht aus allen persistenten Attributen (s.u.), inklusive dem Primärschlüssel. Die Instanz hingegen bezeichnet ein sich im Speicher befindliches Objekt einer Entity-Bean. Diesem muss keine Identität zugeordnet sein. Erst nach Zuordnung der Identität zur Instanz steht die Entity-Bean einem Client für Methodenaufrufe zur Verfügung.

Für den Client ist dieser Mechanismus des Instanzenpoolings vollkommen transparent. Ein Client arbeitet also immer mit derselben Identität einer Entity-Bean, nicht aber mit derselben Instanz.

4.1.3 Persistente und transiente Attribute

Nicht alle Attribute einer Entity-Bean sollen permanent gespeichert werden. Man unterscheidet hierzu persistente und transiente Attribute. Persistente Attribute sind Teil der Bean-Identität, d.h. sie werden z.B. in einer Datenbank gespeichert. Transiente Attribute einer Bean sind z.B. Variablen, die Zwischenergebnisse o.ä. speichern. Sie müssen ein Auslagern der Bean in einen permanenten Speicher (serialisieren) und das spätere Deserialisieren nicht überdauern, da sie ohne großen Aufwand jederzeit neu berechnet werden können.

Alle Attribute einer Entity-Bean sind standardmäßig persistent. Durch das Java-Schlüsselwort *transient* wird dem Container mitgeteilt, dass diese Attribute bei einer Serialisierung des Objektes nicht berücksichtigt werden müssen. In der Praxis ergeben sich oft eine Reihe von Attributen, die nicht persistent gehalten werden müssen bzw. sollen.

4.1.4 get- und set-Methoden

Alle persistenten Attribute einer Entity-Bean sollten, ähnlich den Attributen einer JavaBean, mit Hilfe einer Methode *getAttributname()* gelesen und mittels einer Methode *setAttributname(Attribut)* geschrieben werden können. Dies ist laut der EJB-Spezifikation nicht notwenig, es reicht hiernach diese Attribute als *public* zu deklarieren. In der Praxis erwarten jedoch manche Container solche get- und set-Methoden.

Diese get- und set-Methoden sind im Remote-Interface zu deklarieren und in der Bean zu implementieren.

4.1.5 Finder-Methoden

Es ist möglich sogenannte Finder-Methoden nach dem Muster *findByAttributename()* im Home-Interface zu deklarieren, damit wie in einer Datenbank, eine Entity-Bean über alle ihre Attributen gesucht werden kann. Diese werden dann entweder, im Falle von BMP, in der Bean-Klasse implementiert oder, im Falle von CMP, automatisch vom Container bereit gestellt.

Weiterhin kann der Container im Falle von CMP automatisch die Methoden *findByPrimaryKey()*, die eine Entity-Bean anhand ihres Primärschlüssels sucht, und die Methode *findAll()*, die alle Entity-Beans zurück gibt, implementieren. Der Rückgabewert dieser Finder-Methoden ist im Falle von Java 2 vom Typ *java.util.Collection*, im Falle von Java 1.x vom Typ *java.util.Enumeration*.

Im Home-Interface können weitere Methoden nach dem Muster *find...()* implementiert werden, um Beans z.B. über einen Wertebereich zu finden. Diese Methoden können jedoch nur bei BMP eingesetzt werden, da nur hier der Bean z.B. der Zugriff auf eine relationale Datenbank per SQL zur Verfügung steht.

Der derzeitige EJB-Standard 1.1 sieht keinen Mechanismus vor, dass CMP verwalte Entity-Beans mit Hilfe von Wildcards, über eine boolsche Verknüpfung ihrer Attributwerte oder über einen bestimmten Wertebereich ihrer Attribute gefunden werden können. Manche Persistenzmanager einiger Container bieten dennoch (mit Hilfe von speziellen Kofigurationsfiles, ähnlich dem Deloyment-Descriptor) eine solche Möglichkeit. Diese Mechanismen sind jedoch nicht standardisiert und sehr von dem verwendeten Persistenzmanager abhängig, so dass im Hinblick auf die Portierbarkeit der Beans die Verwendung dieser Möglichkeiten problematisch erscheinen. Der EJB-Standard 2.0 sieht in dieser Hinsicht Verbesserungen vor. Siehe hierzu Kapitel 7.1.

4.1.6 Der Primärschlüssel

Eine Entity-Bean kann über den Primärschlüssel eindeutig identdefiziert werden. Dieser Mechanismus entspricht dem bekannten Verfahren bei Datenbanken (vergl. [VOSS99]). Ein Primärschlüssel ist also eine Redundanz freie eins-zu-eins Abbildung auf eine Entity-Bean-Identität.

Mit Hilfe des Primärschlüssels kann der Container eine Entity-Bean-Identität finden. Dies geschieht über die oben beschriebene Methode *findByPrimaryKey()* des Home-Objekts.

Die Attribute, die den Primärschlüssel bilden, müssen nicht nur in der Bean-Klasse, sondern zusätzlich auch in einer Primärschlüssel-Klasse, die das Interface *java.io.Serializable* implementieren muss, deklariert sein. Die Primärschlüssel-Klasse wird als Teil der Entity-Bean in einem JAR-File zusammen mit dem Home-Interface, dem

Remote-Interface, der Bean-Klasse und dem Deployment-Descriptor im Container installiert.

4.1.7 Persistenzmechanismen

Wie in Kapitel 2.2.2 erwähnt, sieht die Version 1.1. der EJB-Spezifikation zwei verschiedene Persistenzmechanismen für Entity-Beans vor. Dies ist zum einen die automatische, containergesteuerte Persistenz (Container Managed Persistence (CMP)) und zum anderen die von der Bean selbst verwaltete Persistenz (Bean Managed Persistence (BMP)).

Die Vorteile der CMP liegen darin, dass Entity-Beans mit CMP flexibler in verschiedenen Umgebungen eingesetzt werden können, da sie nicht auf ein bestimmten permanenten Speicher (z.B. eine relationale Datenbank) angewiesen sind. Darüber hinaus beinhalten sie keinen SQL-Code o.ä., der für die Portierung der Bean problematisch sein könnte. Der Container und dessen Persistenzmanager sind hingegen an einen bestimmten permanenten Speicher gebunden, was es nahe legt, die Persistenz der Bean durch den Container zu verwalten.

Die EJB 1.1 Spezifikation sieht vor, dass ein EJB 1.1 konformer Container die folgenden Attribute einer Entity-Bean per CMP automatisch persistent halten kann:

- primitive Java-Datentypen
- Serialisierbare Java-Objekte
- Referenzen auf Home-Interfaces (anderer Beans)
- Referenzen auf Remote-Interfaces (anderer Beans)

Probleme beim Einsatz von Container Managed Persistence

In der Praxis setzen die wenigsten EJB-Container die Spezifikation in diesem Punkt vollständig um. Speziell das Speichern und Suchen über serialisierte Java-Objekte scheint ein Problem für die Hersteller von Container zu sein. Die Schwierigkeit besteht darin, dass Java-Objekte eine sehr komplexe Struktur haben können und ggf. in Relationen gespeichert werden müssen. Ein Beispiel für ein solches Objekt ist z.B. eine beliebig dimensionale Struktur beliebig vieler Elemente, das mit Hilfe einer m zu n-Beziehungen auf die Datenbank abgebildet werden muss.

Es ist also bei CMP nicht garantiert, dass alle Attribute einer Entity-Bean von Container gespeichert werden können. Dies schränkt die Portierbarkeit von Entity-Beans in der Praxis häufig mehr ein, als die Verwendung von BMP als Persistenzmechanismus, da derzeit davon ausgegangen werden kann, dass fast alle Container Entity-Beans in relationalen SQL-Datenbanken speichern und durch das JDBC API, das

der Container anbietet, bereits eine genügend abstrakte Schnittstelle zu Datenbankmangementsystemen geschaffen ist.

Weiterhin ist die Suche nach CMP-verwalteten Entity-Beans über die Finder-Methoden, wie oben beschrieben, nur sehr eingeschränkt möglich.

Zurzeit ist in der Praxis die Umsetzung von EJB-Anwendungen mit Hilfe von CMP nur sehr eingeschränkt möglich, da schon bei kleinen Anwendungen die Komplexität der Datenbank zu hoch ist, um vom Persistenzmanager des Containers verwaltet werden zu können.

Die Unterstützung von CMP wird durch die Spezifikation EJB 2.0 verbessert. So sieht EJB 2.0 die Einbindung von Persistenzmanagern von Drittanbietern über eine definierte Schnittstelle und die an SQL angelehnte Abfragesprache EJB QL vor. Siehe hierzu auch Kapitel 7.1.

4.2 Lebenszyklus einer Entity-Bean

Der Lebenszyklus einer Entity-Bean ist eng an das Instanzenpooling und die Unterscheidung zwischen Instanz und Identität gekoppelt. Die Spezifikation EJB 1.1 unterscheidet drei mögliche Zustände:

- nicht existent
- pooled
- ready

Die Zustände *pooled* und *ready* unterscheiden sich dadurch, dass die Bean im Zustand *ready* eine Identität besitzt und Clientanfragen beantworten kann. Bei einer Bean im Zustand *pooled* hingegen handelt es sich um eine Instanz ohne zugewiesene Identität.

Beans im Zustand *pooled* werden benutzt, um die Clientanfragen an das Home-Objekt zu beantworten, also z.B. eine neue Bean zu erzeugen oder eine Entity-Bean zu finden. Diese Aktionen kann jede beliebige sich im Pool befindende Bean durchführen, da hierfür keine Identität notwendig ist.

Abbildung 4.1 gibt einen Überblick über den Lebenszyklus einer Entity-Bean.

In [DENN00] wird der Zustand *ready* in drei Unterzustände aufgeteilt, die von der EJB-Spezifikation nicht explizit aufgeführt werden. Es wird zwischen den folgenden Unterzuständen unterschieden, die sich aus den den Lebenszyklus betreffenden Methoden ergeben:

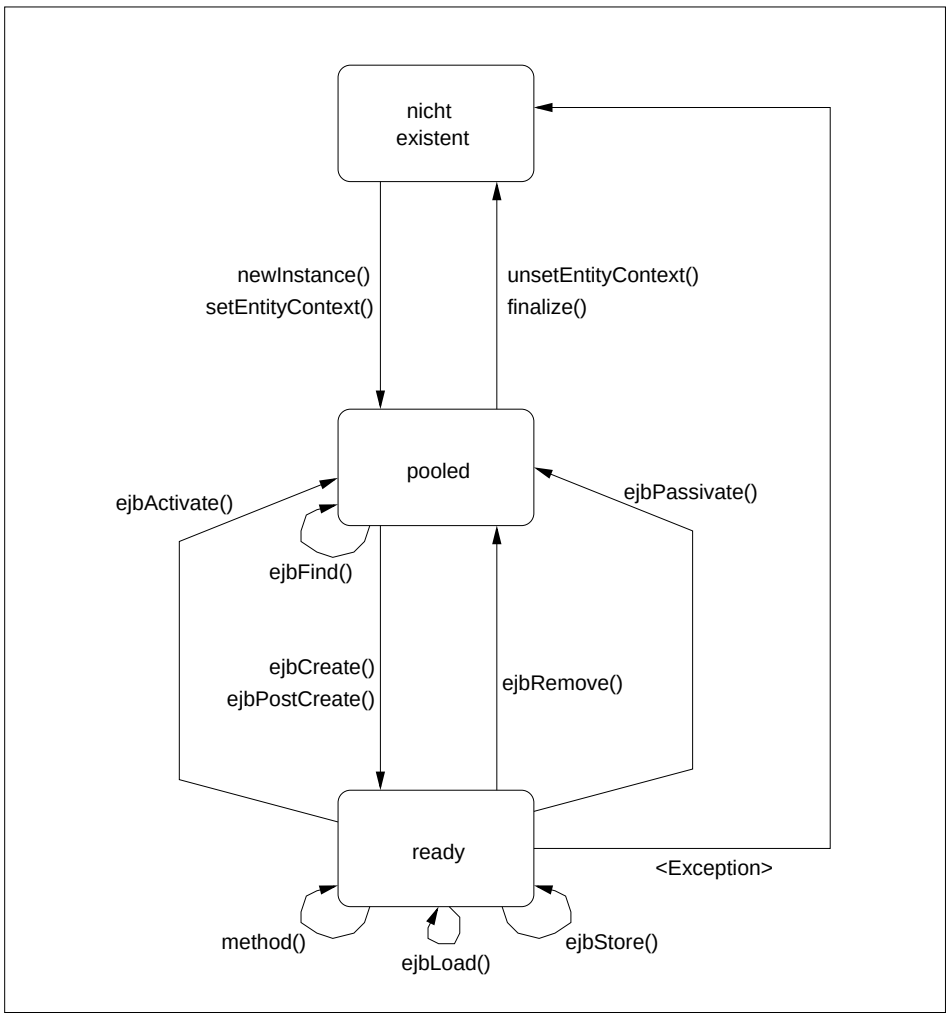

Abbildung 4.1: Lebenszyklus einer Entity-Bean

- *ready async:* Die Attribute der Bean sind evtl. nicht mit denen in der Datenbank abgeglichen.
- *ready sync:* Die Attribute der Bean sind mit den Inhalten der Datenbank synchronisiert.
- *ready update:* Die Attribute der Bean wurden vom Client geändert und sind noch nicht in die Datenbank übertragen worden.

4.2.1 Verwaltung des Instanzen-Pools

Der Container kann die Anzahl der Beans im Pool (Zustand *pooled*) nach bestimmten Kriterien erhöhen bzw. verringern. Hierzu erstellt der Container eine neue Bean-Instanz und ruft ihre *setEntityContext()*-Methode auf, die die Aufgabe hat, das Entity-Kontext-Objekt zu speichern. Will ein Container die Anzahl der sich im Pool

befindlichen Beans dezimieren, so ruft er die Methode *unsetEntityContext()* auf und löscht daraufhin die Beaninstanz.

Der Entity-Kontext entspricht dem in Kapitel 3.3.3 vorgestellten Session-Kontext einer Session-Bean. Über das EntityContext-Objekt (*javax.ejb.EntityContext*) kann die Entity-Bean z.B. Informationen über ihre Umgebung oder gesetzte Rollbacks erfragen und hat die Möglichkeit, Rollbacks zu setzen. Außerdem kann die Entity-Bean über dieses Objekt auf ihre Primärschlüssel-Klasse zugreifen.

Manche Container bieten dem Systemadministrator die Möglichkeit, auf die Anzahl der im Pool befindlichen Entity-Beans Einfluss zu nehmen. So kann er z.B. zu Zeiten mit hoher Belastung den Pool vergrößern und somit ggf. die Performance des Servers zu steigern.

4.2.2 Verwaltung der Identitäten

Der Client erhält über eine Finder-Methode des Home-Objektes eine Referenz auf ein Remote-Objekt. Der Container hat in diesem Fall die Aufgabe, einer beliebigen, sich im Pool befindlichen Instanz eine Identität zuzuweisen. Im Anschluss daran ruft der Container die Methode *ejbActivate()* der Bean-Klasse auf, die der Bean die Möglichkeit gibt, Initialisierungen vorzunehmen (z.B. Datenbankverbindungen zu öffnen oder mittels des Entity-Kontext-Objektes auf die Primärschlüsselklasse zuzugreifen). Die Bean befindet sich nun im Zustand *ready async*. Ihre Attribute sind also noch nicht mit den in der Datenbank gespeicherten Werten synchronisiert.

Vor jedem Clientaufruf wird die Bean durch den Aufruf der Methode *ejbLoad()* in den Zustand *ready sync* überführt. In dieser Methode werden bei BMP die entsprechenden Felder aus der Datenbank gelesen und die Attribute der Bean gesetzt. Bei CMP werden die Bean-Attribute automatisch vom Container gesetzt und die Bean über diesen Methodenaufruf lediglich über einen bevorstehenden Clientaufruf informiert. Durch das Serialisieren der Zugriffe und dem Transaktionsmanagement ist gesichert, das hierbei keine inkonsistenten Zustände auftreten.

Hat der Client Änderungen an den Attributen der Bean vorgenommen, befindet sich die Bean im Zustand *ready update*, und es muss dafür gesorgt werden, dass die geänderten Attributwerte gespeichert werden. Hierzu ruft der Container die Methode *ejbStore()* auf. Bei CMP werden die Attributwerte automatisch vom Container gespeichert, bei BMP erfolgt das Speichern durch Implementation dieser Methode. Direkt nach diesem Methodenaufruf befindet sich die Bean wieder im Zustand *ready sync*.

Wird eine Bean längere Zeit nicht benutzt, so kann der Container die Bean passivieren. Damit die Bean belegte Ressourcen frei geben kann, ruft der Container vorher die Methode *ejbPassivate()* auf.

Über die Methode *create()* des Home-Objektes, kann der Client eine neue Entity-

Bean erstellen. Hierdurch wird der Container angewiesen, eine Bean aus dem Pool bereit zu stellen und deren *ejbCreate()*-Methode aufzurufen. In dieser Methode können allgemeine Initialisierungen der Bean erfolgen. Danach ruft der Container die Methode *ejbPostCreate()* auf, in der die Bean weitere Initialisierungen vornehmen kann, für den sie den ihr nun zur Verfügung stehenden Entity-Kontext benötigt. Für jede *ejbCreate*-Methode muss eine entsprechende *ejbPostCreate*-Methode mit gleicher Signatur implementiert werden.

Um eine Bean zu löschen, benutzt der Client entweder die *remove()*-Methode des Home- oder des Remote-Interface. Der Container leitet den Aufruf an die Methode *ejbRemove()* der Entity-Bean weiter, die entweder die entsprechenden Einträge aus der Datenbank selbst löscht (BMP) und benutzte Ressourcen frei gibt, oder - wenn der Container die Persistenz der Bean verwaltet (CMP) - das Löschen der Datenbankeinträge dem Container überlässt.

Späte Initialisierung

Die EJB-Spezifikation sieht eine Performenceoptimierung des Containers vor, die eng mit der Identitätenverwaltung verknüpft ist: Ein Container kann einzelne Attribute einer Entity-Bean erst dann aus der Datenbank lesen, wenn diese benötigt werden. So verringert der Container, auf eine für den Client transparente Weise, die Größe der im Arbeitsspeicher vorgehaltenen Objekte.

4.2.3 Exceptions

Exceptions in Entity-Beans haben die gleichen Konsequenzen wie Exceptions in Session-Beans. Siehe hierzu Kapitel 3.2.3.

4.3 Implementation einer Entity-Bean

4.3.1 Home- und Remote-Interface

Zusätzlich zu den Methoden, die im Home-Interfaces einer Session-Bean deklariert sind, muss das Home-Interface einer Entity-Bean alle Finder-Methoden, die zur Suche nach der Bean verwendet werden können sollen, deklarieren.

Es ist zu beachten, dass manche Container erwarten, dass die get- und set-Methoden sowie die Finder-Methoden einer Entity-Bean, die den vom Container bereit gestellten Persistenzmechanismus nutzt (CMP), als *abstract* deklariert werden.

Ansonsten entsprechen das Home- und das Remote-Interface einer Entity-Bean de-

nen einer Session-Bean und werden deswegen hier nicht im Detail betrachtet.

4.3.2 Die Primärschlüssel-Klasse

Die Primärschlüssel-Klasse enthält alle Attribute der Entity-Bean, die den Primärschlüssel bilden und somit die Bean eindeutig identdefizieren. Die einzige Auflage, die für die Implementation einer Primärschlüssel-Klasse besteht, ist, dass sie das Interface *java.io.Serializable* implementieren muss, um sicher zu stellen, dass der Container die Bean in z.B. einer Datenbank persistent halten kann.[1]

Eine mögliche Primärschlüssel-Klasse könnte beispielsweise folgendermaßen aussehen:

```
package ckorn.studienarbeit.ebean.interfaces;

public class MyPrimaryKey
   implements java.io.Serializable {

   public int keyIndex;

}
```

Es ist nicht zwingend notwendig eine neue Klasse für den Primärschlüssel zu erstellen. Als Primärschlüssel-Klasse können alle bestehenden Java-Klassen, die das Interfaces *java.io.Serializable* implementieren, genutzt werden.

4.3.3 Die Entity-Bean-Klasse

In der Entity-Bean Klasse werden alle Methoden, die im Home- und Remote-Interface deklariert wurden, sowie die Attribute der Primärschlüssel-Klasse implementiert. Die Bean-Klasse implementiert allerdings nur das Interface *javax.ejb.EntityBean*.

Das Interface *javax.ejb.EntityBean* hat folgenden Aufbau:

```
public Interface EntityBean
   extends javax.ejb.EnterpriseBean {

   void ejbActivate()
     throws EJBException, java.rmi.RemoteException;
```

[1] Das Interface *javax.ejb.EntityBean*, das jede Entity-Bean implementieren muss, ist bereits von dem Interface *java.io.Serializable* abgeleitet und stellt somit die Serialisierbarkeit der Entity-Bean sicher.

```
    void ejbLoad()
      throws EJBException, java.rmi.RemoteException;

    void ejbPassivate()
      throws EJBException, java.rmi.RemoteException;

    void ejbRemove()
      throws EJBException, java.rmi.RemoteException;

    void ejbStore()
      throws EJBException, java.rmi.RemoteException;

    void setEntityContext(EntityContext ctx)
      throws EJBException, java.rmi.RemoteException;

    void unsetEntityContext()
      throws EJBException, java.rmi.RemoteException;

}
```

Die Entity-Bean-Klasse muss die Methoden des Home- und Remote-Interface, sowie die im der Primärschlüssel-Klasse deklarierten Methoden und Attribute implementieren. Im Falle von CMP werden die Finder-Methoden nicht, und die Methoden die den Lebenszyklus der Bean betreffen, leer implementiert.

Eine Entity-Bean-Klasse könnte also folgenden Aufbau haben:

```
package ckorn.studienarbeit.ebean.bean;

import javax.ejb.*;
import java.rmi.*;
import java.util.*;
import ckorn.studienarbeit.ebean.interfaces.MyPrimaryKey;

public class MyEntityBean

    implements EntityBean {

    /* ---- die Attribute der Bean --- */

    public MyPrimaryKey keyIndex; //der Primaerschluessel
    public String attr1; //ein weiteres Attribut


    /* ---- Methoden des Zustandsmanagements ---- */
```

```
public void ejbLoad()
   throws EJBException, RemoteException {

   // leer gelassen, da CMP
}

public void ejbStore()
   throws EJBException, RemoteException {

   // leer gelassen, da CMP
}

public void setEntityContext(EntityContext ctx)
   throws EJBException, RemoteException {

   // Setzen und speichern des EntityContext-Objektes
}

public void unsetEntityContext(EntityContext ctx)
   throws EJBException, RemoteException {

   // EntityContext-Objekt löschen
}

public void ejbPassivate()
   throws EJBException, RemoteException {

   // Vorbereitungen zum Passivieren treffen
   // Hier werden z.B. offene DB-Verbindungen geschlossen etc.
}

public void ejbActivate()
   throws EJBException, RemoteException {

   // Ressourcen wieder reservieren
   // Hier werden z.B. DB-Verbindungen wieder geöffnet
}


/* ---- Methoden der Identitätenverwaltung ---- */

public void ejbRemove()
   throws EJBException, RemoteException {

   // leer gelassen, da CMP
```

```
    }

    public void ejbCreate()
       throws EJBException, RemoteException {

       // Initialisierungen
    }

    public void ejbPostCreate()
       throws EJBException, RemoteException {

       // weitere Initialisierungen
    }


    /* ---- get- und set-Methoden ---- */

    public void setKeyIndex(value)
       throws EJBException, RemoteException {

          keyIndex.keyIndex = value;
    }

    public MyPrimaryKey getKeyIndex()
       throws EJBException, RemoteException {

          return(keyIndex.keyIndex);
    }

    public String setAttr1(value)
       throws EJBException, RemoteException {

          attr1 = value;
    }

    public String getAttr1()
       throws EJBException, RemoteException {

          return(attr1);
    }

}
```

4.3.4 Der Deployment-Descriptor

Der Deployment-Descriptor einer Entity-Bean enthält z.B. Angaben über den Persistenzmechanismus (CMP oder BMP), welche Attribute per CMP persistent gehalten werden sollen usw.

Ein einfacher Deployment-Descriptor einer Entity-Bean könnte beispielsweise folgende Struktur haben:

```
<?xml version="1.0" encoding="Cp1252"?>
<ejb-jar>
   <description>Eine einfache Entity-Bean</description>
   <display-name>MyEntityBean</display-name>
   <enterprise-beans>
      <entity>
         <ejb-name>MyEntityBean</ejb-name>
         <home>
          ckorn.studienarbeit.ebean.interfaces.MyHomeInterface
         </home>
         <remote>
          ckorn.studienarbeit.ebean.interfaces.MyRemoteInterface
         </remote>
         <ejb-class>
          ckorn.studienarbeit.ebean.bean.MyEntityBean
         </ejb-class>
         <prim-key-class>
          ckorn.studienarbeit.ebean.interfaces.MyPrimaryKey
         </prim-key-class>
         <persistence-type>Container</persistence-type>
         <cmp-field><field-name>keyIndex</field-name></cmp-field>
         <cmp-field><field-name>attr1</field-name></cmp-field>
         <primkey-field>keyIndex</primkey-field>
      </entity>
   </enterprise-beans>
</ejb-jar>
```

4.3.5 Datenbank-Mapping

Im Deployment-Descriptor wird nicht festgelegt, welches Attribut einer Entity-Bean auf welches Datenbankfeld gemappt wird. Dies würde auch der Unabhängigkeit vom benutzen permanenten Speicher widersprechen. Diese Informationen werden vom Bean-Deployer zum Zeitpunkt des Deployments anhand der Informationen aus dem Deployment-Descriptor und der jeweiligen Umgebung festgelegt. Die Kommunikation zwischen Bean-Deployer und dem Persistenzmanager des Containers, welche

Datenbankfelder abgebildet werden, ist containerspezifisch. Meist konfiguriert man den Persistenzmanager über XML-Dateien, ähnlich dem Deployment-Descriptor.

4.4 Clientzugriff

Der Clientzugriff entspricht dem im Kapitel 3.4 vorgestellten Verfahren für den Clientzugriff auf Session-Beans. Der Client sollte jedoch möglichst wenig Anwendungslogik, die Operationen auf den Entity-Beans ausführt, besitzen. Um Operationen auf Entity-Beans auszuführen, sollten möglichst Methoden der Session-Beans benutzt werden, um die Aufgabenteilung, die durch die Mehr-Schichten-Architektur vorgegeben ist, einzuhalten.[2]

[2]Dies macht auch aus Performenzgesichtspunkten Sinn, da ggf. die Kommunikation über RMI unterlassen werden kann. Siehe hierzu Kapitel 7.2.1.

Kapitel 5

Transaktionen

Das Thema Transaktionen ist zu komplex, um in dieser Arbeit ausführlich dargestellt zu werden. Daher werden in diesem Kapitel nur die Grundlagen für die Benutzung von Transaktionen in der EJB-Archtektur vorgestellt.

5.1 Zweck von Transaktionen

Transaktionen fassen mehrere Arbeitsschritte zusammen, die entweder alle erfolgreich abgearbeitet werden, oder alle nicht ausgeführt werden. Transaktionen werden oft in Systemen nötig, die parallele Operationen auf dieselben Daten ausführen. In solchen Systemen kann es durch diese parallelen Operationen zu Inkonsistenzen kommen. Ein kurzes Beispiel soll dies verdeutlichen:

Ein gewisser Bestand wird durch Operationen parallel agierender Objekte inkrementiert und dekrementiert. Der Bestand darf nie negativ werden. Objekt A liest den Bestand und stellt einen positiven Bestand fest. Danach liest Objekt B den gleichen Bestand. Als nächstes dekrementieren beide Objekte den Bestand. Da beide Objekte von einer gleich hohen positiven Zahl ausgehen, kann es passieren, dass durch die parallelen Operationen der Bestand negativ wird.

Zur Verhinderung dieser Inkonsistenz werden Transaktionen eingeführt, die z.B. das Lesen und das Schreiben der Objekte zu jeweils einer Aktion zusammenfassen. Während einer solchen Transaktion ist dann jeglicher Zugriff auf die bearbeiteten Daten von anderen Seiten unterbunden. Ein Ausführen von beiden Operationen in einer Transaktion führt also dazu, dass nie ein inkonsistenter Zustand auftreten kann.

Wie oben erwähnt, wird eine Transaktion immer erfolgreich beendet oder alle zu einer Transaktion zusammengefassten Operationen rückgängig gemacht. Für die Transaktionssteuerung sind drei Kommandos notwendig:

- *begin:* Startet eine Transaktion.
- *commit:* Beendet eine Transaktion erfolgreich.
- *rollback:* Stellt den Zustand zu Beginn der Transaktion wieder her.

Tiefer soll hier nicht auf das allgemeine Thema Transaktionen eingegangen werden. Eine Einführung in das Thema unter besonderer Berücksichtigung der EJB-Technologie bietet z.B. [DENN00].

5.2 Transaktionen in der EJB-Architektur

5.2.1 JTS und JTA

Für das Transaktionsmangement hat Sun Microsystems zwei sich ergänzende Spezifikationen definiert. Dies ist zum einen das Java Transaction API (JTA), das die programmiertechnische Schnittstelle zum eigentlichen Transaktions-Service definiert. Zum anderen der Java Transactions Service (JTS), der diesen Transaktions-Service darstellt. Ein EJB 1.1-konformer Container muss die Schnittstelle des JTA anbieten und den Dienst des JTS implementieren.

5.2.2 Flache Transaktionen

Die EJB-Architektur sieht lediglich flache, nicht geschachtelte Transaktionen vor. Das bedeutet, dass immer nur eine Transaktion zurzeit ausgeführt werden kann. Vor dem Start einer neuen Transaktion muss eine bereits laufende Transaktion beendet werden.

5.2.3 Transaktionsattribute

Über sog. Transaktionsattribute im Deployment-Descriptor wird das Verhalten des Containers bezüglich Transaktionen gesteuert. Durch das Feld *trans-attribute* im Deployment-Descriptor wird dieses Verhalten entweder für alle Methoden oder für jede einzelne Methode einer Bean festgelegt. Hier können folgende Transaktionsattribute gewählt werden:

- *NotSupported*: Diese Methode unterstützt kein transaktionales Verhalten und kann z.B. nicht von Methoden, die in einer Transaktion ausgeführt werden müssen (*Required*), benutzt werden.

- *Required*: Diese Methode muss in einer Transaktion ausgeführt werden. Wird die Methode nicht in einer laufenden Transaktion aufgerufen, so kapselt der Container der Aufruf in einer neuen Transaktion.
- *Supports*: Diese Methode muss nicht, kann aber in einer Transaktion ausgeführt werden. Erfolgt der Aufruf in einer bestehenden Transaktion, so wird diese genutzt.
- *RequiresNew*: Für diese Methode muss eine neue Transaktion gestartet werden, die nach dem Methodenaufruf beendet wird. Diese Transaktion steht nur dieser Methode zur Verfügung.
- *Mandatory*: Diese Methode muss in einer durch einen Client gestarteten Transaktion ausgeführt werden. Besteht beim Aufruf dieser Methode keine Transaktion, so gibt der Container dem Client eine entsprechende Fehlermeldung zurück.
- *Never*: Diese Methode darf nicht in einer durch einen Client gestarteten Transaktion ausgeführt werden. Der Container reagiert bei einem solchen Versuch mit einer entsprechenden Fehlermeldung.

5.2.4 Implizite und explizite Transaktionssteuerung

In der EJB-Architektur unterscheidet man zwischen impliziten und expliziten Transaktionssteuerung. Im Falle von impliziten Transaktionssteuerung übernimmt der Container die Transaktionssteuerung. Bei expliziten Transaktionen wird die Transaktionssteuerung mit den oben genannten Kommandos vom Client (bzw. einer Enterprise Bean in der Rolle eines Clients) übernommen.

5.3 Implizite Transaktionssteuerung

Da das Transaktionsverhalten bei der impliziten Transaktionssteuerung vom Application Assembler definiert und vom Container gesteuert wird, braucht weder der Bean Provider noch der Entwickler des Clients Code zur Transaktionssteuerung in der Enterprise Bean zu implementieren. Hierdurch wird die von der EJB-Spezifikation vorgesehene Rollen- und Aufgabentrennung nicht verletzt und die Bean bleibt in Umgebungen mit unterschiedlichen Transaktionsanforderungen einsetzbar.

Der Application Assembler definiert im Deployment-Descriptor mit Hilfe der oben genannten Transaktionsattribute das Transaktionalsverhalten der Anwendung. Hier definiert er z.B. welche Methoden zu einer Transaktion zusammen gefasst werden. Die Transaktionssteuerung wird dann vom Container anhand dieser Attribute vorgenommen.

5.4 Explizite Transaktionssteuerung

Bei der expliziten Transaktionssteuerung wird die Transaktionssteuerung vom Client (der z.B. auch eine aufrufenden Enterprise Bean sein kann) mit Hilfe des Java Transaction API (JTA) übernommen. Dies hat zur Folge, dass das Transaktionsverhalten vom Bean Provider oder Entwickler des Clients definiert und implementiert werden muss und die Bean so nicht mehr unter unterschiedlichen Transaktionsanforderungen eingesetzt werden kann. Der Einsatz von expliziten Transaktionen macht nur Sinn, wenn ein bestimmtes Transaktionsverhalten für eine Operationen einer Bean (und nicht für die gesamte Anwendungslogik) zwingend notwendig ist.

5.5 Fehlerreaktion transaktionaler Objekte

Transaktionale Objekte sind Objekte, die an einer Transaktion teilnehmen. Sie können keine Transaktion starten oder beenden, aber bei einem Fehler ein Rollback erzwingen. Im Falle der EJB-Architektur sind die Enterprise JavaBeans diese transaktionalen Objekte.

Tritt ein Fehler innerhalb einer Transaktion in einer Methode einer Enterprise Bean auf, so kann diese Bean auf zwei Arten reagieren. Entwerder sie erzeugt eine Exception oder erzwingt mit Hilfe ihres entsprechenden Kontext-Objektes ein Rollback.

Eine erzeugte Exception wird an diejenige Bean, die die Methode aufgerufen hat, weiter geleitet. Diese kann dann wiederum entscheiden, ob sie ein Rollback setzt oder eine Exception erzeugt. Im Falle der impliziten Transaktionssteuerung setzt der Container ein Rollback, sobald die in dieser Hierarchie oberste Bean eine Exception erzeugt und bricht somit die Transaktion unter Herstellen des Ausgangszustandes erfolglos ab. Im Falle der expliziten Transaktionssteuerung wird die Exception an den Client weitergeleitet, der selbst entscheiden kann, ob ein entsprechendes Rollback gesetzt wird.

Erzwingt die Bean ein Rollback über ihr Kontext-Objekt, so muss die Transaktion nicht sofort abgebrochen werden, und es werden ggf. noch weiter Operationen der Transaktion ausgeführt. Enterprise Beans können über ihr entsprechendes Kontext-Objekt jedoch ein gesetztes Rollback erkennen und ggf. komplexe Operationen unterlassen, da sie keinen Einfluss mehr auf den Erfolg einer Transaktion nehmen können.

Kapitel 6

Beispielanwendung

Die vorgestellten Konzepte sollen an einer einfachen Beispielanwendung verdeutlicht werden.

Das Beispielprogramm, der Quellcode des Programmes sowie ein freier EJB-Server (JBoss) befinden sich auf der beiliegenden CD im Archiv *ejb-beispielanwendung.tgz*. Hinweise zur Installation und zum Starten der Anwendung enthält die Datei *readme.txt*.

In diesem Kapitel wird lediglich das allgemeine Design der Beispielanwendung besprochen. Für weitergehende Erläuterungen wird auf die Kommentare im Quellcode und den Quellcode selbst verwiesen.

6.1 Bibliotheksverwaltung

Als Beispiel wird eine einfache Bibliotheksverwaltung implementiert, die den Bücherbestand in einer Bibliothek verwaltet. Es soll möglich sein, Bücher mit einer Inventarnummer in die Bibliothek aufzunehmen, sie wieder aus der Bibliothek zu löschen, Bücher nach bestimmten Kriterien zu suchen sowie den Bestand der Bibliothek zu zählen.

Clientanwendungen sollen auf die Anwendungslogik der Bibliotheksverwaltung nur über Methoden von Session-Beans zugreifen können. Diese Session-Beans verwalten die als Entity-Beans implementierten Bücher in der Bibliothek.

Es wäre denkbar, weitere Funktionalitäten der Bibliothek zu implementieren. So könnte eine weitere Session-Bean den Verleih der Bücher an Bibliotheksbenutzer (die z.B. als Entity-Beans implementiert werden könnten) verwalten. Hierauf wurde aus Gründen der Übersichtlichkeit des Beispiels verzichtet.

Weiterhin nicht implementiert ist ein Sicherheitsmanagement. Jeder Client hat vollen

Zugriff auf den Bestand der Bibliothek.

6.2 Die Bücher

Die Bücher in der Bibliothek sollen per CMP automatisch vom Container persistent gehalten werden. Dies ist möglich, da die Bücher nur über sehr einfache Finder-Methoden gesucht werden sollen, die im Fall von CMP noch vom Container bereit gestellt werden können.

Ein Buch wird durch die folgenden Attribute beschrieben:

- die Inventarnummer (Primärschlüssel)
- den Titel des Buches
- den Namen des Autors
- den Verlag

Weiterhin sollen die Bücher nach folgenden Kriterien gesucht werden können:

- mit Hilfe der Inventarnummer
- alle Bücher eines Autors
- über den Buchtitel
- alle Bücher eines Verlags
- alle vorhandenen Bücher

Der Primärschlüssel eines Buchs in der Bibliothek ist seine ganzzahlige Inventarnummer. Hierzu wird zunächst eine Primärschlüsselklasse implementiert (*ckorn.studienarbeit.buch.interfaces.InvNo*).

Das Home-Interfaces der Bean (*ckorn.studienarbeit.buch.interfaces.BuchHome*) deklariert die entsprechenden Finder-Methoden sowie eine *create()*-Methode, die ein Buch mit allseinen Attributen initialisiert.

Im Remote-Interface (*ckorn.studienarbeit.buch.interfaces.BuchRemote*) werden lediglich die get- und set-Methoden, über die auf die Attribute der Bean zugegriffen werden kann, sowie eine Methode, die eine einfache Stringrepräsentation des Buches liefert, deklariert.

In der eigentlichen Bean-Klasse (*ckorn.studienarbeit.buch.bean.BuchBean*) wird relativ wenig Anwendungslogik implementiert. Da der komplette Lebenszyklus der Bean und die Persistenzmechanismen vom Container verwaltet werden, werden die entsprechenden Methoden leer implementiert. Die Finder-Methoden werden in der Bean-Klasse hingegen nicht implementiert. Diese Funktionalität wird ebenfalls vom Container bereit gestellt. Lediglich die get- und set-Methoden, die *create()*-Methode, die Methode, die eine Stringrepräsentation erzeugt, sowie die Methoden, die den Entity-Kontext setzen und löschen sind hier zu implementieren.

Im Deployment-Descriptor wird festgelegt, dass die Bean per CMP verwaltet wird, welche Attribute der Bean vom Container persistent gehalten werden sollen und dass jeder Zugriff auf die Bean vom Container in einer Transaktion gekapselt werden muss.

6.3 Die Bibiothek

Die Verwaltung des Bibliotheksbestandes übernimmt die Session-Bean *BibliothekBean*. Sie wird als zustandslose Session-Bean implementiert, da keine ihrer Methoden den internen Zustand der Bean ändern muss.

Diese Session-Bean kapselt alle Zugriffe auf die Bücher in der Bibliothek und bietet einem Client folgende Funktionalität an:

- ein Buch der Bibliothek hinzuzufügen
- ein Buch aus der Bibliothek zu löschen
- alle Bücher der Bibliothek aufzulisten
- alle Bücher eines Autors aufzulisten
- die Bücher in der Bibliothek zu zählen

Da es sich um eine zustandslose Session-Bean handelt, wird in dem Home-Interface (*ckorn.studienarbeit.bibliothek.interfaces.BibliothekHome*) lediglich eine *create()*-Methode ohne Parameter deklariert.

Im Remote-Interface (*ckorn.studienarbeit.bibliothek.interfaces.BibliothekRemote*) sind die Methoden deklariert, die einem Client die oben genannte Funktionalität zur Verfügung stellen.

Die eigentliche Bean-Klasse (*ckorn.studienarbeit.bibliothek.bean.BibliothekBean*) implementiert diese Methoden, die auf die als Entity-Beans implementierten Bücher zugreifen. Da es sich um eine zustandslose Session-Bean handelt, werden die Methoden, die den Lebenszyklus und den Session-Kontext verwalten, leer implementiert.

Im Deployment-Descriptor wird für diese Bean lediglich die Angabe gemacht, dass sie zustandslos ist und das jeder Zugriff auf eine Methode der Bean in einer vom Container gesteuerten Transaktion zu erfolgen hat.

6.4 Ein einfacher Client

Zum Testen der Beispielanwendung steht ein simpler Client zur Verfügung (*ckorn.studienarbeit.client.BibliotheksClient*), der auf die Bibliothek über entsprechende Methoden der Session-Bean zugreift.

Um den Client möglichst einfach zu halten, bietet dieser keine Möglichkeit der Benutzerinteraktion. Der Client fügt lediglich ein paar Bücher in die Bibliothek ein, gibt die Anzahl der Bücher in der Bibliothek aus, Listet alle Bücher eines Autors und löscht zum Schluss wieder alle Bücher aus der Bibliothek.

Kapitel 7

Ausblick und abschließende Bemerkungen

7.1 Neuerungen in EJB 2.0

Die Version 2.0 der EJB-Spezifikation steht zurzeit kurz vor ihrer endgültigen Freigabe. Der derzeitige Status von EJB 2.0 wird von Sun als 'Proposed Final Draft' bezeichnet. Man kann davon ausgehen, dass sich in der endgültigen Version keine grundlegenden Änderungen mehr ergeben.

Die wichtigsten Neuerungen in EJB 2.0 sind:

- Ein EJB 2.0 konformer Container muss das API des JMS (Java Massaging Service) bereitstellen.
- Mit der Version 2.0 wird ein weiterer Bean-Typ eingeführt, sogenannte Message-Driven-Beans.
- Die Container-Managed Persistance für Entity-Beans soll durch eine Schnittstelle zu Persistenz-Managern von Drittanbietern verbessert werden.
- Es wird eine Abfragesprache (Enterprise JavaBean Query Language (EJB QL)) analog zu SQL eingeführt.

7.1.1 Message-Driven-Beans und JMS

Mit Hilfe von Message-Driven-Beans ist es möglich, bestimmte Aktionen auf einem EJB-Server an asynchrone Nachrichten zu koppeln. Eine Message-Driven-Bean implementiert das Interface *javax.jms.MessageListener* und stellt somit eine Nachrichtensenke im Sinne des Java Messaging Service (JMS) dar. Die Message-Driven-Bean

benutzen dann die Anwendungslogik von Session-Beans, um Aktionen auf dem Server auszuführen. Nur über diesen neuen Typ von Beans ist es möglich, Aktionen auf dem Server an Events des JMS zu koppeln. Session- und Entity-Beans dürfen das API des JMS nicht nutzen.

Da Clients lediglich über den JMS mit der Bean kommunizieren, besitzt eine Message-Driven-Beans kein Home- und kein Remote-Interface. Weiterhin speichern sie keinen Conversational-State und stehen somit, wie zustandslose Session-Beans, mehreren Clients (für diese transparent) zur Verfügung. Neben dem oben erwähnten Interface *javax.jms.MessageListener* muss eine solche Enterprise Bean zusätzlich das Interface *javax.ejb.MessageDrivenBean* implementieren.

7.1.2 EJB QL

Die Abfragesprache EJB QL dient dazu, Entity-Beans in Anwendungen, die CMP nutzen, zu suchen und zu selektieren. EJB QL ist eine deklarative Sprache, die sich an den Standard SQL92 anlehnt bzw. eine Untermenge dessen ist. So ist es möglich, dass z.B. Session-Beans auf einem höheren Abstraktionslevel auf eine Datenbank und die darin gespeicherten Entity-Beans Zugriff haben. Somit können Beans unabhängig von der Abfragesprache der zu Grunde liegenden Datenbank gesucht werden. Außerdem wird es so ermöglicht, nach Entity-Beans mit Hilfe von Wildcards, über einen bestimmten Wertebereich ihrer Attribute oder mit Hilfe von boolschen Verknüpfungen ihrer Attributwerte zu suchen. Außerdem wird die Implementation von komplexeren Finder-Methoden im Fall von CMP mit Hilfe von EJB QL ermöglicht.

Die Sprache EJB QL ist in [SUN00] vollständig definiert und hat eine an SQL angelehnte Syntax. Die Tabellennamen in SQL entsprechen den Namen der Entity-Beans und die Feldnamen einer Tabelle entsprechen den persistenten Attributen einer Entity-Bean. Diese Ausdrücke werden dann von dem Persistenzmanager zu einer Zielspache (z.B. SQL) kompiliert und auf die Datenbank angewendet.

7.2 Praktische Erfahrungen

7.2.1 Performanz von EJB-Anwendungen

Die Flexebilität, die durch ein Komponentenmodell wie die EJB-Architektur erreicht wird, bedingt einen gewissen Overhead, der EJB-Anwendungen im Vergleich zu maßgeschneiderten Anwendungen für genau ein Einsatzgebiet weniger performant macht. Dies liegt zum einen an der Kommunikation über Protokolle wie RMI oder IIOP, zum anderen werden eine Vielzahl von Objekten pro Bean benötigt. Hierzu zählen unter anderem die Home- und Remote-Objekte, das Bean-Objekt selbst sowie die Skeleton- und Stub-Objekte, die für die Netzwerkkommunikation benötigt werden.

Diese Objekte referenzieren sich oft untereinander. Ein Methodenaufruf muss immer zuerst über den RMI-Stub, das RMI-Skeleton, das Home- oder Remote-Objekt an die Bean weiter geleitet werden. Weiterhin tragen Transaktionen, die Suche über den Namens- und Verzeichnisdienst und das Passivieren und Aktivieren dazu bei, dass EJB-Anwendungen vergleichsweise unperformant werden.

Um die Kommunikation über das Netz und die Gesamtanzahl der Klassen einer EJB-Anwendung zu begrenzen, sollten Session-Beans möglichst wenige aber mächtige Operationen abbilden. So werden die Clientaufrufe dezimiert, was die Datenmenge, die per RMI oder IIOP kommuniziert werden muss, beschränkt.

Weiterhin sollten Entity-Beans möglichst große Datenobjekte abbilden, also grob granuliert sein, was ebenfalls die Anzahl der Klassen dezimiert und so zu einer Performencesteigerung beiträgt.

Eine Reihe von Containern optimieren den Zugriff auf Beans, die sich in derselben Virtual Machine, wie der Client befinden, indem sie die Kommunikation per RMI umgehen und somit einen schnelleren Zugriff auf die Methoden einer Bean erlauben.

7.2.2 EJB-Server-Übersicht

Auf dem Markt gibt es inzwischen eine Vielzahl von freien und kommerziellen EJB-Servern. Diese werden (wie oben bereits erwähnt) oft als Application-Server im Verbund mit einer Servlet- und JSP-Engine, sowie einem Webserver und ggf. weiteren Diensten angeboten.

Die oben erwähnte Beispielanwendung lies sich problemlos von einem frei verfügbarem (JBoss) zu einem kommerziellen EJB-Server (IBM WebShere) portieren. Dies mag u.a. daran gelegen haben, dass die in Kapitel 6 beschriebene Beispielanwendung ein relativ simples Testprogramm ist, das längst nicht alle Aspekte und Möglichkeiten abdeckt, die die EJB-Spezifikation bietet.

Die Unterschiede der einzelnen EJB-Server und -Container liegen oft in ihrer Performanz und Skalierbarkeit, sowie in Zusatzmöglichkeiten, die nicht von der EJB-Spezifikation abgedeckt sind. Weiterhin unterscheiden sich die angebotenen J2EE-konformen Application-Server durch die mitgelieferten Tools, die den Systemadministrator bei der Überwachung einer EJB-Anwendung oder den Bean-Deployer bei der Installation der Beans unterstützen. Manche EJB-Server bieten ferner die Möglichkeit, Beans zur Laufzeit auf entfernte Container desselben Herstellers zu verschieben, um das System dynamisch zu skalieren.

Da ein Debugging von Enterprise JavaBeans ähnlich dem von Servlets oder Java Server Pages ein Problem darstellt, weil die Beans auf eine Laufzeitumgebung angewiesen sind, bieten einige Hersteller eine Kombination aus Entwicklungsumgebung und Application-Server an[1], die ein Debugging und das Erstellen und Deployment

[1] Als Beispiel sei hier die Kombination der Produkte IBM WebShere und IBM VisualAge for

der Beans durch entsprechende Zusammenarbeit der Komponenten unterstützen.

7.3 Abschließende Bewertung

Sun Microsystems hat mit der Enterprise JavaBean-Architektur ein stabiles und in der Praxis gut nutzbares Framework für komponentenbasierte, verteilte Anwendungen geschaffen. Die Verbesserungen, die der Versionsprung von 1.0 auf 1.1 gebracht hat und der Versionssprung von 1.1 auf 2.0 erhoffen lässt, zeigen, dass diese Technologie von Sun konstant weiterentwickelt wird und nützliche Änderungen einfließen.

Die Anzahl der verfügbaren EJB-konformen Server zeigt ebenso wie ein sich langsam etablierender Markt für EJB-Komponenten, dass diese Technologie auch vom Markt akzeptiert wird.

Neben kleinen Ungereimtheiten und Lücken in der Spezifikation ist die mangelnde Performanz von EJB-Anwendungen ein Manko. Dies ist der Preis für die Flexebilität, die diese Architektur bietet.

Java genannt.

Literaturverzeichnis

[DENN00] Denninger, Stefan / Peters, Ingo, Enterprise JavaBeans, Addison-Wesley, 2000.

[GAMM96] Gamma, Erich et al., Entwurfsmuster, Addison-Wesley 1996.

[HALT01] Halter, Steven / Munroe, Steven, Enterprise Java Performance, Prentice Hall, 2001.

[JBOS01] Boone, Kevin et al., JBoss Manual, http://www.jboss.org/documentation/HTML/index.html, 2001.

[MONS99] Monson-Haefel, Richard, Enterprise JavaBeans, O'Reilly, 1999.

[ORFA98] Orfali, robert / Harkey, Dan / Edwards, Jeri, Instant CORBA, Addison-Wesley, 1998.

[ROMA99] Roman, Ed, Mastering Enterprise JavaBeans and the Java 2 Platform, Enterprise Edition, Wiley, 1999.

[SUN99] Sun Microsystems Inc., Enterprise JavaBeans Specification v.1.1, Sun Microsystems Inc., 1999.

[SUN00] Sun Microsystems Inc., Enterprise JavaBeans Specification v.2.0, Sun Microsystems Inc., 2000.

[TUT01] Sun Microsystems Inc., Java Tutorial, http://www.javasoft.com/docs/books/tutorial/, 2001.

[VOSS99] Vossen, Gottfried, Dr., Datenbankmodelle, Datenbanksprachen und Datenbankmanagement-Systeme, Oldenburg, 1999.

[WAN00] Wanner, Gerhard / Koch, Volker, Objektpersistenz in EJB-basierten Applikations-Servern, Java Spekturm, Ausgabe 5/2000, 2000.