Diplomarbeit

Integration von Corso Spacebased Computing in J2EE

ausgeführt am

Institut für Computersprachen
der technischen Universität Wien

unter der Leitung von
A.o. Univ. Prof. Dr. tech. Dipl. Ing. eva Kühn

durch

Johannes Marchart
Matr. Nr.: 9226417
Felixgasse 95, 1130 Wien

Wien, Mai 2004

Kurzfassung

Diese Arbeit beschreibt die Integration der Corso ™ Middleware in das J2EE Framework von SUN Microsystems ™. Corso Middleware ist ein MOM (Message Oriented Middleware) basierendes Softwareprodukt der Firma Tecco ™ AG, das erfolgreich und vorwiegend im Enterprise-Segment heterogener IT Landschaften eingesetzt wird.

Der in den letzten Jahren aufwärtsstrebende Trend der Architekturen im Bereich der Server-Softwarekomponenten und deren Einsatzgebiete im Enterprise Softwaremarkt waren der Anstoß zu dieser Arbeit und zur Integration von Corso in J2EE.

Die Spezifikation von J2EE sieht eine Integration von EIS (Enterprise Informations-Systemen) allgemeinster Art in der Spezifikation von JCA (Java Connector Architektur) vor. Mit Implementierung der JCA Spezifikation in dieser Arbeit liegt dem J2EE Framework ein Ressource Adapter vor, der in jedem beliebigen Applicationserver ausführbar ist und J2EE Komponenten die Türen zum „space based Computing“ öffnet und Vorteile von Corso wie Failover, Scaleability und Faulttolerance nutzbar machen.

Anhand einer Migration einer verteilten Corso Applikation zu einer Corso J2EE Applikation wird im Abschluss dieser Arbeit der Nutzen von Corso sichtbar und gibt Ideen zu weiterführenden Arbeiten.

Abstract

This paper describes the integration of Corso™ Middleware into the J2EE Framework von SUN Microsystems™. Corso Middleware is a software product of TECCO™ Corporation based on MOM (Message Oriented Middleware) which is successfully applied predominantly within the enterprise segment of heterogeneous IT landscapes.

The recent rapid development in the field of server software component architecture and its deployment in the enterprise server software market gave the impetus to this project and to the incorporation of Corso in J2EE.

The specification of J2EE provides for the integration of EIS (Enterprise Information Systems) in a most general form according to the specification of JCA (Java Connector Architecture). With the implementation of the JCA specification in this paper a resource adapter for the I2EE framework is provided which can be implemented on any application server . Thus, it opens the door to J2EE components for "space based computing" and for the utilization of Corso's advantages such as failover, scaleability and fault tolerance.

The final part of the paper visualizes the usefulness of Corso by means of the migration of a shared Corso application to a J2EE application and provides ideas for work leading further.

Danksagung

Diese Arbeit ist mit großer Unterstützung meiner Eltern und insbesondere von P. Daniela Woller entstanden, die mir gerade in der Abschlussphase meines Studiums viel Zuversicht und Kraft zur Fertigstellung dieser Arbeit gegeben hat.

Ich möchte mich herzlich bei meiner Betreuerin Dr. eva Kühn bedanken, die den Anstoß zu dieser Arbeit gab und mich auf dem Weg der Ausarbeitung stets unterstützend begleitet hat.

Andreas W. Schotten gilt besonderer Dank für die Geduld und berufliche Entlastung, die er mir innerhalb des letzten Jahres zukommen ließ.

Dank möchte ich auch allen Kollegen der Firma Tecco aussprechen, die mir stets mit Kritik und guten Ideen zur Seite standen.

Johannes Marchart

Inhaltsverzeichnis

1 Einleitung

1.1 Problemstellung

Die letzten Jahre sind von Architekturen im Bereich des Markplatzes von serverseitigen Softwarekomponenten geprägt. Kommerzielle Hersteller von Software bieten verschiedenste serverseitige Komponenten an und erlauben es den Applikationsentwicklern beliebige Komponenten unterschiedlicher Hersteller „von der Stange“ zu beziehen, zu kombinieren, daraus Applikationen zu entwickeln und in Applicationservern auszuführen.

Die Motivation dieser Arbeit war, das Paradigma von Corso™ Middleware – das spaced based Programming – in die Welt der Server Softwarekomponenten von J2EE zu integrieren. Neben dem Paradigma des „spacebased Programming“ leistet Corso auch einen führenden technologischen Schritt im Bereich des Verbindens heterogener IT Landschaften. Corso sorgt in seiner aktuellen Version durch das Konzept von „Programmiersprache&Co“ („Java&Co“, „C++&Co“, „.NET&Co“) und den für alle gängigen Betriebsysteme erhältlichen Corso Koordinationkernel bereits für starke Vernetzung heterogener Systeme, doch war die Integration von Corso in die Welt von J2EE und in die der Applicationserver ein weiterer wichtiger Schritt zur Kooperation von IT Systemen im Bereich des Business/Enterprise Segmentes.

1.2 Die Lösung

Die Integration von Corso in J2EE erfolgte über die JCA (Java Connector Architektur) Spezifikation, die Bestandteil der J2EE Spezifikation von Sun Microsystems ist. Die Umsetzung der JCA Spezifikation resultierte in einer Implementierung eines JCA Ressource Adapters, der in Applicationservern geladen wird und EJB- Komponenten ein Verbindungs- und Transaktions- Management zum Corso zur Verfügung stellt.

1.3 Aufbau dieser Arbeit

Kapitel 2 beschreibt in Kürze Corso und das Konzept des „spacebaced Programmings“

Kapitel 3 ist der J2EE Spezifikation und der Welt der Enterprise Java Beans (EJBs) gewidmet. Man gewinnt Einblick in das Konzept der serverseitigen Softwarekomponenten mit EJBs und erkennt das Zusammenspiel der Komponenten mit dem Applicationserver.

Kapitel 4 beschreibt die Spezifikation von JCA. Es werden die für in dieser Arbeit entstanden Ressource Adapter relevanten Themen der JCA Spezifikation erläutert und parallel dazu Codeausschnitte der Implementierung des Adapters gezeigt.

Kapitel 5 erläutert die Adaption einer bestehenden verteilten Corso Applikation in die Welt von J2EE und EJBs. Die Funktionalität der bestehenden Applikation wurde vom Präsentationlayer (GUI) entkoppelt, die Logik in Businessmethoden von EJBs verpackt, die EJBs und der Ressource Adapter im *jBoss* Applicationserver lauffähig gemacht.

Kapitel 6 bewertet die Lösung und beschreibt die Vorteile, die sich mit der Integration von Corso für die J2EE Welt ergeben.

Kapitel 7 gibt Ausblicke und Ideen für weiterführende Arbeiten

2 Grundlagen Corso

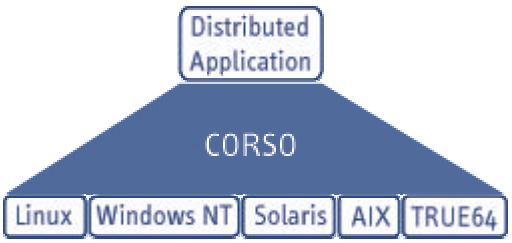
2.1 Einführung

CORSO (Co-ORdinated Shared Objects) basiert seit 1992 auf Forschungsarbeiten an der Technischen Universität Wien am Institut für Computersprachen und ist seit 1998 als führende kommerzielle Implementierung von Virtual Shared Memory [1] erhältlich.

Softwareentwicklung in heterogenen verteilten Systemen bringt stets zusätzliche Komplexität für den Entwickler in folgenden Bereichen mit sich:

- Lokation: Adressierung und Lokalisierung von Ressourcen
- Replikation: Verwalten von Daten Kopien an dislozierten Orten
- Transaktionen: Synchronisationsmechanismen konkurrierender Zugriffe auf verteilte Ressourcen
- Skalierbarkeit: Die Möglichkeit zur transparenten Erweiterung des Systems im Zuge von wachsenden Anforderungen und Belastungen an das Gesamtsystem
- Lastverteilung: Die faire und gleichmäßige Verteilung von Aufgaben an verteilte Komponenten
- Fehlersicherheit: Die Kompensation ausfallender Rechner und Persistenz von flüchtigen Daten

Corso ist eine Softwareschicht (Middleware) zwischen Applikationsschicht und Betriebsystem, die oben genannte Aspekte adressiert und soviel als möglich an Komplexität dieser Bereiche dem Entwickler abnimmt.



Corso läuft auf den in der Abbildung genannten Betriebsystemen und stellt ein Bindeglied für den Datenaustausch und die Synchronisation von verteilten Applikationen in heterogenen Betriebsystem- Umgebungen dar.

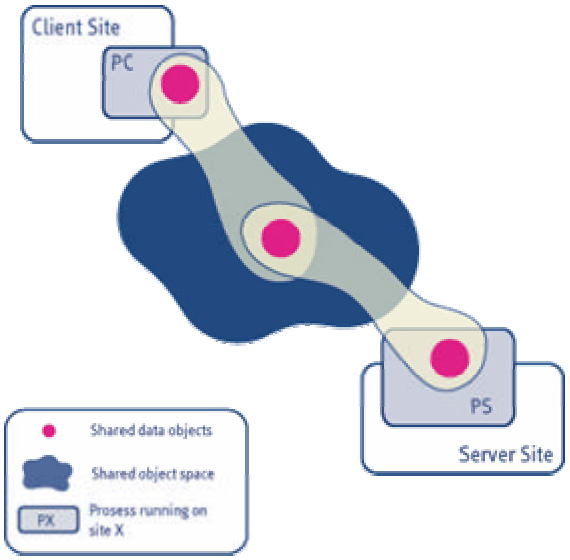
2.2 Virtual shared Memory

Ein Virtual shared Memory stellt einen konzeptionell hohen Abstraktionslevel für den Datenaustausch in verteilten Systemen dar. Es stellt einen allgemeinen verteilten Objekt Raum (object space) dar, auf den alle parallel verteilte Prozesse eine konsistente Sicht

haben. Die verteilten Daten Objekte (shared Data Objects) werden für Speicherung von Informationen, Kommunikation und Synchronisation von Prozessen verwendet.

Die Abstraktion findet in der Art statt, dass der Hauptspeicher jedes einzelnen Rechners um die Hauptspeicher aller anderen teilnehmenden Rechner erweitert wird, der Programmierer aber dabei von Caching- und Replikations – Implementierungen befreit ist.

Shared Data Objects erlauben ein Design von symmetrischen Applikations- Architekturen und vermeiden dabei jedoch typische Client/Server Engpässe.



Typische Zwei-Weg Kommunikation kann wie in obiger Abbildung mit einem Corso Datenobjekt implementiert werden. Der Client schreibt seine Anfrage in das verteilte Datenobjekt, das synchron vom Serverprozess gelesen wird. Das Resultat des Requests wird vom Server in dasselbe Objekt geschrieben, das vom Client synchron gelesen wird. Die Skalierbarkeit dieser Lösung resultiert in diesem Fall in einer Implementierung eines Pools von Request/Answer Objekten und dem Hinzufügen von beliebigen Servern, die konkurrierend aus dem Daten Pool Anfragen lesen und Antworten schreiben.

2.3 Das CORSO Koordinationsmodell

Das Corso Koordinationsmodell unterstützt:

- Zuverlässige Kommunikation durch gemeinsam verwendete Datenobjekte.
- Nebenläufigkeit, Parallelität und Verteilung durch CORSO Prozesse.
- Flexible Koordinationspatterns und Wiederehrherstellbarkeit durch das hoch entwickelte Transaktionsmodell.

Im CORSO Modell kommunizieren autonome Services über verteilte Datenobjekte. Diese Objekte werden mit einer OID (Object IDentification) im Netz eindeutig identifiziert. Objektdaten können strukturiert sein und Referenzen zu anderen Objekten beinhalten (Subobjekte).

Kommunikationsobjekte werden von Agenten verwaltet, welche für die Garantie der „konsistenten Sicht“ der Objekte gegenüber allen Prozessen verantwortlich sind. Objekte können innerhalb von Transaktionen gelesen und beschrieben werden. Diese Transaktionen sind atomar: Mehrere Aktionen auf einem Set von Objekten werden entweder vollständig oder gar nicht durchgeführt.

Die Nebenläufigkeit wird mit Hilfe von CORSO Prozessen realisiert, die parallel auf einem Rechner oder verteilt auf mehreren Rechnern laufen können. Diese Prozesse erfüllen eine bestimmte Aufgabe und werden entweder durch einen Systemprozess oder einen Thread realisiert. Sie können Referenzen auf Kommunikationsobjekte austauschen, wodurch diese Objekte zwischen den Prozessen geteilt werden. Ein Prozess repräsentiert einen Softwarevertrag zwischen CORSO und dem lokalen System, zu welchem der Prozess gehört. Ein Prozess kann hierbei als „relieable“ definiert werden. Solch ein zuverlässiger Prozess startet nach einem Systemfehler (Absturz) erneut und erhält automatisch auch alle seine Kommunikationsobjekte.

Objektzugriff erfordert Autorisierung. Es ist nur Prozessen, die eine Referenz auf ein Objekt haben erlaubt, diese zu lesen oder zu schreiben. Ein Prozess besitzt eine Referenz auf ein Objekt, wenn er entweder das Objekt kreiert hat, die Objektreferenz beim Prozessstart übergeben wurde oder das Objekt ein Subobjekt eines bereits bekannten Objektes ist. Das Konzept der „named objects“ erlaubt es Objekte innerhalb einer Domäne mit einem eindeutigen Namen zu identifizieren und zu referenzieren.

Mit Hilfe dieses Autorisierungsschemas entscheidet ein CORSO Agent auch, ob ein Objekt von einem Prozess noch referenziert wird oder nicht. Die Speicherbereinigung von Kommunikationsobjekten erfolgt in CORSO automatisch.

2.4 Der Corso Koordinations- Kernel

Die oben präsentierte Metapher vom CORSO Koordinationsmodell, wo Agenten verteilte Objekte verwalten und nebenläufige Serviceprozesse unter Verwendung dieser verteilten Objekten kooperieren, wird technisch wie folgt realisiert:

Ein lokaler Agent ist durch einen lokalen Systemprozess realisiert, der als Koordinations-Kernel (kurz: CoKe) bezeichnet wird. Alle CoKes formen den „agent space“. Koordination Kernels kommunizieren untereinander via IP über UDP oder TCP Sockets und verwenden in der Regel dieselbe konfigurierbare Portnummer.

Der CoKe administriert alle CORSO Prozesse und verteilten Objekte. Der konsistente Blick auf die verteilten Objekte wird mittels Verteilungsstrategien, basierend auf Replikationstechniken, realisiert. Nachrichten zwischen den einzelnen CoKes, die zusammen das verteilte VSM bilden, werden mittels eines End-to-End Kommunikationsprotokoll übertragen, welches die sichere Nachrichtenzustellung gewährleistet. Falls eine unzuverlässige Kommunikationsverbindung vorliegt, Nachrichten verloren gehen, mehrfach verschickt oder im Netzwerk dupliziert werden, ist die Applikation nicht beeinträchtigt. Es werden auch Systemfehler maskiert, wenn eine wiederherstellbare Verteilungsstrategie für Objekte und Prozesse angegeben wurde. In diesem Fall erfolgen Zustandsspeicherungen je nach Reliabilitätsklasse auf einer oder mehreren Dateien des Koordinationkernels, wie es auch in Datenbanksystemen üblich ist. Nach einem Systemfehler werden alle gespeicherten

Prozesse und Objekte durch den CoKe wiederhergestellt und die Kommunikation zu anderen CoKes wieder aufgenommen.

2.5 Applikationen mit Corso

CORSO Prozesse können in einer Programmiersprache geschrieben werden, zu der es eine bestehende CORSO Sprachanbindung gibt. Eine Programmiersprache, die um CORSO Eigenschaften erweitert ist, bezeichnet man als Programmiersprache&Co.

Zur Zeit werden C&Co, C++&Co, Java&Co und .net&Co unterstützt.

2.6 Die Corso Konsole

Eine weitere CORSO Komponente ist die CORSO Konsole, mit deren Hilfe verteilte Applikationen konfiguriert und administriert werden können. Diese Konsole ist ein spezielles Service, das sich zum CoKe verbindet. Alle Funktionen, die die Konsole offeriert, können auch mit Hilfe der Sprachanbindungen innerhalb eines Services ausgeführt werden. Die Konsole bietet weiters auch die Möglichkeit, gerade stattfindenden Aktivitäten im Space zu beobachten und die aktuelle Konfigurationseinstellung des Cokes zu zeigen.

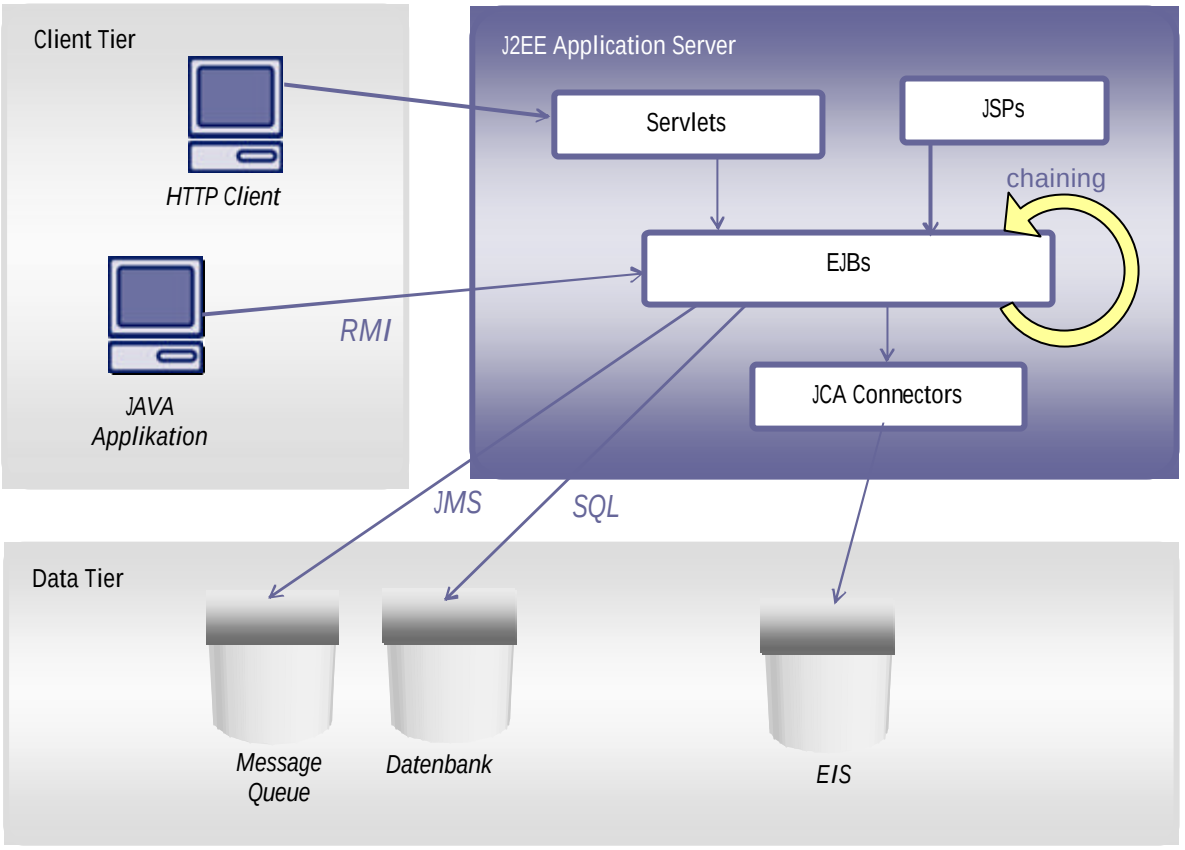
3 Enterprise Java Beans (EJB)

Der EJB Standard ist eine komponenten-basierende Architektur für serverseitige Java Softwareentwicklung und ermöglicht EJB Komponenten in beliebigen J2EE konformen Applicationserver zur Ausführung zu gelangen. EJB ist ein Teil der J2EE (Java 2 Enterprise Edition) Spezifikation von Sun Microsystems, die ein plattformunabhängiges Framework für Java Enterprise Softwareentwicklung bildet. Die Standards der J2EE Spezifikation formen zusammen eine robuste Suite von Middleware-Diensten, die heute eine starke Vereinfachung in der Softwareentwicklung ermöglichen. Zu den J2EE Standards zählen:

- **EJB** - Enterprise Java Beans: Serverseitige Komponenten und deren Integration in Java Applicationservern
- **RMI** - Java Remote Method Invocation: Kommunikation zwischen verteilten Java Objekten
- **JNDI** - Java Naming and Directory Interface: Zugriff auf Namens- und Verzeichnisdienste
- **JDBC** – Java Database Connectivity: API für den Zugriff auf relationale Datenbanken
- **JTA** – Java Transaction API, JTS – Java Transaction Service: API für transaktionalen Zugriff von Komponenten
- **Java Servlets**: Request/Response orientierte Programme, die die Funktionalität eines Webservers erweitern
- **JSP** – Java Server Pages: Serverseitige Skripts die eine Trennung von Design und Logik in der Webseiten Entwicklung ermöglichen
- **JCA -** J2EE Connector Architecture: Adapter zum Zugriff auf Enterprise Information Systeme (EIS)
- **JAXP** - Java XML Parsing: Parsen von XML Daten
- **JASS** – Java Authentication and Authorization Service
- **JMS** – Java Messaging Service: Zugriff auf message orientierte Middleware (MOM) Systeme, wie MQ Series oder MSMQ

3.1 Übersicht über J2EE

Folgende Abbildung zeigt eine schematische Übersicht über die Java 2 Enterprise Edition (J2EE) Plattform und die verwendeten Dienste bzw. Standards. EJB Komponenten implementieren gemeinsam die Business Logik und sind durch Schnittstellen von den Schichten der Präsentation (Client- und Presentationtier) und dem Datenhaushalt (Datatier)



entkoppelt.

3.1.1 EJB Typen

EJB 2.0 definiert drei unterschiedliche Typen von Enterprise Beans

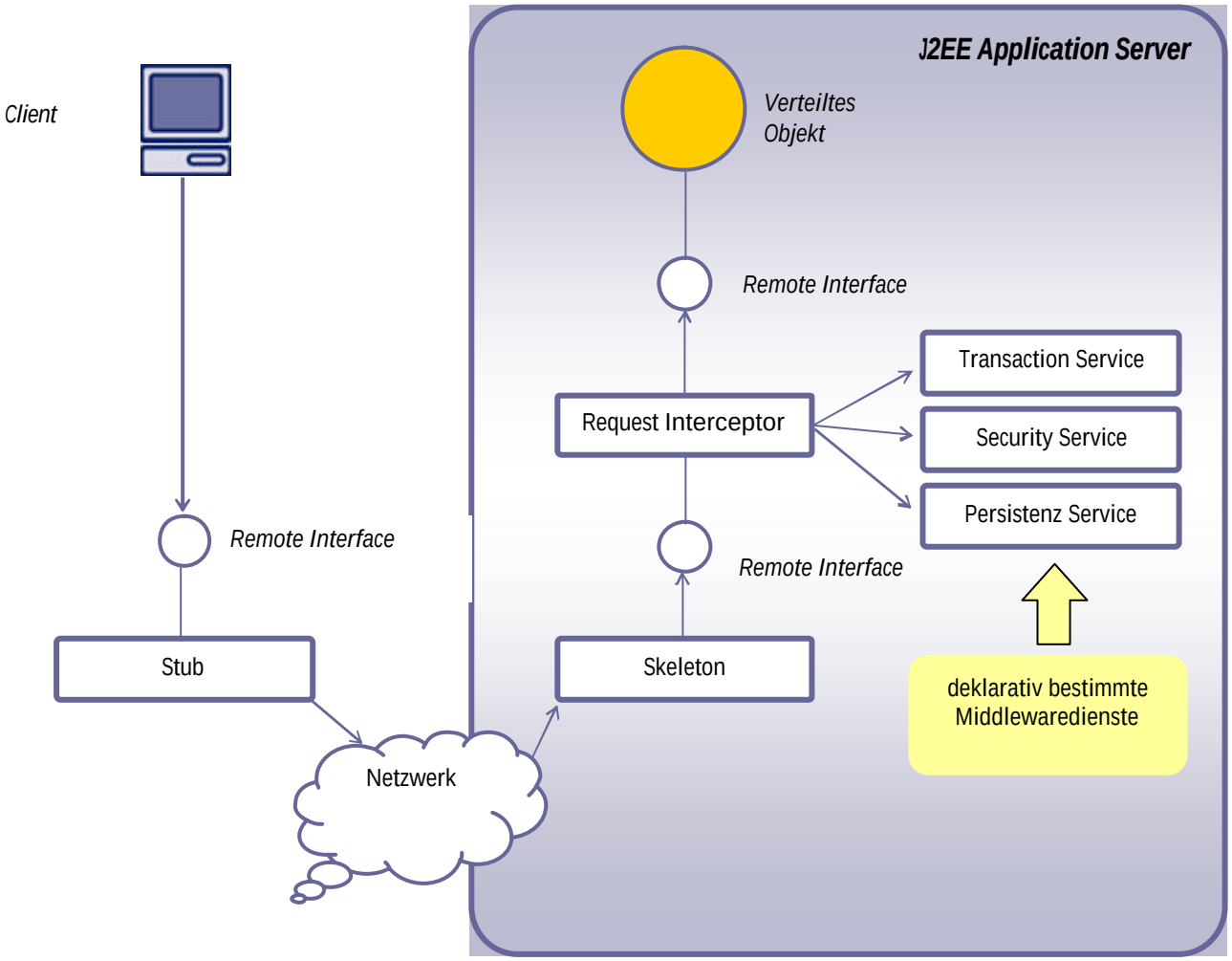
- Entity Beans. Entity Beans modellieren Business Daten. Sie repräsentieren persistente Daten Objekte, die ihre Informationen aus relationalen Datenbanken lesen und zwischenspeichern.

- Session Beans: Session Beans modellieren Business Prozesse, Business Regeln Algorithmen und Workflows. Sie können Anfragen an Legacy Systeme oder Enterprise Information Systeme (EIS) stellen oder auch andere Enterprise Beans aufrufen (chaining).

- Message driven Beans: Message Driven Beans wurden eingeführt, um asynchrone nicht blockierende Aufrufe von Enterprise Beans zu ermöglichen. Der Aufruf einer Session Bean via RMI-IIOP birgt einige Nachteile in sich, die durch einen asynchronen „message“ - orientierten Aufruf kompensiert werden können.

3.1.2 Aufruf verteilter Objekte (EJBs)

Ein Client ruft den lokalen Stub auf, der das clientseitiges Proxy Objekt darstellt. Der Stub maskiert die Netzwerkkommunikation und transformiert Parameter in ihre Netzwerkrepräsentation. Er ruft das Skeleton über das Netzwerk auf, welches das serverseitige Proxy Objekt darstellt. Die Delegation der Anfrage an das gewünschte verteilte Objekt erfolgt schlussendlich durch den Request Interceptor. Er erweitert die reine Business Logik der Enterprise Bean um Middleware Funktionalität wie Transaktionsmanagement oder Security Dienste. Das Verhalten des „Request Interceptors“ und der Middlewaredienste wird deklarativ durch eine Deployment Deskriptor Datei bestimmt. Dies bezeichnet man auch als Verwendung implizierter Middleware. Der Unterschied zum Einsatz von traditioneller Middleware wie CORBA, DCOM oder RMI ist, dass der Programmcode nicht mit Code gegen eine Middleware API vermischt ist. Vermengung von Businesslogik und Middleware API bedeutet zusätzliche Komplexität im Code, erschwert die Wartbarkeit und führt zu geringer Wiederverwendbarkeit bei Austausch der Middleware.



3.1.3 Gegenüberstellung von Verwendung expliziter/impliziter Middleware:

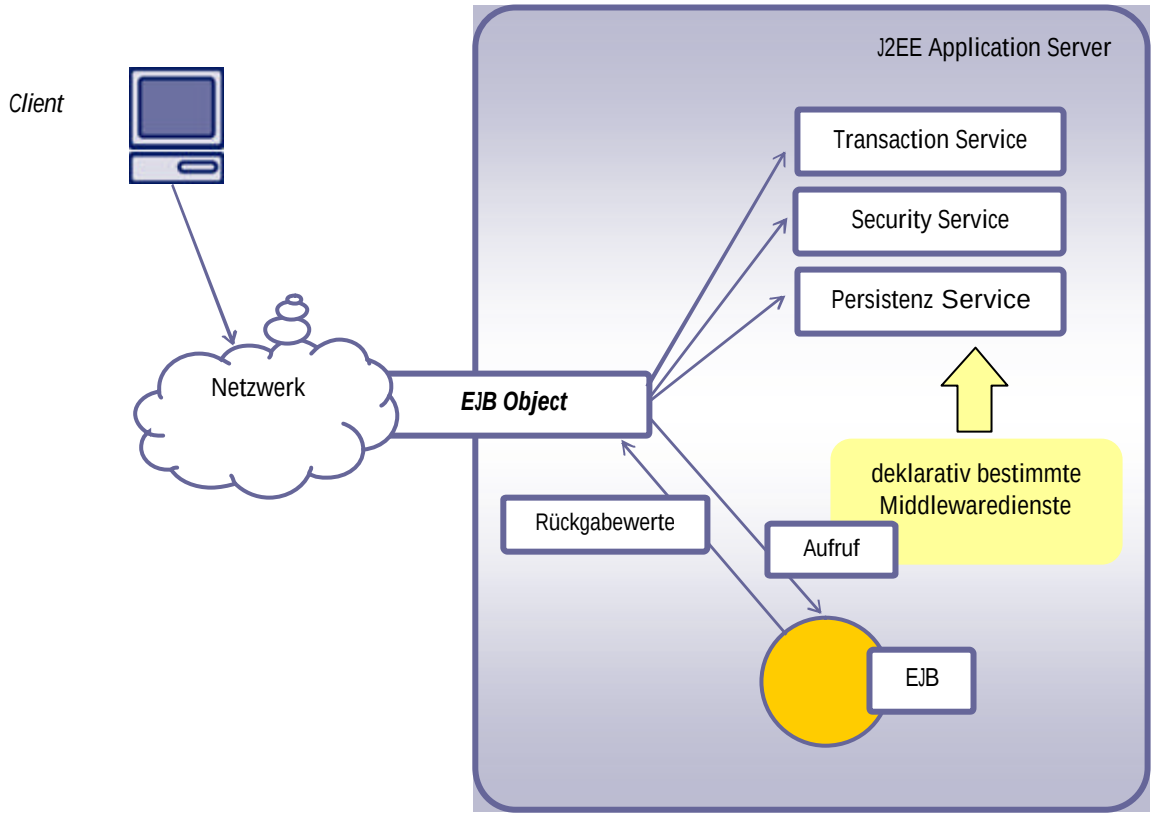
Source Code mit Verwendung expliziter Middleware	Source Code mit Verwendung impliziter Middleware
1. Middleware API: Führe Sicherheits-Check aus 2. Middleware API: Erzeuge eine Verbindung zur Datenbank 3. Middleware API: Öffne eine Transaktion 4. Middleware API: Lade Daten aus einer Tabelle (Abfrage) **5. Business Logik: Rechenoperation** 6. Middleware API: Speichere Daten in Tabelle 7. Middleware API: Schließe Transaktion	**1. Business Logik: Rechenoperation**

Middlewaredienste, die durch den J2EE Container (Request Interceptor) zur Verfügung gestellt werden umfassen:

- Implizites verteiltes Transaktionsmanagement mit JTA
- Implizite Security: Authentifikation und Authorisierung
- Implizites Ressourcen Management für Threads, Sockets, Datenbanken Verbindungen und Instanzen von Enterprise Beans
- Implizite Datenpersistenz: Automatisches Speichern von Datenobjekten in relationale Datenbanken
- Impliziter verteilter Zugriff auf Objekte: Enterprise Beans sind automatisch via Netzwerk aufrufbar
- Implizites Threading: Konkurrierende Anfragen werden automatisch aufgelöst bez. serialisiert
- Implizite Lokalisierungs-Transparenz
- Implizites Monitoring

3.1.4 Das EJB Objekt

Der J2EE Container für EJBs stellt in der Kommunikation zwischen Client und den EJB Komponenten eine Schnittstelle dar, die oben aufgezählte Dienste einbringt. Der Request Interceptor ist durch das ***EJB Object*** repräsentiert. Das EJB Object ist ein „intelligentes Objekt“ das die Middleware Logik des Containers ausführt bevor die tatsächliche Methoden der Bean Instanz aufgerufen werden. EJB Objects replizieren und präsentieren alle Business Methoden der EJBs und delegieren Client Anfragen schlussendlich an die Bean selbst.



EJB Objects sind containerspezifisch, da jeder Container Middleware Funktionalität unterschiedlich behandelt und unterschiedliche Qualität von Diensten anbietet. EJB Objects klonen alle Business Methoden, die EJBs anbieten durch das „remote interface“, das der Bean Entwickler schreibt. Das „remote interface“ muss vom ***javax.ejb.EJBObject*** abgeleitet sein, dass wiederum von *java.rmi.Remote* ableitet. Somit sind EJB Objects über RMI-IIOP von anderen Java VMs aufrufbar. EJB Objects werden nicht direkt instanziert sondern über eine EJB Object Factory referenziert. Die Factory ist für die Instanzierung und Zerstörung der EJB Objects verantwortlich. Die EJB Spezifikation bezeichnet diese Factory als „Home Objects“. Home Objects werden vom Container automatisch generiert in dem der Bean Entwickler das home interface spezifiziert. Home Interfaces definieren wie EJB Objects gefunden, erstellt und wieder zerstört werden und müssen von ***javax.ejb.EJBHome*** abgeleitet sein.

Der Aufruf über RMI-IIOP und das Home Interface sieht wie folgt aus:

- Der Client ruft den lokalen Stub auf
- Der Stub „marshaled“ die Parameter in die Netzwerkrepräsentation
- Der Stub geht über das Netzwerk zum Skeleton
- Das Skeleton „demarshaled“ die Parameter
- Das Skeleton ruft das EJB Object auf
- Das EJB Object ruft orthogonale Middlewareservices auf
- Das EJB Object delegiert die Business Methoden an die EJB

Diese Schritte bedeuten einen massiven Rechen- und Kommunikationszusatzaufwand insbesondere wenn der Aufruf aus der selben VM erfolgt, in der sich auch die EJB befindet. Mit EJB 2.0 wurden daher weitere Methoden eingeführt eine EJB schneller und effizienter aufzurufen. Anstelle des EJB Objects tritt das „Local Object“ . Local Objects implementieren das „local interface“ .Die Schritte des Stub's, Marshalling's, der Netzwerkkommunikation und des Skeleton's fallen weg und erlauben schlankere schnellere „cross Bean“ Aufrufe. Analog zum „home interface“, das für die Generierung der Factory zur Erzeugung von EJB Objects verantwortlich ist gilt es das „local home interface“ zu erstellen, mit dessen Hilfe der Applicationserver „Local Home Objects“ implementiert. Das „local interface“ ist von ***javax.ejb.EJBLocalObject*** abgeleitet, das „local home interface“ von ***javax.ejb.EJBLocalHome***.

Übersicht über Definitionen:

Bezeichnung	
EJB	Enthält nur Business Methoden, netzwerklos, singlethreaded
Remote Interface	Interface, das nur die Business Methoden des EJB aufzählt.von ***javax.ejb.EJBObject*** abgeleitet.
EJB Object	Vom Container des Applicationserver erzeugte Implementation des Remote Interfaces, fügt Middleware Dienste ein, delegiert an EJB
Home Interface	Interface für die EJB Object Factory, von ***javax.ejb.EJBHome*** abgeleitet
Home Object	Implementation des Home Interfaces
Local Interface	High-performing Version des Remote Interfaces, von ***javax.ejb.EJBLocalObject*** abgeleitet
Local Object	HIgh-performing Version des EJB Objects
Local Home Interface	High-performing Version des Home Interfaces, von

	javax.ejb.EJBLocalHome abgeleitet
Local Home Object	High-performing Version des Home Objects

3.1.5 Der Aufruf von EJBs

Grundsätzlich gibt es 2 Typen von EJB Clients:

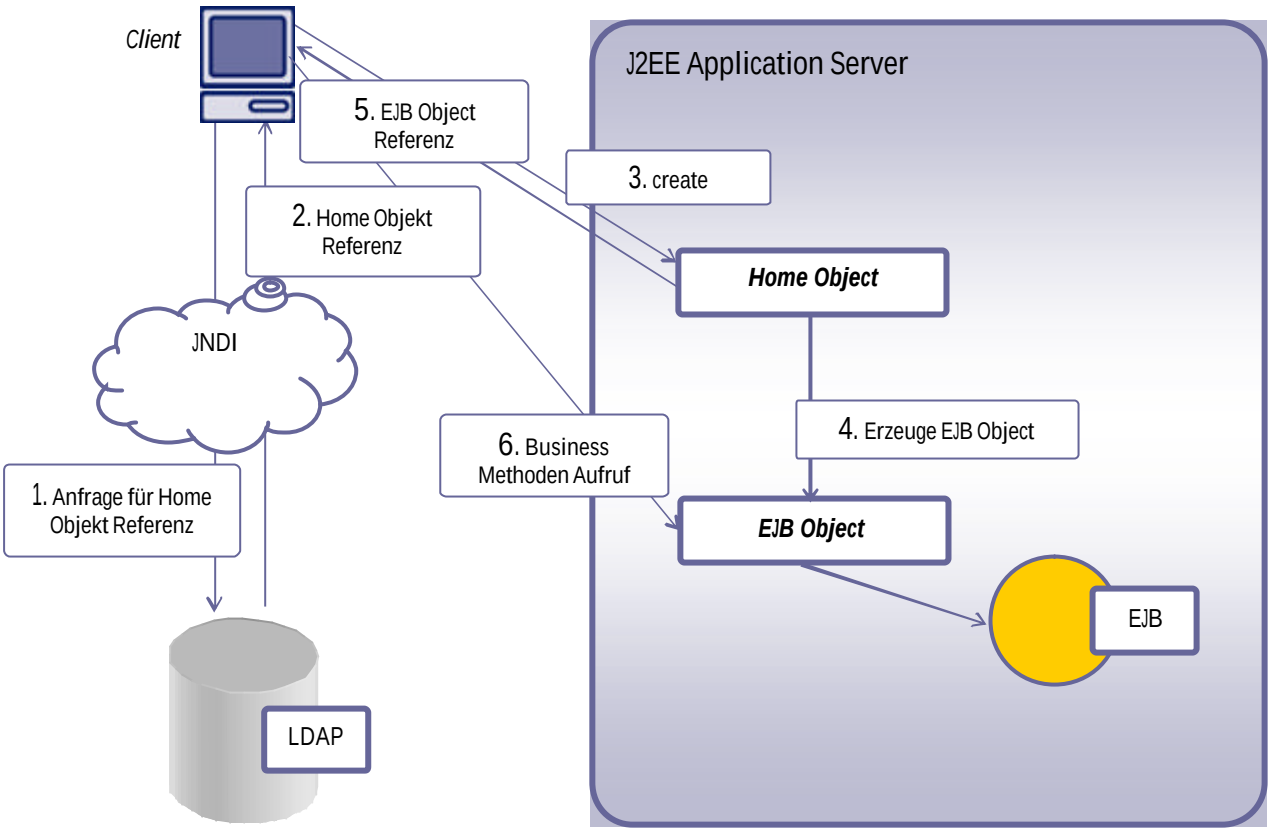
- Java RMI-IIOP basierende Clients: Diese Clients benutzen JNDI um Objekte zu lokalisieren und JTA um Transaktionen zu kontrollieren

- CORBA basierende Clients: Diese Clients können z.B. in C++ geschrieben sein und benutzen COS Naming (CORBA Naming Service) für Namensauflösung und OTS für Transaktionskontrolle

EJBs erzielen „Location Transparency" durch Verwendung von Naming und Directory Services Traditionell wurden Directory Services zur Speicherung von Benutzernamen, Kennwörtern, Rechneradressen verwendet. EJB Server verwenden Directories um den Speicherorte für EJB Home Objekte, EJB Environment Properties, Datenbanktreiber usw. zu speichern.

Die de facto API um mittels Java auf Naming Services zuzugreifen ist JNDI. Clients, die EJB Home Objekte lokalisieren wollen identifizieren diese über „Nicknames". Im Deployment Schritt einer EJB bindet der Container automatisch diesen „Nickname" an das Home Objekt der Bean. Jeder Client kann unabhängig seines physischen Standortes dieses Objekt über den Namen finden und referenzieren.

Abbildung: Erhalten einer Referenz auf ein Home Objekt.



3.2 Session Beans

Session Beans implementieren Business Logik, Workflows und Algorithmen. Ihre Lebensdauer entspricht in der Regel der Zeitspanne, die ein Client für die Ausführung der EJB benötigt. Die Länge der Client Session bestimmt daher generell die Lebensdauer einer Session Bean woher sich auch ihr Name ableitet. Im Gegensatz zu Entity- Beans sind Session Beans nichtpersistente Objekte.

3.2.1 Stateful Session Beans

Stateful Session Beans sind für Business Prozesse gedacht, die mehrere Methodenaufrufe umfassen und Transaktionen ausführen. Sie bewahren den auf den Client bezogenen Status der Aufrufe der Methoden.

3.2.2 Stateless Session Beans

Stateless Session Beans sind naturgemäß für Anfragen einer einzigen Request Konversation vorgesehen. Sie behalten keine Information über den Status von Methodenaufrufen. Ein gutes Beispiel für eine Stateless Session Bean ist die Validation von Kreditkarteninformationen. Die Eingabeparameter sind die Kartennummer, der Name und das Ablaufdatum, der Rückgabewert eine Ja oder Nein Antwort. Nach Rückgabe der Antwort ist die Bean sofort für einen neuen Client bereit.

Da Stateless Session Beans keinen Status und keine Informationen über ihre Methodenaufrufe speichern sind Instanzen von ihnen auch unterscheidungslos. Tatsächlich werden Instanzen von Stateless Session Beans „gepooled“ und wiederverwendet.

3.2.3 Lebenszyklus einer Stateful Session Bean

Pooling Mechanismen für Stateful Session Beans erfolgen nach ähnlichem Prinzip wie Context Switching von Applikationen auf Ebene des Betriebssystems. Um die Anzahl von Stateful Session Bean Instanzen im Hauptspeicher gering zu halten werden diese ggf. auf externe Speicher ausgelagert und bei Bedarf wieder in den Hauptspeicher geladen.

Eine reaktivierte Instanz ist in der Regel nicht die selbe, wie jene Instanz, die den Client bis zum Zeitpunkt des Auslagerns bediente. Einer reaktivierten Instanz wird bei Aktivierung einfach der Status, den sie zum Zeitpunkt der Passivierung hatte zugewiesen.

Der Vorgang des Auslagerns und der Reaktivierung von Bean Instanzen ist dadurch möglich, da das `javax.ejb.EnterpriseBean` Interface von `java.io.Serializable` ableitet.

Membervariablen von Beans sind Bestandteil eines Status wenn folgendes zutrifft:

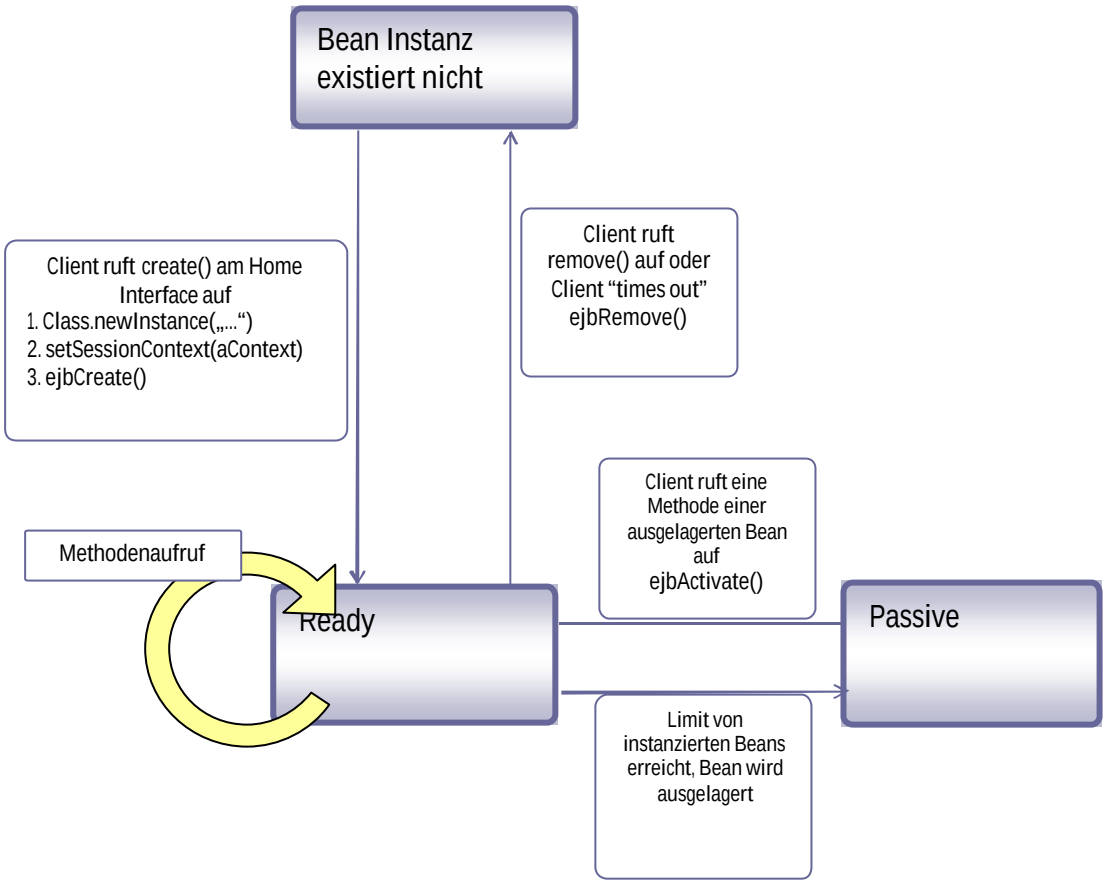
- Die Membervariable ist ein nontransienter primitiver Typ
- Die Membervariable ist ein nontransientes Java Objekt (leitet von java.lang.Object ab)

Container speichern auch Referenzen folgender Art bei Aktivierung/Passivierung:

- EJB Objekt Referenzen
- Home Objekt Referenzen
- EJB Session- Kontext Referenzen
- JNDI Naming Kontext Referenzen

Bevor eine Enterprise Bean ausgelagert wird, ruft der Container die Methode `ejbPassivate()`der Bean auf. In dieser Methode können Ressourcen wie Datenbankverbindungen, geöffnete Dateien oder Socket Verbindungen freigegeben werden, die für Objekt Serialisierung nicht geeignet sind.

Eine weitere Callback Methode einer Session Bean ist `setSessionContext()`. Über eine `SessionContext` Instanz lassen sich die Methoden `getEJBLocalObject()` und `getEJBObject()` aufrufen falls Referenzen auf die eigene Bean Instanz benötigt werden (analog dem `this` keyword jeder Java Klasse) .



Obige Abbildung zeigt den Lebenszyklus einer Stateful Session Bean

3.3 Entity Beans

Entity Beans sind persistente Daten Komponenten, die über Wissen verfügen, ihre Informationen in relationalen Datenbanken oder Legacy Systemen zu speichern. In ähnlicher Weise sind sie wie serialisierfähige Java Objekte. Der Unterschied ist jedoch, dass sie über eine Abbildung von „Objekt zu Relational“ in relationalen Datenbanken persistent gemacht werden. Entity Beans sind quasi die „im Hauptspeicher Sicht“ in eine Datenbank. Eigenschaften von Entity Beans sind:

- Die Entity Bean Klasse bildet auf eine Entity Definition eines Datenbankschemas ab. Eine Instanz entspricht daher einer Zeile einer Tabelle. Sie bietet auch Methoden zur Manipulation der Daten an

- Die Primary Key Klasse identifiziert eindeutig eine Instanz der Bean

- Entity Beans sind langlebige Objekte. Sie überleben kritische Fehler wie Abstürze des Application Servers

- Entity Bean Instanzen sind eine „View“ in eine Datenbank. Die Instanzen werden durch die eigenen Methoden der Bean in der Java VM manipuliert und mittels der CallBack Methoden `ejbStore()` und `ejbLoad()` mit dem darunterliegenden

Datenspeicher synchronisiert. Die Synchronisationszeitpunkte – d.h. die Aufrufe von `ejbLoad()` und `ejbStore()` werden durch den Container bestimmt.

- Mehrere Entity Bean Instanzen repräsentieren die selben darunterliegenden Daten. Alle Enterprise Beans Instanzen sind single threaded und ein serialisierter Zugriff auf eine einzige Instanz durch konkurrierenden Clients wäre aus Performancegründen undenkbar. Container erzeugen daher mehrfache Instanzen einer Entity Bean. Ein Transaktionsmanagement bewahrt hierbei die Integrität und Konsistenz der Daten.

- Entity Bean Instanzen werden „gepooled". Wie auch bei Session Beans existieren auch hier zwei CallBack Methoden `ejbActivate()` und `ejbPassivate()`, die vom Container aufgerufen werden, um der Instanz die Möglichkeit zu geben, Ressourcen anzufordern bzw. freizugeben. Beim Vorgang des Passivierens speichert zusätzlich die Instanz ihren Status in den darunterliegenden Speicher. Der Container ruft daher zuvor die Methode `ejbStore()` auf. Analog dazu ruft der Container `ejbLoad()` nach Aktivierung einer Entity Bean auf.

- Es gibt 2 Arten von Entity Beans:

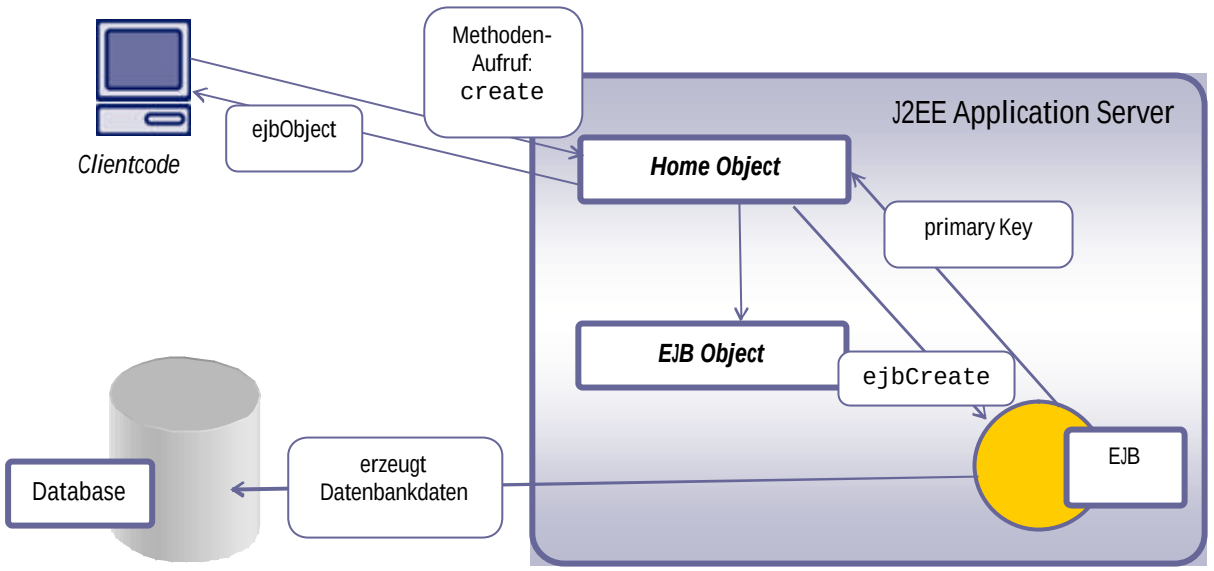
    - Bean Managed persistent Entity Beans: Diese Art implementiert eigenständig den Code um Instanzen im den zugrundeliegenden Speicher zu persistieren

    - Container managed persistent Entity Beans: Der Container implementiert den Code zur Persistenz

3.3.1 Erzeugen und Zerstören von Entity Beans

Entity Beans sind eine „View" in eine Datenbank. Das Erzeugen einer neuen Instanz bedeutet daher das Anlegen von Daten in einer Datenbank. Bei Initialisierung der Entity Bean durch `ejbCreate()` werden in der Datenbank die korrelierenden Daten angelegt, bei Ausführen von `ejbRemove()` die zugehörige Daten gelöscht. `ejbCreate()` liefert als Rückgabewert eine Primary Key Instanz, die eine Unterscheidung der Instanz für den Container möglich macht, während `create()` des Home Objektes ein `EJBObject` zurückliefert. Beide Methoden müssen durch ihre Parameter übereinstimmen.

Abbildung: Erzeugen einer **Bean managed Entity Bean**, Zusammenhang von `create` und



`ejbCreate`

Die Methode `ejbRemove()` wird bei Entity Beans nicht aufgerufen, wenn ein Client einen Timeout überschreitet. `ejbRemove()` löscht die korrelierenden Daten in der Datenbank, zerstört aber nicht notwendigerweise die Instanz der Bean.

Entity Bean Instanzen können auch durch einen Suchvorgang erzeugt werden. Das Finden einer Instanz ist analog einem SQL Select Statement. Das Home Objekt einer Entity Bean bietet dazu „Finder" Methoden an.

3.3.2 Entity Kontexte

Alle Enterprise Beans besitzen ein Kontext Objekt, das die Umgebung des Objektes identifiziert. Dieses Kontext Objekt wird durch den Container beschrieben und beinhaltet Informationen bezüglich Security oder Transaktionen. Das Interface dazu ist: `javax.ejb.EntityContext`, das von `javax.ejb.EJBContext` ableitet.

Methoden dieses Interfaces sind:

getEJBObject() und getEJBLocalObject(): Analog dem this Argument liefern diese Methoden eine Referenz auf sich selbst.

getPrimaryKey(): Liefert den primären Schlüssel der im Moment gerade assoziierten Instanz der Bean. Entity Bean Instanzen werden gepooled, je nach Bedarf ausgelagert und wieder in den Speicher geladen. Dies bedeutet, dass eine Bean Instanz im Zuge von einer Aktivierung zu einer anderen Dateninstanz mit einem anderen primären Schlüssel wechseln kann. `getPrimaryKey()` ist die geeignete Methode herauszufinden, welche Dateninstanz nach einem `ejbLoad()` bzw. vor einen `ejbRemove()` vorliegt.

3.3.3 Bean- Managed Entity Beans

Implementierungs- Richtlinien:

`setEntityContext()`

Falls ein Container den Pool seiner Instanzen vergrößert assoziiert er jede neue Instanz mit einer Umgebung, die von der Bean abgefragt werden kann. Er ruft diese Methode auf um der Bean die Möglichkeit zu geben, diesen Kontext in einer Membervariablen zu speichern. Weiters können alle daten-unspezifischen Ressourcen hier angefordert werden.

`ejbFind*()`

Zumindest eine „Finder" Methode muss implementiert sein, die `ejbFindByPrimaryKey` heißt. „Finder" Methoden lokalisieren eine oder mehrere Daten Instanzen im darunterliegenden Speicher. Rückgabewerte sind Primary Key Objekt Instanzen, die Entity Beans an EJB Objekte binden.

Bean „Finder" Methodennamen müssen mit „ejbFind" beginnen und entweder einen Primary Key oder eine `Collection` zurückliefern. Alle „Finder" Methoden müssen auch im Home und Local Home Interface angeführt werden. Jedoch liefern die Methoden des Home Interfaces anstatt des Primary Key's bzw. einer `Collection` von Primary Keys, EJB Objekte bzw. eine Collection von EJB Objekten zurück. Ihre Methodennamen beginnen ohne „ejb" jedoch mit „Find"

Wenn ein Client einen „Finder" am Home Objekt aufruft, delegiert der Container den Aufruf an die Bean. Die Bean liefert eine Collection von Primary Keys an den Container zurück unter dessen Zuhilfenahme der Container eine Collection von EJB Objekten erstellt und dem Client übergibt.

`ejbHome*()`

globale Methoden einer Entity Bean, die unabhängig zu einer konkreten Assoziation zu einer Daten Instanz sind. z.B. Totale Zeilenanzahl einer Tabelle.

`ejbCreate()`

Wenn ein Client `create()` im Home Objekt aufruft, ruft der Container die korrespondierende `ejbCreate()` Methode der gepooled-ten Bean Instanz auf. Die entsprechenden Daten werden in der Datenbank mit z.B. einem SQL Insert Statement angelegt und ein Primary Key Objekt wird an den Container zurückgegeben.

`ejbPostCreate()`

Zu jeder `ejbCreate()` Methode existiert eine `ejbPostCreate()` Methode. Diese Methode wird nach `ejbCreate()` vom Container aufgerufen. Die Initialisierung der Instanz kann hier vervollständigt werden, z.B. das Zurücksetzen von Transaktionsparametern.

`ejbActivate()`

Der Container ruft diese Methode auf, wenn ein Client eine Anfrage an eine ausgelagerte Bean Instanz absetzt. In dieser Methode können ggf. notwendige Ressourcen wie Socket Verbindungen initiiert werden.

`ejbLoad()`

Der Container ruft diese Methode auf, um Daten der Datenbank in eine Bean Instanz zu laden. Die `getPrimaryKey()` Methode des Entity Kontextes liefert den primären Schlüssel um die richtigen Daten aus der Datenbank zu selektieren.

`ejbStore()`

Der Container ruft diese Methode auf, um die Daten in der Datenbank zu aktualisieren. Diese Methode wird auch direkt vor `ejbPassivate()` aufgerufen. In der Regel finden in dieser Methode SQL Update Statements Verwendung.

`ejbPassivate()`

Diese Methode wird aufgerufen, wenn der Container eine Bean Instanz zum Pool zurückgeben möchte. Notwendige Ressourcen wie Socket Verbindungen können hier freigegeben werden.

`ejbRemove()`

Der Container ruft diese Methode auf um korrespondierende Daten in der Datenbank zu löschen. Die `getPrimaryKey()` Methode des Entity Kontextes liefert hierbei den primären Schlüssel der zu löschenden Daten.

`unsetEntityContext()`

Diese Methode disassoziiert die Umgebung von einer Entity Instanz und wird direkt vor `ejbRemove()` aufgerufen.

3.3.4 Die Deployment Deskriptor Datei: „ejb-jar.xml“

Beispiel einer ejb-jar.xml Datei für eine Bean Managed Entity Bean, die via JDBC in eine Datenbank persistiert:

```
<?xml version="1.0"?>

<!DOCTYPE ejb-jar PUBLIC
"-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
"http://java.sun.com/dtd/ejb-jar_2_0.dtd">

<ejb-jar>

 <enterprise-beans>
   <entity>
      <description>
            Diese Bean repräsentiert ein Bank Konto.
      </description>

      <ejb-name>Account</ejb-name> Eine Bezeichnung für die Bean

      <home>examples.AccountHome</home> Der voll qualifizierte Name des Home Interfaces
      <remote>examples.AccountRemote</remote> Der voll qualifizierte Name des Remote Interfaces
     <local-home>examples.AccountLocalHome</local-home> Der voll qual. Name des Local Home
Interfaces
```

```
      <local>examples.AccountLocal</local>
      <ejb-class>examples.AccountBean</ejb-class>
      <persistence-type>Bean</persistence-type>
      <prim-key-class>examples.AccountPK</prim-key-class>
```

Der voll qualifizierte Name des Local Interfaces

Der voll qualifizierte Name der Bean Klasse

Art der Persistenzmethode

Der voll qual. Name des Prim. Schlüssels

```
      <reentrant>False</reentrant>
```

Erlauben/Nichterlauben von 2 Threads innerhalb der Bean.
Bsp.: Bean A ruft Bean B auf, die wieder Bean A aufruft

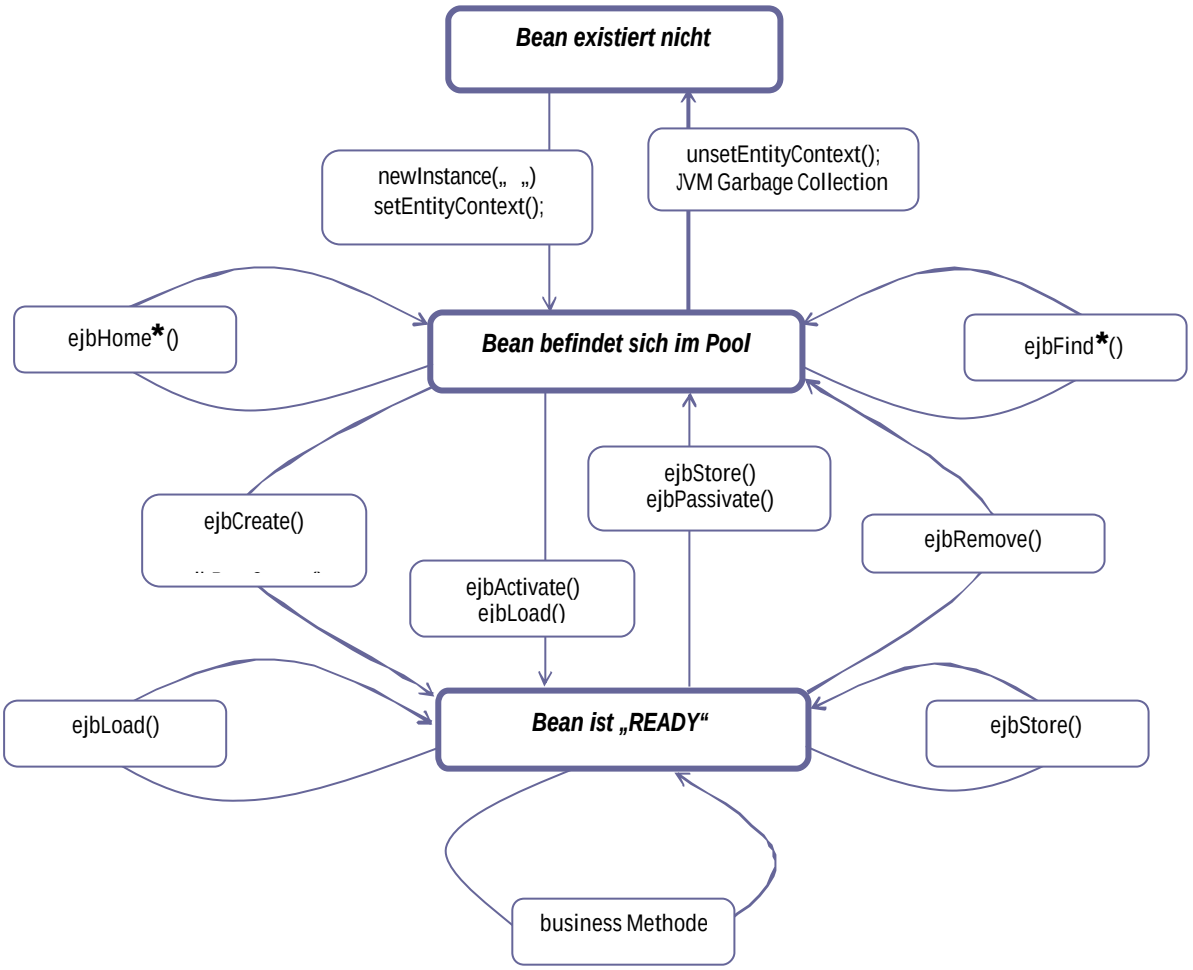
```
      <resource-ref>
           <description>DataSource for the Account database</description>
           <res-ref-name>jdbc/AccountDB</res-ref-name>
           <res-type>javax.sql.DataSource</res-type>
           <res-auth>Container</res-auth>
      </resource-ref>
  </entity>
 </enterprise-beans>

</ejb-jar>
```

Richtet den JDBC Treiber ein und macht ihn über JNDI verfügbar

3.3.5 Der Lebenszyklus einer Bean Managed Entity Bean.

Jeder Methoden Aufruf erfolgt vom Container an die Bean Instanz:



Um eine Instanz einer Bean zu erzeugen ruft der Container `newInstance()` auf. Dies ruft den Konstruktor der Bean auf. Der Bean wird ein Kontext zugewiesen und der Container ruft `setEntityContext()` auf. Diese Schritte erfolgen, wenn der Container den Pool der Instanzen vergrößert.

Die Bean befindet sich nun im Pool, hat jedoch noch keine Daten geladen bzw. Bean-spezifische Ressourcen gebunden. Instanzen im Pool können verwendet werden um Finder bzw. Home Methoden auszuführen. Ein Container, der eine Instanz aus dem Pool entfernen möchte, ruft die Methode `unsetEntityContext()` auf.

Ein Client der neue Daten erzeugen will, ruft die Methode `create()` mit den entsprechenden Daten als Parametern am Home Objekt auf. Der Container nimmt eine Instanz aus dem Pool und ruft die den Argumenten entsprechende Methode `ejbCreate()` an der Instanz auf. Die Daten sind nun in der Datenbank gespeichert und die Bean im „Ready“ Zustand.

Im „Ready“ Zustand erfolgen Business Methoden Aufrufe der Bean und `ejbStore()` bzw. `ejbLoad()` Aufrufe. Der Container ruft basierend auf den Transaktionseinstellungen `ejbStore()` bzw. `ejbLoad()` auf um die Daten mit der Datenbank zu synchronisieren.

Eine Bean kann den „Ready Zustand“ auf 2 unterschiedliche Möglichkeiten verlassen:

- Ein Client ruft `remove()` am Home Objekt auf, wodurch die Daten in der Datenbank durch die Bean Methode `ejbRemove()` gelöscht werden.

- Ein Client überschreitet einen Timeout oder der Container benötigt die Instanz einer Bean aufgrund von Ressourcen Engpässen. Bei diesem Vorgang der Passivierung `ejbPassivate()`wird zuvor noch die Methode `ejbStore()` aufgerufen um die aktuellen Daten in die Datenbank zu propagieren.

Einem Client wird ein EJB Objekt durch den Vorgang der Aktivierung `ejbActivate()` wieder zugeordnet. Der Container ruft `ejbLoad()` auf um die Bean mit den Daten aus der Datenbank zu assoziieren.

3.3.6 Container Managed Entity Beans

Bean Managed Entity Beans sind nützlich, wenn Kontrolle auf die zugrundeliegenden Operationen der Datenbank notwendig ist. Der größere Vorteil von Enterprise Entity Beans entsteht jedoch mit der Verwendung von Container Managed Entity Beans. Container Managed Entity Beans sind frei von jeglichem JDBC- oder ähnlichem Datenbankzugriffscode. Die EJB Spezifiktion sieht in Bezug auf CMP (Container Managed Persistence) eine strikte Trennung zwischen einer Bean und ihrer Logik zur Persistenz vor. Diese Trennung ist insofern nützlich, da die Repräsentation der Persistenz ohne Änderung der Businesslogik erfolgen kann, sprich eine Änderung der Datenbank ohne Quellcode der Bean möglich ist. Der Container generiert den Datenbankcode indem er die Entity Bean Klasse ableitet.

Eigenschaften von Container Managed Entity Beans:

- Jeder Container implementiert die Persistenz- Repräsentation auf unterschiedliche Weise. Aus diesem Grund werden in der Entity Bean Klasse keine zu persistierende Felder (Membervariablen) definiert. Da Business Methoden in der Bean- Klasse jedoch Zugriff auf Membervariablen benötigen, wird dieser Zugriff über abstrakte Getter/Setter Methoden gelöst. Folglich wird die Implementierung einer CMP Entity Klasse auch als abstrakt deklariert.

- CMP Entity Beans haben ein abstraktes Persistenz Schema: Der EJB Container entscheidet aufgrund des Deployment Descriptors welche Felder und Getter/Setter Methoden implementiert werden. Bsp:

  - ```
    <cmp-version>2.x</cmp-version>
       <abstract-schema-name>Account    Bean</abstract-schema-
    name>
          <cmp-field>
              <field-name>accountID</field-name>
          </cmp-field>
          <cmp-field>
              <field-name>accountName</field-name>
          </cmp-field>
          <primary-field>accountID</primary-field>
    ```

- Finder Methoden werden vom Container mittels einer EJB Query Language generiert. EJB-QL ist eine SQL ähnliche Abfragesprache, die eine SELECT Klause, eine FROM und eine WHERE Klause enthält. EJB-QL Statements werden im Deployment Descriptor angeführt.

  - Bsp:
    ```
    SELECT OBJECT(a) FROM Account AS a WHERE a.balance > ?1
    ```

    ?1 entspricht dem 1. Parameter, welcher der Methode `find*` übergeben wird

- Nicht alle Felder einer CMP Entity Bean müssen durch den containergenerierten Code persistiert werden. Es besteht die Möglichkeit Felder zu deklarieren, die auch auf Daten aus anderen Datenquellen zuzugreifen. Der Container notifiziert hierbei die Bean vor Aufruf der Persistenz- Methoden

- CMP Entity Beans können `ejbSelect*()` Methoden haben. ejbSelect*() Methoden sind interne Abfrage Methoden, die für Clients nicht sichtbar sind. Sie sind Hilfsmethoden für den Datenzugriff und werden analog zu Finder Methoden mittels EJB-QL spezifiziert. Select Methoden können auf Daten von anderen Entity Beans zugreifen, Return Werte können Entity Beans aber auch andere Objekt-Typen sein.

3.3.7 Implementierungs- Richtlinien für Container Managed Entity Beans

`setEntityContext()`

analog BMP

`ejbFind*()`

Finder Methoden enthalten keinen vom Entwickler geschriebenen datenbankspezifischem Code. Der Container generiert den Code mittels der im Deployment Descriptor angeführten EJB-QL Statements.
    ```

`ejbSelect*()`

Select Methoden sind interne Hilfsmethoden, die Abfragen ausführen. Sie werden `abstract` definiert und vom Container unter Berücksichtigung von EJB-QL Statements implementiert.

`ejbHome*()`

globale Methoden einer Entity Bean, die unabhängig zu einer konkreten Assoziation zu einer Daten Instanz sind. z.B. „Berechne die totale Zeilenanzahl einer Tabelle“. Sie können entweder durch JDBC Code oder mittels Select Methoden implementiert sein.

`ejbCreate()`

Diese Methode enthält keinen Code zur Erstellung von Daten in der Datenbank. Es sollten hier Validierung der Parameter und Aufrufe der abstrakten Setter Methoden stattfinden.

`ejbPostCreate()`

analog BMP

`ejbActivate()`

analog BMP

`ejbLoad()`

Der Container lädt die Daten automatisch aus der Datenbank bevor diese Methode aufgerufen wird. In dieser Methode können Funktionen wie Datendekompression der aus der Datenbank gelesenen Daten implementiert sein.

`ejbStore()`

Diese Methode wird vom Container aufgerufen, bevor die Daten persistiert werden. Hier können die Container managed Felder zur Speicherung vorbereitet werden – z.B. Datenkompression.

`ejbPassivate()`

analog BMP

`ejbRemove()`

Der Container löscht die mit dem Objekt assoziierten Daten

`unsetEntityContext()`

analog BMP

3.4 Message-Driven Beans

Verteilte Methoden Aufrufe von Entity und Session Beans mittels RMI-IIOP sind in vielen Szenarien nützlich, jedoch sprechen auch einige Punkte gegen RMI-IIOP.

- Performance: Ein RMI-IIOP Client blockiert solange bis der Server die angeforderte Operation ausgeführt hat. Erst wenn der Server diesen Prozess beendet hat, kann der Client in seinem Prozess fortfahren

- Verlässlichkeit: Wenn ein Client Funktionen eines RMI-IIOP Server aufruft muss dieser laufen und das Netzwerk intakt sein. Im Falle eines Serverabsturzes kann der Client nicht mit seiner Arbeit fortfahren

„Messaging" ist eine alternative Methode für verteilten Methodenaufruf. Ein messaging basierender Client blockiert nicht bei Absetzen eines Requests. Falls die zugrunde liegende messaging-orientierte Middleware auch garantierte Zustellung unterstützt, können Anfragen nach dem „Fire and Forget" Paradigma betrachtet werden. Die Zustellung des Requests ist auch bei verübergehender Nichtverfügbarkeit eines Servers garantiert. Weiters können Requests auch in einem Broadcast verfahren an mehrere Server versendet werden.

Message-Driven Beans sind EJB Komponenten, die JMS Nachrichten empfangen können. Sie konsumieren Messages von Queues oder Topics, die von JMS Clients gesendet wurden. Folgende Eigenschaften treffen auf sie zu:

- Man ruft Message- Driven Beans nicht über ein objektorientiertes Remote-Interface auf. Deshalb besitzen Message-Driven Beans kein Home-, kein Local Home-, kein Remote- und kein Local- Interface

- Eine Message-Driven Bean besitzt nur eine Business Methode `onMessage()`. Sie empfängt Nachrichten von JMS Zielen, die kein Wissen über den Inhalt von Nachrichten haben. Die Methode `onMessage()` akzeptiert Nachrichten folgender Typen: `BytesMessage, ObjectMessage, ObjectMessage, StreamMessage, MapMessage`.
  Der `instanceOf` Operator ermittelt den Typ der Nachricht zur Laufzeit. Das Verhalten der Bean ist meist durch ein Bündel von `if-Statements` definiert, die auf den Inhalt der Nachricht bezogene Aktionen ausführen.

- Message- Driven Beans liefern keine Return Werte sondern verwenden Design Patterns um Antworten an den Sender zu schicken.

- Message- Driven Beans können keine Exceptions an den Client senden. Ein Nachrichtensender wartet nicht auf die Entgegennahme der Nachricht der Bean. Aus diesem Grund kann der Client auch keine Exceptions entgegennehmen. Die Spezifikation verbietet das Werfen von Application- Exceptions erlaubt jedoch das Generieren von System- Exceptions, die durch den Container behandelt werden

- Message- Driven Beans sind stateless. Alle Instanzen sind anonym und besitzen keine Identität, die für einen Client sichtbar wäre. Daher können mehrere Bean-Instanzen gleichzeitig Nachrichten einer JMS Destination verarbeiten.

- Message- Driven Beans können dauerhafte oder nicht dauerhafte Subscriber sein. Dauerhafte Subscriber erhalten alle Nachrichten auch im Falle von Inaktivität des Subscribers. Nachrichten werden im Falle der Inaktivität persistiert und nach Aktivierung zugestellt. Ein nicht dauerhafter Subscriber erhält nur Nachrichten, die publiziert werden während der Subscriber aktiv ist.

Message- Driven Beans sind Klassen, die zwei Interfaces implementieren:

`javax.jms.MessageListener` und `javax.ejb.MessageDrivenBean`.

Die zu implementierenden Methoden sind:

`onMessage(Message)`
Diese Methode wird bei jeder Konsumation einer Nachricht vom Container aufgerufen. Eine Instanz einer Bean verarbeitet immer nur eine Nachricht. Es ist die Aufgabe des Containers gleichzeitige Konsumation von Nachrichten zu ermöglichen und auf gepoolte Instanzen aufzuteilen.

`ejbCreate()`

Diese Methode wird aufgerufen, wenn die Bean erzeugt und zum Pool anderer Beans hinzugefügt wird. In dieser Methode können Variablen initialisiert, Referenzen zu Ressourcen, wie andere EJB oder Datenbankverbindungen erzeugt werden.

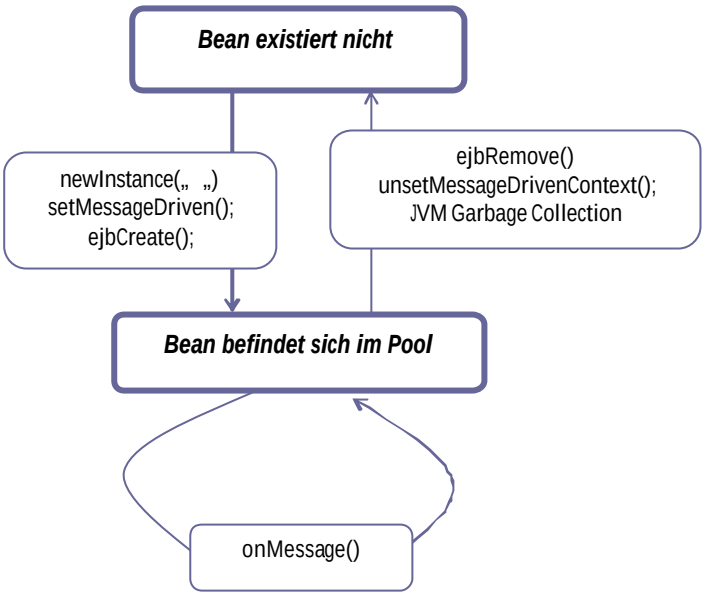
`ejbRemove()`

Diese Methode wird aufgerufen, wenn eine Bean aus dem Pool entfernt wird. Hier sollten bestehende Ressourcen freigegeben werden.

`setMessageDrivenContext()`

Diese Methode wird von ejbCreate aufgerufen und bietet die Möglichkeit, das Environment der Bean abzuspeichern. Die Methoden des MessageDrivenContext Objektes beschränken sich jedoch auf transaktions-bezogene Methoden.

3.4.1 Der Lebenszyklus einer Message - Driven Bean:



Ein Deployment Descriptor einer Message Driven Bean:

```
<ejb-jar>
 <enterprise-beans>
   <message-driven>
      <ejb-name>LogBean</ejb-name>
      <ejb-class>examples.LogBean</ejb-class>
      <transaction-type>Container</transaction-type>
      <acknowledge-mode>auto-acknowledge</acknowledge-mode>
      <message-driven-destination>
         <destination-type>javax.jms.Topic</destination-type>
      </message-driven-destination>
    </message-driven>
  </enterprise-beans>
</ejb-jar>
```

Optionale Subelemente des <message-driven> Tags:

```
<message-selector>
```

Ein Container wendet einen Message Selector auf Nachrichten einer JMS Destination an, um Nachrichten nach bestimmten Kriterien zu filtern. Ein Client kann mit der Methode message.setStringProperty(„key“, „value“) Eigenschaften der Nachricht bestimmen.

Beispiel:

```
<message-selector>JMSType = ‚log’ AND logLevel = ‚severe’</message-selector>
```

```
<acknowledge-mode>
```

Die Konsumation von Nachrichten kann dem Container bestätigt werden. Falls der Container Transaktionen kontrolliert ist eine Bestätigung der Konsumation nicht notwendig. Der Container fügt beim Rollback der Transaktion die Nachricht automatisch zurück in die Queue. Im Falle von Bean managed Transaktionen findet die Konsumation außerhalb einer Transaktion statt. Aus diesem Grund muss dem Container die Konsumation mitgeteilt werden. „Auto-acknowledge" bedeutet eine Bestätigung nach Ausführung der onMessage Methode, „Dups-ok-acknowledge" eine zeitlich lockerere, containerbezogen performantere Bestätigungsart mit dem Nachteil einer Mehrfachzustellung von identischen Nachrichten.

```
<message-driven-destination>
<destination-type>
<subscription-durability>
```

Der Destination Typ kann entweder `javax.jms.Topic` oder `javax.jms.Queue` sein. Mit „durable" bzw. „nondurable" kann ein Subscriber als dauerhaft bzw. nichtdauerhaft spezifiziert werden.

3.4.2 Weitere Eigenschaften Message Driven Beans

Transaktionen und Message-Driven Beans

Message-Driven Bean können in keinem Fall Bestandteil einer Transaktion sein, der ein Produzent der Nachricht angehört. Typischerweise existieren 2 Transaktionen, eine für den Produzent, der die Nachricht in die Queue stellt und eine zweite für den Konsument.

Sicherheit

Message-Driven Beans erhalten keine sichere Information über die Identität des Produzenten der Nachricht. Es ist nicht vorgesehen, sicherheitsbezogene Informationen an eine JMS Nachricht anzuhängen.

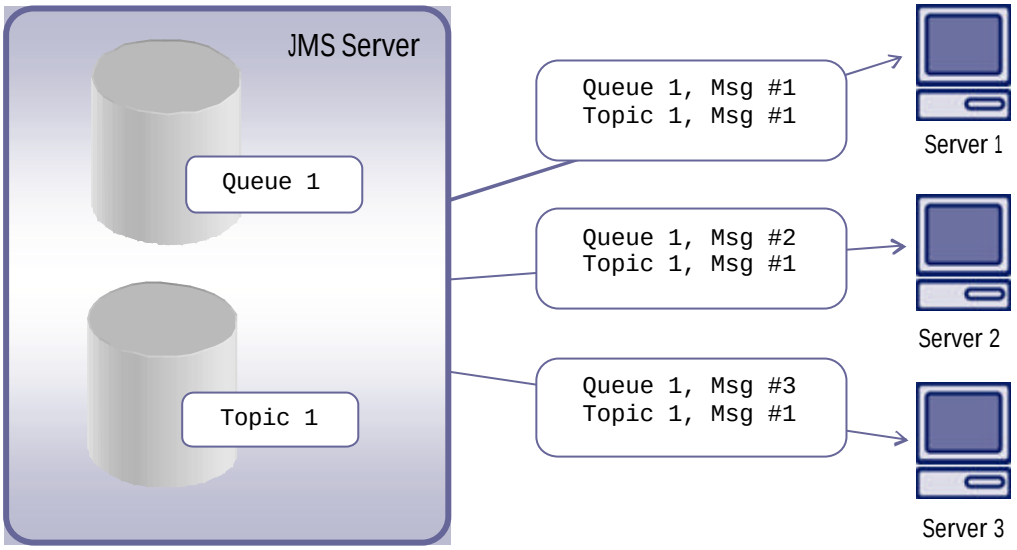
Load Balancing

Push Model: Load Balancing Algorithmen versuchen auf Basis von gesammelten Daten in der Vergangenheit Rückschlüsse auf zu erwartende Auslastung vorherzusagen. Der Algorithmus berechnet im Falle von Entity und Session Beans den wenigst ausgelasteten Server und schiebt ihm den Request zu. (Push)

Pull Model: Im Falle von message-driven Beans werden Anfragen einfach in eine Queue gestellt. Mehrere Container konkurrieren in der Abarbeitung der Nachrichten aus der Queue.

Im Falle von Load Balancing und Publish/Subscribe erhält nur eine Instanz eines Pool von Message- driven Beans die Message, jedoch erhält jeder Container des Clusters alle Messages eines Topics. Bei Bedarf einer exakt einmaligen Konsumation ist daher ein Design mittels einer Queue zu verwenden.

Jeder Container registriert sich als Konsument am JMS Server, der die Zustellung der



Nachrichten zu den Konsumenten ausbalanciert.

Ordnung von Nachrichten

Ein JMS Server garantiert nicht die Auslieferung von Nachrichten in einer vorhersehbaren Reihenfolge. Message-Driven Beans sollten darauf vorbereitet sein, Nachrichten unabhängig von der Reihenfolge ihres Eintreffens verarbeiten zu können.

Ausbleibende `ejbRemove()` Aufrufe

Analog zu Session und Entity Beans ist nicht garantiert, dass die Methode `ejbRemove()` ausgeführt wird falls die Bean Instanz zerstört wird (System Crash). Auch im Falle von geworfenen System- Exceptions wird `ejbRemove()` nicht aufgerufen, der Entwickler hat daher vor Werfen der Exceptions relevante Aufräumarbeiten zu codieren. `ejbRemove()`wird nur dann vom Container aufgerufen, wenn eine Instanz nicht mehr benötigt wird.

„Poison Messages“

Im Falle von Container-managed Transaktionen kann es vorkommen, das von der Bean „unacknowledged“ Nachrichten erneut durch den JMS Server versendet werden. Dies kann im Falle von zurückgerollten Transaktionen geschehen, in dem die Bean eine Exception wirft oder `MessageDrivenContext().setRollbackOnly()` aufruft. Ein Rollback verhindert die Bestätigung der Erhaltes der Nachricht, was das erneute Zustellen der Message an den Container hervorruft.

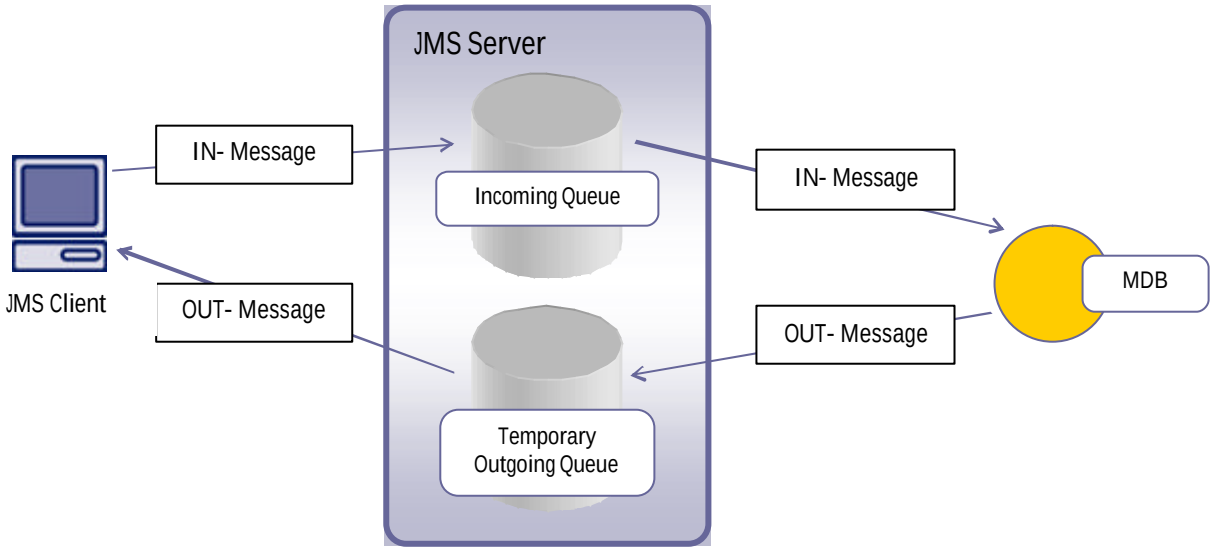
Strategien zur Verhinderung von Poison Messages:

- Vermeiden von Werfen von System Exceptions bei Business-Logik bezogenen Fehlern. Die EJB Spezifikation verbietet auch das Werfen von Application-Exceptions in der `onMessage()` Methode. Eine Lösung ist hier das Protokollieren des Fehlers in einer Log Datei

- Bean Managed Transactions sind Container Managed Transactions vorzuziehen. Nachrichten Konsumation und Bestätigung ist nicht Teil der Transaktion im Falle von Bean Managed Transaction. Eine Bean Managed Transaction kann zurückgerollt werden und der Erhalt der Message bestätigt werden.

- Application Server bieten Möglichkeiten Poison Messages zu kontrollieren. Dies umfasst das Kennzeichnen von Messages bei erneuten Zustellversuchen, eine max. Anzahl von Zustellversuchen, ein max. Zeitfenster für Zustellversuche

3.4.3 Ein einfaches Request/Response Paradigma



Der Client generiert zu einer Nachricht eine temporäre Destination, die mit seiner JMS Connection assoziiert ist. Der JMS Server erzeugt für die Lebensdauer der Verbindung dazu ein temporäres Topic oder eine Queue. Die Nachricht enthält zusätzlich ein `JMSReplyTo` Feld, das den Namen der temporären Queue spezifiziert und ein `JMSCorrelationID` Feld, das die Request Message identifiziert. Der Client erzeugt eine Session und einen MessageListener um Nachrichten der temporären Queue konsumieren zu können. Die Message Driven Bean verarbeitet die Nachricht des Clients, erzeugt unter Zuhilfenahme der `JMSReplyTo` und `JMSCorrelationID` Felder die Antwort.

Vorsicht ist bei Stateful Session Beans als JMS Clients gegeben. Im Zuge der Passivierung der Bean werden sämtliche Ressourcen wie die JMS Connection frei gegeben. Die mit der Connection verbundene Destination und sämtliche Antworten der temporären Queue sind verloren.

Vorteile/Nachteile einer temporären Queue

- kein zusätzlicher Administrationsaufwand für Konfiguration von Response Topics
- Da temporäre Queues an Connections gebunden sind können sich bösartige Client-Applikationen nicht daran binden und Nachrichten verfälschen oder abfangen
- Temporäre Queues können an keine persistenten Speicher gebunden werden und daher keine garantierte Nachrichtenauslieferung unterstützen

Vorteile/Nachteile einer permanenter Queue

- Permanente Queues garantieren persistente sichere Auslieferung von Nachrichten. Sie eignen sich sehr gut als Datenauslieferungsquellen für Daten, die fragmentiert an Clients (Mobile Geräte) abgegeben werden
- Filtermöglichkeiten: Ein Client kann einen Message Selector registrieren um, nur spezifische Nachrichten erhalten. Bsp: ClientID=1234
- JMS sieht keine Sicherheitsrestriktionen bezüglich des Bindens und Verwenden von Message Selectoren vor. Aus diesem Grund kann jeder beliebige Client Nachrichten konsumieren, die ursprünglich nicht für ihn gedacht waren.

3.5 Zusätzliche Funktionalität für Beans

3.5.1 Beanaufrufe von Beans

Analog zu Beanaufrufen außerhalb des Applicationservers muss zuerst via JNDI eine Referenz auf die Bean ermittelt werden. Bei Lookups auf Beans von Beans müssen jedoch keine JNDI Initialisierungsparameter angegeben werden. Folgendes Statements sind ausreichend:

```
Context ctx = new InitialContext();
Object result = ctx.lookup("java:comp/env/ejb/CatalogHome");
CatalogHome                    =                    (CatalogHome)
javax.rmi.PortableRemoteObject.narrow(result,
      CatalogHome.class);
Catalog c = home.create(...);
```

Die JNDI Location java:comp/env/ejb/CatalogHome ist in diesem Beispiel ein Ort, den die EJB Spezifikation empfiehlt, jedoch nicht verpflichtend vorschreibt. Um Flexibilität zu gewährleisten werden Beans in der Regel über EJB Referenzen gefunden. Eine EJB Reference ist hierbei ein Nickname der vom Deployer an die tatsächliche JNDI Location gebunden und im Deployment Descriptor definiert ist.

```
<ejb-ref>
  <ejb-ref-name>ejb/ExampleHome</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <home>examples.ExampleHome</home>
  <remote>examples.ExampleRemote</remote>
</ejb-ref>
```

bzw. für Local Interfaces:

```
<ejb-local-ref>
  <ejb-ref-name>ejb/ExampleHome</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <local-home>examples.ExampleHome</local-home>
  <local>examples.ExampleLocal</local>
</ejb-local-ref>
```

3.5.2 Resource Factories

Eine Resource Factory ist ein Provider für Ressourcen. Beispiele sind JDBC Treiber, JMS Treiber, oder JCA Resource Adapter. Analog Beans sucht man Ressourcen via JNDI.

```
Context jndiCntx = new InitialContext();
DataSource                          ds                                  =
(DataSource)jndiCntx.lookup("java:comp/env/jdbc/AccountDB");
return ds.getConnection();
```

java:comp/env/jdbc/AccountDB ist der EJB- vorgeschlagene Ort für JDBC Ressourcen.

Der korrespondierende Deployment Deskriptor sieht wie folgt aus:

```
<resource-ref>
           <description>DataSource       for       the       Account
database</description>
           <res-ref-name>jdbc/AccountDB</res-ref-name>
           <res-type>javax.sql.DataSource</res-type>
           <res-auth>Container</res-auth>
</resource-ref>
```

Das <res-auth> Tag kann auf „Container“ gesetzt werden, wenn Sign-On Informationen im Descriptor File angegeben, oder auf „Application“, wenn Sign-On Informationen wie Benutzername und Kennwort im Code angeführt sind.

3.5.3 Umgebungs- Variablen für Beans

Beans können durch Eigenschaften, die im Deployment Descriptor angeführt sind zur Laufzeit konfiguriert werden.

```
Context jndiCntx = new InitialContext();
String                          taxAlgorithmus                          =
(String)jndiCntx.lookup("java:comp/env/Price/Algorithmus");
```

Umgebungsvariablen müssen unterhalb von „java:comp/env“ abgelegt sein.

Der Deploymentdescriptor sieht wie folgt aus:

```
<env-entry>
  <env-entry-name>Price/Algorithmus</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>NoTaxes</env-entry-value>
</env-entry>
```

3.5.4 Handles

Viele EJB Applikationen verlangen, dass Clients die Verbindung trennen dürfen und zu einem späteren Zeitpunkt wiederkehren, um die Bean weiterzubenützen. Der Status der Bean kann festgehalten und beim Reconnect des Clients wiederhergestellt werden.

EJB Object Handles unterstützen diese Anforderung. Sie sind persistente Referenzen auf EJB Objekte.

Code Beispiel :

```
javax.ejb.Handle aHandle = myEJBObject.getHandle();
ObjectOutputStream stream = …
stream.writeObject(myHandle);

// Zeit vergeht...

ObjectInputStream stream = ...
Handle myHandle = (Handle) stream.readObject();

myBeanRemote          myEJBObject          =          (myBeanRemote)
PortableRemoteObject.narrow(myHandle.getEJBObject(),
myBeanRemote.class);
```

Es ist jedoch nicht garantiert, dass solche Handles auf andere Container oder Maschinen portabel sind.

3.6 Transaktionen

3.6.1 Vokabular

Ein **transaktionales Objekt** ist eine Applikations- Komponente, die in einer Transaktion involviert ist. Dies können EJB, eine .NET oder Corba Komponente sein.

Ein **Transaktionsmanager** ist für die Verwaltung von transaktionalen Operationen auf transaktionalen Objekten verantwortlich

Eine **Ressource** ist ein persistenter Speicher, wie eine Datenbank, Queue oder ein anderer Speicher.

Ein **Ressourcemanager** verwaltet Ressourcen. Beispiele sind Datenbanktreiber oder Messagequeuetreiber. Das meist bekannte Interface für Ressourcemanager ist das X/Open XA Ressource Manager Interface. Fast alle Datenbanktreiber unterstützen dieses Interface.

ACID

Acid steht für Atomicity, Consistency, Isolation, Durability.

Atomizität garantiert, dass ein Bündel von Operationen als eine einzige Arbeitseinheit aussehen. Innerhalb einer Transaktion bedeutet dies, dass alle oder keine dieser Operationen ausgeführt werden. (Alles oder nichts Paradigma)

Konsistenz bedeutet, dass eine Transaktion das System von einem konsistenten Status wieder in einen konsistenten Status überführt. Als Konsistenz kann im Falle eines Geldtransfers definiert sein, dass die Summen beider Konten vor und nach der Transaktion gleich sind. Während der Transaktion kann sich das System vorübergehend in einem inkonsistentem Zustand befinden.

Isolation erlaubt mehrfache parallel ausgeführte Transaktionen ohne Auftreten von unerwünschten Nebenerscheinungen wie gegenseitiges Überschreiben von selben Datenobjekten. Jede Operation ist von anderen Operationen durch Synchronisationsmaßnahmen wie Sperrmaßnahmen isoliert.

Dauerhaftigkeit garantiert, dass Operationen vollständig und persistent ausgeführt werden und Systemabstürze und Festplattenfehler keinen Einfluss auf den Datenbestand nehmen.

3.6.2 Transaktionale Modelle

Flache Transaktionen (Flat Transactions)

Eine flache Transaktion ist das einfachste Transaktionsmodell – eine Serie von Operationen als eine Arbeitseinheit zusammengefasst. Sie endet entweder erfolgreich – „commited“ oder fehlerhaft - „aborted“. Im erfolgreichem Falle werden alle persistenten Operationen dauerhafte Veränderungen. Im Fehlerfall werden alle Operationen rückgängig gemacht.

Verschachtelte Transaktionen (Nested Transactions)

Verschachtelte Transaktionen erlauben das Ausführen von Untertransaktionen im Kontext einer Haupttransaktion. Untertransaktionen können bei Fehlschlagen innerhalb eines Zeitrahmens erneut ausgeführt werden, ohne dass bereits erfolgreich ausgeführte Teiltransaktionen rückgängiggemacht werden müssen. Schlussendlich müssen jedoch für das Erfolgreichsein der Haupttransaktion, alle Untertransaktionen erfolgreich abgeschlossen sein.

3.7 Transaktionen und EJB

Enterprise Beans interagieren nie direkt mit einem Transaktions- oder Ressource- Manager. Das Low-Level Transaktionssystem wird durch den Container abstrahiert und Beans bestimmen nur, ob eine Transaktion erfolgreich bestätigt oder abgebrochen werden soll.

3.7.1 Programmatische Transaktionen

Bei Verendung von programmatischen Transaktionen ist der Applikations- Code mit Transaktions- Logik vermischt. Der Programmierer ist für das Absetzen des Begins-, Commit- oder Abort-Statement verantwortlich. Vorteil: Volle Kontrolle, Möglichkeit für Transaktionsserien innerhalb einer Bean Methode

3.7.2 Deklarative Transaktionen

Deklarative Transaktionen erlauben es Komponenten, automatisch an Transaktionen teilzunehmen. Der Container führt statt der Bean die Begin-, Commit- oder Abort-Statements aus. Der EJB Container schaltet sich zwischen den Request an die Bean und beginnt an ihrer Stelle die Transaktion. Der Request wird weiterer Folge an die Bean delegiert, die im Fehlerfall ein Signal an den Container sendet, die Transaktion abzubrechen. Der Vorteil ist, dass Bean Entwickler mit keiner Transaktions- API in Berührung kommen.

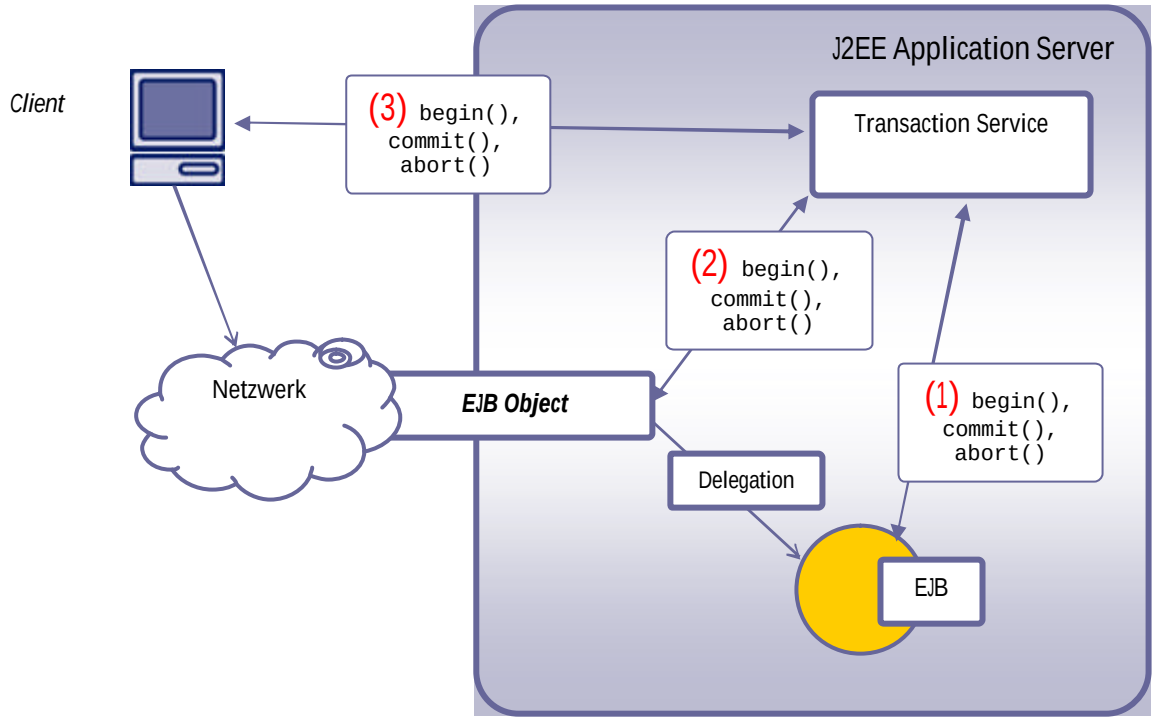
3.7.3 Client- Initiierte Transaktionen

Bei dieser Form interagiert der Client mit dem Transaktions- Service, Begin und Ende der Transaktion werden vom Client bestimmt. Vorteil: Wissen über den Ausgang der Transaktion im Client vorhanden

Abbildung:

a. Programmatische Transaktion,

b. Deklarative Transaktion,



c. Client-initiierte Transaktion

3.7.4 Transaktionen und Entity Beans

Eine Transaktion muss im Falle von Entity Beans folgende Methodenaufrufe umklammern:

```
ejbLoad()

Business-Methoden

ejbStore()
```

Im Falle von bean-managed (programmatische) Transaktionen müsste die Transaktion innerhalb von `ejbLoad()` begonnen und bei `ejbStore()` beendet werden. Da jedoch nicht der Programmierer sonder der Container diese Methoden aufruft, ist nicht gewährleistet, dass sie verbindlich in der richtigen Reihenfolge, bzw. tatsächlich auch beide aufgerufen werden. Der Container ruft `ejbLoad()` und `ejbStore()` nicht bei jedem Businessmethoden Aufruf auf, sondern im Zuge von jeder Transaktion.

Entity Beans müssen daher Deklarative Transaktionen anwenden, Session- oder Messagedriven- Beans können bean-managed Transaktionen verwenden.

3.7.5 Transaktionen und Message- Driven Beans

Bei Verwendung von container-managed Transaktionen gehört das Lesen der Nachricht zur selben Transaktion wie die Business Methode. Im Fehlerfall wird die Transaktion zurückgerollt

Bei bean-managed Transaktionen beginnt und endet die Transaktion und nachdem die Nachricht empfangen wurde. Bestätigungsarten werden über den Deploymentdeskriptor definiert.

3.7.6 Container-Managed Transaktionen

Ein Transaktionsattribut ist eine Einstellung, die das transaktionales Verhalten der Bean bestimmt. Transaktionsattribute werden im Deploymentdeskriptor angeführt und können für gesamte Beans oder individuelle Bean Methoden spezifiziert werden. Falls für die Bean und für eine Methode ein Attribut existiert gelten die methodenbezogenen.

3.7.7 EJB Transaktions- Attribut Werte

`Rquired`

Die Bean läuft immer innerhalb einer Transaktion. Falls die Bean Bestandteil einer übergeordneten Transaktion ist, nimmt sie an dieser Transaktion teil, falls nicht, eröffnet sie eine neue Transaktion.

Beispiel: Eine Worklflow Bean benutzt 2 Bean Komponenten - eine Kreditkartenüberprüfung und eine Produktionsbestellung. Beide Komponenten nehmen an der Transaktion der Workflow Bean teil, jede für sich allein würde eine neue Transaktion beginnen.

`RequiresNew`

Dieses Attribut verlangt, dass immer eine neue Transaktion gestartet wird, wenn die Bean aufgerufen wird. Falls bei Aufruf eine übergeordnete Transaktion läuft, wird diese während des Beanaufrufes angehalten. Der Container bestätigt oder rollt die Transaktion zurück und fährt ggf. mit der übergeordneten Transaktion fort.

`Supports`

Die Bean ist nur dann Bestandteil einer Transaktion, wenn der Client an einer Transaktion beteiligt ist. Die Bean nimmt immer nur an bestehenden Transaktionen teil.

`Mandatory`

Eine Transaktion muss laufen wenn eine Methode der Bean aufgerufen wird. Dies garantiert im Gegensatz zum Wert „Supports“, dass die Bean immer Bestandteil einer Transaktion ist. Der Container startet jedoch keine neue Transaktion, Im Fehlerfall wird eine Exception an den Aufrufer geworfen (`javax.ejb.TransactionRequiredException`)

`NotSupported`

Eine Bean mit dieser Einstellung nimmt an keiner Transaktion teil.

`Never`

Das „Never“ Attribut bedeutet, dass diese Bean an keiner Transaktion teilnehmen kann. Falls diese Bean in einer Transaktion aufgerufen wird, wirft der Container eine Exception an den Client. Dies ist nützlich um Clients von der Nicht-Transaktionalität der Bean zu informieren.

Übersicht über die Transaktionsattribute. T1 ist eine Transaktion eines Clients, T2 die des Containers

Attribut	Client Transaktion	Bean Transaktion
`Requires`	T1	T1
`Requires`	Keine	T2
`RequiresNew`	T1	T2
`RequiresNew`	Keine	T2
`Supports`	T1	T1
`Supports`	keine	keine
`Mandatory`	T1	T1
`Mandatory`	keine	Fehler
`NotSupported`	T1	keine
`NotSupported`	keine	keine
`Never`	T1	Fehler
`Never`	keine	keine

Entity Beans und Stateful- Session Beans müssen Transaktionen verwenden. Daher sind die Attribute `Never`, `NotSupported` und `Supports` nicht erlaubt. Message Driven Beans

werden nicht von Clients aufgerufen, daher machen die Attribute `RequiresNew`, `Supports`, `Mandatory` und `Never` keinen Sinn.

3.7.8 Die Java Transaktions- API (JTA)

JTA ist neben JTS (Java Transaction Service) eine API, die von Komponenten- und Applikationsentwicklern verwendet wird. JTA besteht aus 2 Interfaces: Eines für X/Open XA Ressourcen und eines für programmatische Transaktionskontrolle `javax.transaction.UserTransaction`

Die Methoden sind wie folgt:

`begin()`

Beginnt eine neue Transaktion, die Transaktion wird mit den aktuellen Thread assoziiert.

`commit()`

Startet das 2 Phasen Commit-Protokoll der am aktuellen Thread assoziierten Transaktion.

`getStatus()`

Liefert den Status der mit dem Thread assoziierten Transaktion

`rollback()`

Forciert das Zurückrollen der mit dem Thread assoziierten Transaktion (Bean Managed Transactions)

`setRollbackOnly()`

Forciert das Zurückrollen der mit dem Thread assoziierten Transaktion (Container Managed Transactions)

`setTransactionTimeout(int)`

Der Transaktions- Timeout ist die maximale Zeitspanne, die eine Transaktion dauern darf, bevor sie abgebrochen wird. Dies ist zur Vermeidung von Deadlocks nützlich.

`javax.transaction.Status` Konstanten

`STATUS_ACTIVE`	Eine Transaktion läuft und ist aktiv
`STATUS_NO_TRANSACTION`	Es ist keine Transaktion aktiv
`STATUS_MARKED_ROLLBACK`	Die aktuelle Transaktion wird schlussendlich abbrechen, da sie für ein Rollback markiert ist
`STATUS_PREPARING`	Die aktuelle Transaktion ist „preparing“ für ein Commit
`STATUS_PREPARED`	Die aktuelle Transaktion ist „prepared“
`STATUS_COMMITING`	Die aktuelle Transaktion ist im Prozess des Commit's
`STATUS_COMMITED`	Die aktuelle Transaktion wurde commited
`STATUS_ROLLING_BACK`	Die aktuelle Transaktion ist im Prozess des Zurückrollens
`STATUS_ROLLEDBACK`	Die aktuelle Transaktion wurde zurückgerollt
`STATUS_UNKNOWN`	Der aktuelle Status kann nicht ermittelt werden

3.7.9 „Doomed Transactions“

„Dooming a transaction“ bedeutet eine Transaktion zum Abbruch zu forcieren. Im Falle von Container Managed Transactions ist die Methode `setRollbackOnly()` am EJB Context Objekt auszuführen, um den Container zu veranlassen, die Transaktion abzubrechen. Für selbstinitiierte clientseitig Transaktionen ist `rollback()` zu verwenden, ebenso für Transaktionen, die nicht vom Thread selbst gestartet wurden.

Container Managed Transaction Beans können die Methode `getRollbackOnly()` am EJB Context Objekt aufrufen um eine bereits abgebrochene Transaktion festzustellen, um nicht unnötigen Code auszuführen. Andere Teilnehmer wie Bean Managed Transaction Beans rufen via JTA die `getStatus()` Methode auf.

3.7.10 Transaktionen vom Client Code

Client bedeutet in diesem Zusammenhang alles, was Enterprise Beans aufruft, daher auch Beans selbst.

Code Beispiel:

```
Properties env = new Properties();
…
Context ctx = new InitialContext(env);
UserTransaction    trans    =    (javax.transaction.UserTransaction)
jndiCntx.lookup("java:comp/UserTransaction");
trans.begin() ;
//Business Operationen
```

```
trans.commit();
```

3.7.11 Transaktionale Isolation

EJB Komponenten Entwickler können den Isolationslevel von Transaktionen bestimmen.

Es gibt 4 Transaktions- Isolations- Levels:

`READ UNCOMMITTED Mode:`

liefert keine Isolations- Garantien, jedoch aber die beste Performance. Dieser Modus kann gewählt werden, wenn sichergestellt ist, dass keine Parallelität im Datenzugriff auftritt.

`READ COMMITTED Mode:`

löst das "Dirty-Read" Problem, garantiert Datenkonsistenz bei Leseoperationen

`REPEATABLE READ Mode:`

löst das "Dirty-Read" und das „Unrepeatable-Read" Problem, wiederholtes Lesen innerhalb einer Transaktion liefert immer die selben Werte.

`SERIALIZEABLE Mode:`

löst das "Dirty-Read", das „Unrepeatable-Read" und das „Phantom" Problem. Es ist garantiert, dass keine „uncommited" Daten gelesen werden, wiederholtes Lesen die selben Werte liefert und keine neu eingefügten Daten in der Datenbank aufscheinen.

Das „Dirty Read" Problem:

„Dirty Reads" entstehen, wenn Datenupdates, die noch nicht "commited" wurden, von anderen gelesen werden. Im `READ COMMITED` Modus werden keine Daten gelesen, die geschrieben aber nicht „commited" wurden.

Das „Unrepeatable-Read" Problem

"Unrepeatable Reads" treten auf, wenn eine Komponente Daten ließt, zu einem späteren Zeitpunkt die Daten erneut ließt und die Werte sich verändert haben. Im `REPEATABLE READ` Modus ist gewährleistet, dass wiederholtes Lesen innerhalb einer Transaktion die selben Werte liefert.

Das „Phantom" Problem

Das Phantom Problem tritt auf, wenn innerhalb zweier Leseoperationen neue Daten eingefügt werden und im der zweiten Leseoperation aufscheinen. Der Unterschied zum „Unrepeatable-Read" Problem ist, dass sich nicht bestehende Daten zwischen den Leseoperationen verändern sondern neue Daten aufscheinen. SERIALIZEABLE verhindert das „Phantom" Problem und garantiert volle ACID Eigenschaften.

3.7.12 Isolation und EJBs

Bean Managed Transaktionen: Isolationslevels werden über die Ressourcemanager API konfiguriert. z.B. für JDBC: `java.sql.Connection.setTransactionIsolation(...)`

Container Managed Transaktionen: Der Isolationslevel kann nicht über den Deployment Deskriptor bestimmt werden. Container Tools und Datenbank Tools müssen herangezogen werden.

Pessimistische und optimistische Kontrolle von Nebenläufigkeit

EJB verfolgen 2 Arten von Nebenläufigkeitskontrolle:

pessimistisch: Die EJB sperrt den Zugriff auf Daten bis zum Ende der Transaktion. Vorteil ist der Zugriff auf verlässlichen Daten

optimistisch: Die EJB implementiert eine Strategie um Datenveränderung festzustellen. Sperren sind nur für kurze Momente währen des Zugriffes auf die Datenbank vorgesehen.

3.8 Verteilte Transaktionen

Typische Szenarien für flache verteilte Transaktionen (distributed flat transactions) sind

- Mehrer Applikationsserver, koordiniert in der selben Transaktion
- Datenbankupdates zu unterschiedlichen Datenbanken in einer Transaktion
- Ausführen einer Datenbankoperation und Senden/Empfangen einer Nachricht zu/von einer Message Queue innerhalb der selben Transaktion

3.8.1 Dauerhaftigkeit und das 2 Phasen Commit Protokoll

Dauerhaftigkeit garantiert, dass alle Änderungen an Ressourcen permanent ausgeführt werden. Wenn mehrere Ressourcen Manager involviert sind wird Dauerhaftigkeit mittels Log Journal Dateien und dem 2 Phasen Protokoll gewährleistet.

Im verteilten 2 Phasen Commit existiert ein ausgezeichneter Transaktionsmanager, der „Distributed Transaction Coordinator“ bezeichnet wird. Dieser Koordinator verwaltet alle an der Transaktion beteiligten Transaktionsmanager.

1. Der Transaktions- Koordinator sendet eine „Prepare to commit“ Nachricht an alle involvierten Transaktionsmanager.
2. Jeder Transaktionsmanager propagiert diese Nachricht an den an ihm gebundenen Ressourcemanager
3. Jeder Transaktionsmanager bestätigt an den Transaktions- Koordinator. Falls jeder dem Commit zustimmt, wird die im Begriff befindliche Commit Operation in einer Log Datei protokolliert

4. Der Transaktions- Koordinator gibt das Kommando „Commit“ an alle Transaktionsmanager. Diese wiederum rufen die Ressourcenmanager zum „Commit“ auf. Im Falle eines Absturzes kann an Hand der Log Datei dieser Schritt erneut ausgeführt werden.

Vorraussetzung für den Einsatz des 2 Phasen Commit Protokolls ist das alle beteiligten Transaktionsmanager einem gemeinsamen Kommunikationsprotokoll – dem „transactional communications protokoll“ zustimmen. Da die EJB Spezifikation diese Interoperabilität nicht verpflichtend vorschreibt ist ein Zusammenspiel unterschiedlicher Applicationserver nicht garantiert.

3.8.2 Transaktionen und Stateful Session Beans

Der Abbruch eines Business Prozesses einer Stateless Session Bean (nur eine Business Methode) ist mit dem Werfen einer Exception leicht durchgeführt. Stateful Session Bean Business Prozesse umfassen mehrere unterschiedliche Methodenaufrufe, Stateful Beans befinden sich daher immer in einem Status der Konversation mit dem Client. Eine gut entwickelte Session Bean kann im Falle eines Abbruches einer Transaktion Änderungen ungeschehen machen und den ursprünglichen Status der Bean wiederherstellen.

Eine Stateful Session Bean, die über den Transaktionsstatus informiert sein möchte, implementiert zusätzlich das Interface `javax.ejb.SessionSynchronization`

Die Methoden des Interfaces sind:

`afterBegin()`

Diese Methode wird direkt nach Beginn einer Transaktion vom Container aufgerufen

`beforeCompletion()`

Diese Methode wird genau bevor eine Transaktion beendet ist vom Container aufgerufen

`afterCompletion(boolean)`

Diese Methode wird direkt nach einer beendeten Transaktion vom Container aufgerufen.

Im Falle eines erfolgreichem Commit ist der boolesche Parameter auf wahr gesetzt, im Abort Fall auf falsch. Im Abort Fall kann der Status der Bean zurückgerollt werden.

Code Beispiel einer einfachen Stateful Session Bean, deren Business Methode eine integer Membervariable um 1 inkrementiert.

```
public class CountBean implements SessionBean, SessionSynchronization
{
      public int value;
      public int old_value;

      public int count() { return ++value; }

      public void afterBegin() { old_value = value; }
```

```
      public void beforeCompletion() {}
      public void afterCompletion(commited) {
            if ! (commited) {
                  value = old_value;
            }
      }

      public void ejb*
}
```

Bemerkung: Die Implementierung von SessionSynchronization ist nur für Stateful Session Beans mit deklarativer Container Managed Transaction sinnvoll.

3.9 Persistenz Praktiken

Eine elementares Thema von Enterprise Beans beschäftigt sich damit, auf welche Art die in einer Bean gekapselter Informationen, persistiert werden können.

Herangehensweisen sind:

- Session Beans + JDBC
- Entity Beans, sowohl mit CMP als auch BMP

Kontrolle

Zwischen Session- und Entity Beans existieren Unterschiede in der Persistenz- Kontrolle. Session Beans sind serviceorientiert und bieten aufrufbare Methoden zum explizitem Laden und Speichern an. Dies ist dem Microsoft Ansatz von Business Komponenten sehr ähnlich. Bei Entity Beans hingegen bestimmt der Container den Zeitpunkt des Ladens und Speicherns der Daten, was einen Verlust an Kontrolle bedeutet.

Performance und Analogie der Parameterübergabe

Im Falle einer Session Bean liefert eine Abfrage typischerweise einen Resultset an den Client. Dies ist ähnlich zu „Pass by value“ (Werteübergabe). Eine Abfrage einer Entity Bean liefert hingegen Stubs zu serverseitigen Objekten, was dem Typ „Pass by reference“ entspricht. In einem Szenario in dem Session Beans Clients von Entity Beans sind und über die lokalen Interfaces kommunizieren hat die Übergabe von serverseitigen Stubs keinen gravierenden Einfluss auf die Performance. Ist der Client jedoch eine GUI Applikation bedeutet die Übergabe der Referenzen über RMI IIOP eine Performance Einbuße. Diese Situation kann jedoch mit Session Beans, die Wrapper für Entity Beans bilden und dem Client serialisierte Java Objekte senden verbessert werden.

Caching

Middle-tier caching ist ein wichtiger Mechanismus um Datenbank Datentransfer zu reduzieren und die Performance zu erhöhen. Session Beans repräsentieren keinen Cache zur Datenbank. Die Datenzeilen, die sie präsentieren sind nur für die Dauer einer Transaktion konsistent. Entity Beans Daten können über mehrere Transaktionen gecached sein, wenn der Applicationserver exklusiven Zugang zur Datenbank hat. Der Vorteil ist gegeben, wenn

die Daten gemeinsam genutzt sind und Leseoperationen die Mehrheit bilden. (z.B.: Die 100 meistverkauften Produkte)

3.9.1 Die Wahl zwischen CMP und BMP

Code Reduktion und schnelles Entwickeln von Applikationen

CMP bietet einen großen Fortschritt in der Reduktion von Code und in der Software Entwicklungszeit. Der Deploymentdescriptor konfiguriert das Verhalten der Bean und der Container implementiert sämtlichen Datenbankcode.

Performance

CMP Entity Beans liefern sofern sie gut konfiguriert sind eine bessere Performance als BMP Entity Beans. Mit BMP und x bean-managed Entity Beans erfolgen x + 1 (ein Finder Methodenaufruf + x Load Aufrufe) Aufrufe, um die Daten zu laden. Mit CMP reduziert der Container den Aufwand auf einen einzelnen Aufruf mit einem umfangreichen SELECT Statement.

Fehler

CMP Entity Beans sind schwerer zu debuggen als BMP Entity Beans, da der BMP Entwickler die komplette Kontrolle über den JDBC Code hat. Im Fehlerfall ist es bei CMP Beans schwer zu analysieren, was im Datenbankzugriffscode vorgeht.

Beziehungen

Das EJB 2.0 CMP Modell bietet einige Möglichkeiten wie referentielle Integrität, Kardinalität, Beziehungsmanagement für Daten-Beziehungen an. Mit BMP kenn es aufwendig sein all diese Beziehungen korrekt zu implementieren.

3.9.2 Die Wahl zwischen Stateful oder Stateless Session Beans

Statelessness hat 2 Vorteile gegenüber Statefulness

- Der EJB Container kann stateless Session Beans sehr leicht in einem Pool verwalten und eine geringe Anzahl von Stateless Beans kann eine große Anzahl Clients bedienen. Dies trifft auch für Stateful- Beans zu, jedoch kann der Passivierungsmechanismus bei Erreichen des Instanzenlimits zum I/O Flaschenhals werden.

- Mit Stateful Beans geht der Konversationsstatus bei Beanfehlern oder Abstürzen verloren. Im stateless Modell können Requests einfach an andere Komponenten oder Applicationserver gerichtet werden.

Der größte Nachteil von Statelessness ist, dass für jeden Methodenaufruf clientspezifische Informationen in die Stateless Bean gebracht werden müssen. Diese Informationen können über die Parameter der Methode übergeben werden, wobei die Datengröße der Parameter aus Performancegründen gering zu halten ist. Aus diesem Grund kann der Status der Bean auch in einem persistenten Speicher abgelegt werden. Ein Identifier (primary Key), der der Business-Methode übergeben wird identifiziert den zu speichernden/ladenden Status der Bean. Eine weitere Möglichkeit besteht die über den Identifier unterscheidbaren Clientdaten

mittels JNDI in einem Directory Service abzulegen. Ein Vorteil zur Datenbankspeicherung ergäbe sich jedoch nur, wenn der Verzeichnisdienst eine „in-memory“ Implementation ist.

3.10 EJB Security

Authentication (Authentifizierung):

Authentifizierung ist der Vorgang zur Verifizierung der Identität eines Benutzers. In der Regel ist für diesen Vorgang eine der folgenden Informationen notwendig: Etwas, was ein Benutzer weiß (Passwort), was er besitzt (Chip-Karte) oder was er ist (Biometrik). Sobald ein Client einer EJB authentifiziert ist, ist im für den Rest der Session eine „Security Identität" zugeordnet

Authorization (Autorisierung):

Autorisierung bezieht sich in der Regel auf Zugriffserlaubnis oder Verweigerung zu Ressourcen, nachdem die Identität eines Benutzers bestimmt ist. Wer auf welche Ressourcen zugreifen darf ist meist mit ACLs (Access-Control-Lists) modelliert. Grundlage für korrekte Autorisierungsentscheidungen ist die korrekte Authentifizierung.

Auditing (Prüfung):

Auditing ist die Protokollierung von Authentifizierungs- und Autorisierungsvorgängen. Sie dient der nachträglichen Kontrolle über Aktionen, die am System stattgefunden haben.

Encryption (Verschlüsslung):

Verschlüsselung ist der Vorgang, Daten temporär unleserlich zu machen, um Vertraulichkeit zu gewährleisten. In Kerberos kommen DES (Data-Encryption-Standard), Tripple DES und AES (Advanced-Encryption-Standard) zum Einsatz.

3.10.1 Ein Überblick über JAAS (Java Authentication and Authorization Service)

JASS ist ein Java Package, das Authentifikation und Autorisation ohne detailliertes Wissen über die plattformspezifischen Implementierungen der genannten Dienste ermöglicht. Der Vorteil von JAAS liegt in seiner Fähigkeit fast jedes Security System, sei es User/Passwort Listen, Directory Services oder Zertifikat basierende Authentifikation zu unterstützen.

In der Regel gibt es zwei typische Einsatzszenarien für JAAS:

Standalone Applications, die EJBs aufrufen. Die Applikation verwendet die JAAS API, um den Benutzer vor Aufruf der EJB zu authentifizieren. Der Applicationserver verifiziert die Berechtigungsnachweise (Credential).

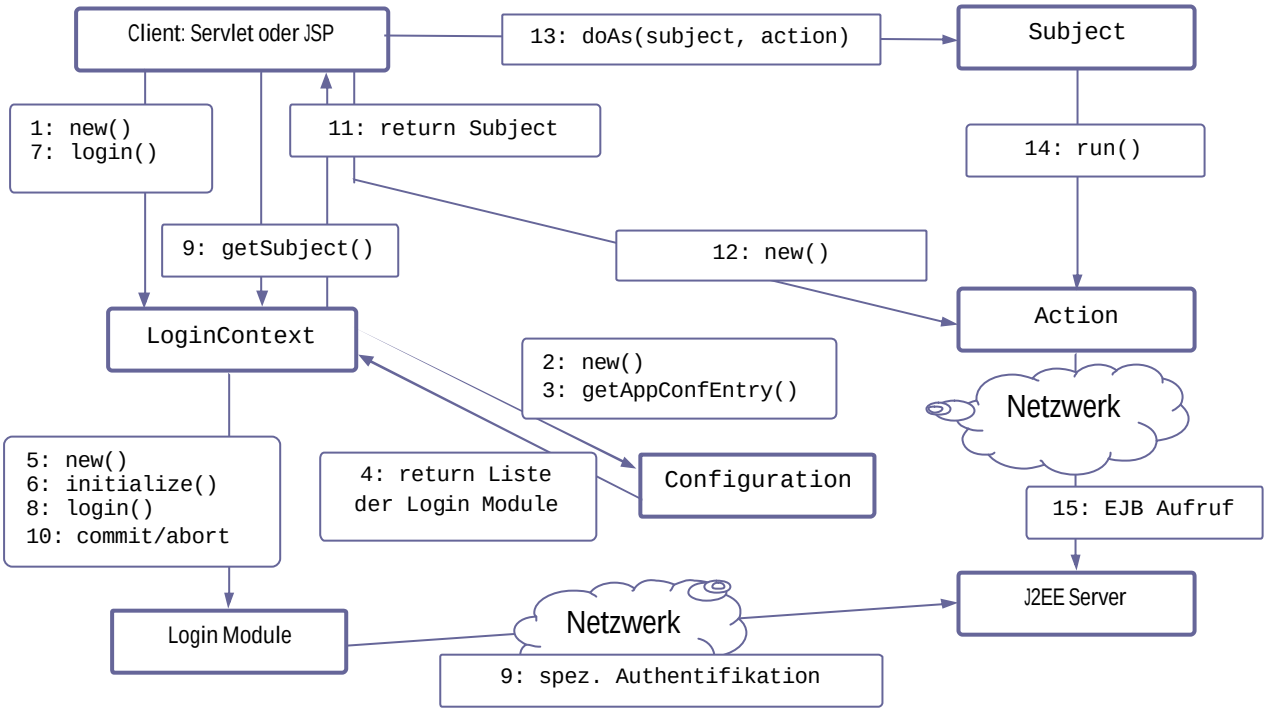
Webbrowser Clients, die sich zu einem Servlet oder einer JSP Seite verbinden. Die Servlet/JSP Schicht verwendet JAAS für die Authentifikation. Der Webbrowser bietet folgende Arten der Benutzername/Kennwort Übergabe an:

- Basic Authentication: Der Browser fordert den Benutzer zur Eingabe auf, der Webserver verifiziert die Berechtigungsnachweise (SSL ist für Vertraulichkeit Voraussetzung)

- Formularbasierende Authentifikation: Übergabe der Informationen mittels HTML Formular

- Digest Authentication: Der Browser übermittelt einen Hash Wert der Informationen, der am Server mit dem dort gespeicherten Hash verglichen wird.
- Zertifikatbasierende Authentifikation: X.509 Zertifikate werden übermittelt

3.10.2 Die JAAS Architektur



1. Der Client instanziert ein `LoginContext` Object. Es ist für den Prozess der Authentifikation verantwortlich.
2. Die `LoginContext` Instanz instanziert ein `Configuration` Objekt, das über die Art der Authentifikation informiert ist (zB Passwort oder Zertifikat)
3. Die `LoginContext` Instanz befragt die `Configuration` Instanz nach einer Liste unterstützter Authentifikationsmechanismen
4. Das `Configuration` Objekt liefert eine Liste von Authentifikationsmechanismen, die als Module bezeichnet werden. Jedes der Module implementiert eine spezifische Art der Authentifikation
5. Der `LoginContext` instanziert die Login Module.
6. Der `LoginContext` initialisiert die Login Module
7. Der Client startet den Loginversuch mit Aufruf der Methode `Login()`
8. Der LoginContext delegiert den `Login()` Aufruf an die Login Module
9. Das Login Modul authentifiziert auf seine spezielle Weise

10. Im Falle einer erfolgreichen Authentifikation erhält das Modul die Anweisung „commit“ andernfalls „abort“

11. Der Client erhält eine `Subject` Instanz. Ein `Subject` Objekt repräsentiert jemand oder etwas, dass authentifiziert wurde.

12. Der Client instanziert eine neues `Action` Objekt. Ein `Action` Objekt ist mit Informationen zur Ausführung der gewünschten Operation ausgestattet (Aufruf einer EJB, Datenbankoperation...)

13. Das `Subject` Objekt wird angewiesen die `Action` als `Subject` auszuführen – `doAs()` Methode

14. Die `Subject` Instanz ruft die Methode `run()` des `Action` Objektes auf.

15. Die `Action` Instanz führt zusammen mit dem LoginContext die gewünschte Operation aus. Der Applicationserver kann anhand des Contextes eine Autorisationsprüfung ausführen

3.10.3 Autorisierung

Mit EJB gibt es 2 Arten der Autorisierung. Mit programmatischer Autorisierung sind Sicherheits- Überprüfungen im Bean-Code implementiert, mit deklarativer Autorisierung werden diese Checks durch den Container ausgeführt.

Security Roles

Um Flexibilität zu gewährleisten werden eine oder mehrer Identitäten zu Security Roles assoziiert. Der EJB Deployer ist vor Installation der Bean im Applicationserver für die Zuordnung der Identitäten zu den Rollen verantwortlich. Diese Abstraktion verhindert, dass konkrete Benutzer Identitäten im Bean Code kodiert sind.

Programmatische Autorisierung

Das `EJBContext` Objekt stellt 2 Methoden zur Bestimmung des Aufrufers der Bean zur Verfügung:

```
public interface javax.ejb.EJBContext
{
       …
       public java.security.Principal getCallerPrincipal();
       public Boolean isCallerInRole(String roleName);
       …
}
```

Die Methode `isCallerInRole()` überprüft, ob der aufrufende Client Teilnehmer der angegebenen Rolle ist.

Die Methode `getCallerPrinicipal()` liefert das Principal Objekt des aufrufenenden Client.

Beispielkode einer EJB

```
public class EmployeeManagementBean implements SessionBean {

	private mySessionContext ctx;

	public void modifyEmplyee(String employeeID) throws SecurityException
{
		…
		if (!ctx.isCallerInRole(“administrator”)) {
			throw new SecurtiyException();
		}

	}
}
```

4 Die Java Connector Architektur (JCA)

4.1 Überblick

Die J2EE Connector Architektur definiert eine standard Architektur um die J2EE Plattform an heterogene EIS Systeme anzubinden. Beispiele für EIS Systeme sind Datenbanken, ERP (Enterprise Ressource Planing) Systeme, Mainframe Transaktions- Systeme wie CICS oder SAP, oder Middleware Produkte wie Corso. Die Architektur erlaubt es einem EIS Hersteller einen standard Resource Adapter herzustellen, der in jedem J2EE kompatiblen Applicationserver integrierbar ist. Ein Resource Adapter stellt somit einen system-software bezogenen Treiber dar, der eine Java Applikation oder EJBs mit dem EIS verbindet.

4.1.1 JCA Spezifikations- Versionen

Version: 1.0 (Juli 2001)

- Eine Menge von standard systemebene bezogenen Kontrakten zwischen einem J2EE Application Server und einem EIS (Enterprise Information System). Diese Kontrakte beziehen sich auf das Management von Verbindungen, Transaktionen und Sicherheit.
- Ein allgemeine Client Schnittstelle (Common Client Interface CCI) die eine Client API definiert, um auf ein EIS zuzugreifen
- Ein standard Deployment und Packaging Protokoll

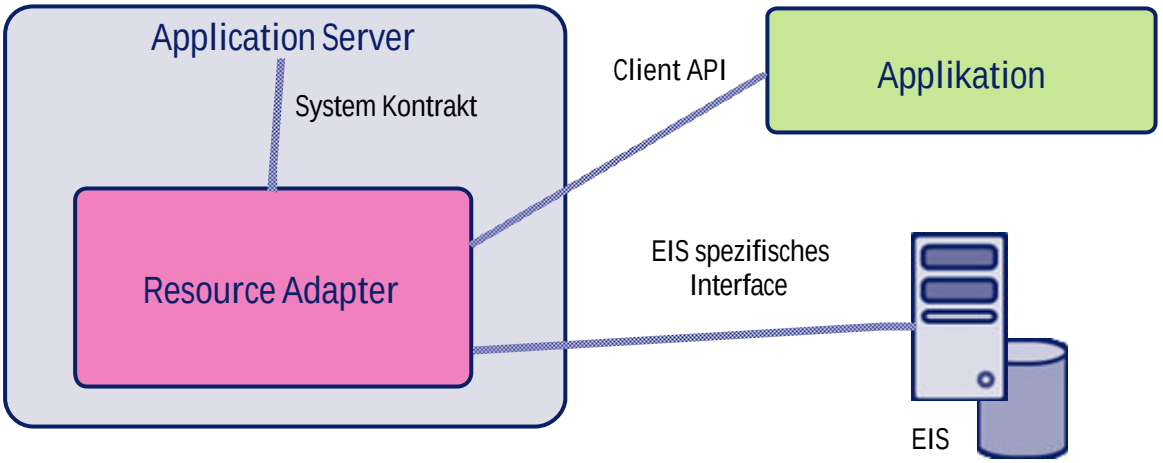
Version: 1.5 (November 2003)

- Einen Kontrakt für den Lebenszyklus eines Resource Adapters. Dieser Kontrakt ermöglicht es einen Resource Adapter zu „bootstrappen“, zu starten, zu stoppen, zu notifizieren und herunterzufahren
- Work Management: Ein Kontrakt zwischen Application Server und Resource Adapter, der es dem Adapter ermöglicht „Work“ Instanzen dem Application Server zu übergeben. Der Application Server ist für Threadding, Thread-Pooling und Security zuständig.
- Kontrakt für eingehende Transaktionen: Einem Ressource Adapter ist es möglich, eine importierte Transaktion an den Applicationserver zu propagieren, das Beenden von Transaktionen zu übertragen und Crash-Recovery Aufrufe des EIS's auszuführen
- Kontrakt für eingehende Nachrichten: Einem Adapter ist es möglich, beliebige Nachrichten asynchron an Nachrichtenendpunkte im Application Server zu senden

4.1.2 Implementierte Version

In dieser Arbeit wurde die Spezifikation Version 1.0 implementiert. Zum Zeitpunkt des Schreibens unterstützte kein Hersteller J2EE konformen Applicationserver die Version 1.5.

4.1.3 *Die Connector Architektur*



4.1.4 *System Kontrakt*

Ein RA ist spezifisch für ein EIS. Er ist ein system-level Softwaretreiber, der eine Verbindung von einer Applikation oder dem Application Server zum EIS ermöglicht und im Adressbereich eines Application Servers läuft. Der System Kontrakt ermöglicht die Portabilität des RA's zu Application Servern beliebiger Hersteller.

Ein RA kann folgende unterschiedliche Typen der Verbindung unterstützen:

- ausgehend
- Ab Version 1.5: eingehend. Der Adapter erlaubt es dem EIS Applikations-Komponenten aufzurufen. Die Kommunikation ist vom EIS initiiert
- Ab Version 1.5: bidirektional

Die Connector Architektur definiert folgende Systemkontrakte zwischen dem Application Server und dem RA:

- Ein Connection Management Kontrakt, der es dem Applicationserver erlaubt Verbindungen zum EIS herzustellen und diese in einem Connectionpool zu verwalten
- Ein Transaktionsmanagement Kontrakt sowohl für den Einsatz von externen Transaktionsmanagern als auch für lokale Transaktionen, die intern vom EIS Resource Manager verwaltet werden
- Ein Sicherheits- Kontrakt für sicheren Zugang zum EIS
- Ab Version 1.5:
  Management für den Lebenszyklus eines RA's (Boot, Start, Stop, Shutdown)
- Ab Version 1.5:
  Work Management: Ein Kontrakt zwischen Application Server und Resource Adapter,

der es dem Adapter ermöglicht „Work“ Instanzen dem Application Server zu übergeben

- Ab Version 1.5:
  Ein Kontrakt, um eingehende Transaktionen weiter an den Applicationserver zu propagieren

- Ab Version 1.5:
  Asynchrone Nachrichten Auslieferung von Nachrichten an Nachrichten Endpunkte des Application Servers

4.1.5 Client API

Die Connector Architektur verlangt, dass ein RA Adapter die System Kontrakte unterstützt und empfiehlt (verlangt nicht) das Common Client Interface (CCI) zu unterstützen.

CCI ist designed um folgende Ziele zu ermöglichen

- Es definiert ein „remote-function-call“ Interface, das auf einen Funktionsaufruf am EIS fokussiert ist.

- Es richtet sich vorwiegend an Applikations- Entwicklungswerkzeuge und EAI Frameworks

Im Zuge dieser Arbeit wurde der verbindungsbezogene Teil des CCI implementiert. Dieser umfasst folgende Interfaces:

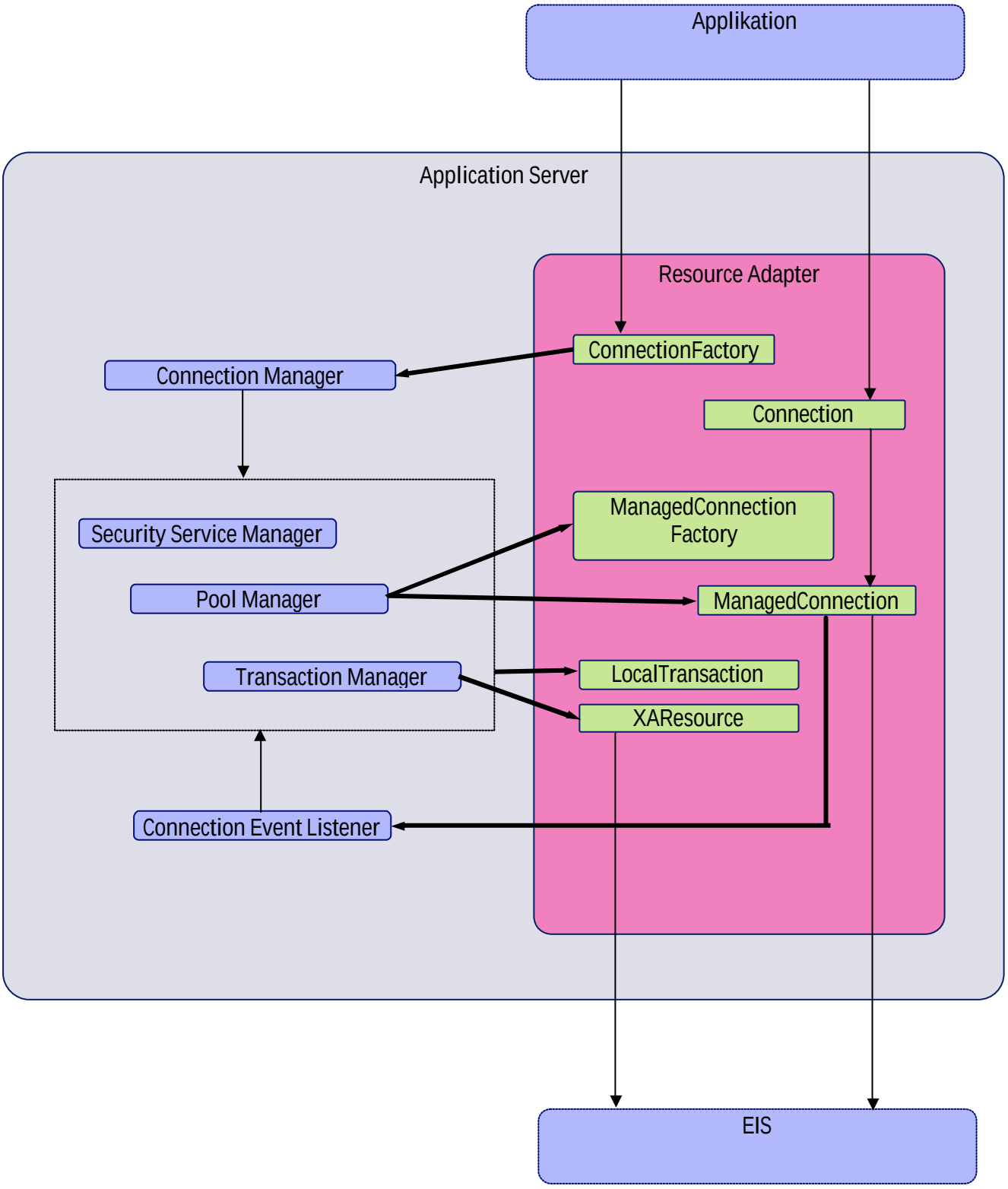
```
javax.resource.cci.ConnectionFactory
javax.resource.cci.Connection
javax.resource.cci.ConnectionSpec
javax.resource.cci.LocalTransaction
```

4.2 Connection Management

4.2.1 Überblick

Eine Applikation verwendet eine „Connection Factory“ um Verbindungen zum EIS herzustellen. Ein RA agiert als solch eine „Factory“ und implementiert die Mechanismen, die es dem Applicationserver ermöglichen, physikalische Verbindungen des RA’s zum EIS wiederzuverwenden (Connectionpooling).

4.2.2 Architektur Connection Management



Eine Applikation lokalisiert mittels JNDI eine `ConnectionFactory` für das EIS. Sie verwendet die Methode `getConnection` der `ConnectionFactory` um eine Verbingung zum EIS zu erhalten.

Die `ConnectionFactory` delegiert das Erzeugen einer Verbindung an die Instanz des `ConnectionManager` des Applicationservers. Der `ConnectionManager` erlaubt dem ApplicationServer Dienste wie Transaktionsmanagement, Sicherheit, Protokollierung, Connectionpooling zu erfüllen. Die `ConnectionManager` Instanz sucht nach einer passenden Verbindung in seinem Connection Pool.

Falls keine passende Verbindung in Benzug auf die Requestparameter gefunden werden kann verwendet der Application Server die `ManagedConnectionFactory` Instanz des RA's um eine neue physische Verbindung zum EIS herzustellen. Die neu erzeugte `ManagedConnection` wird in den Connectionpool aufgenommen. Der Applicationserver registriert einen `ConnectionEventListener` mit der `ManagedConnection` Instanz, um Event Notifikationen bezüglich des Status der `ManagedConnection` Instanz zu erhalten. Diese Notifikationen ermöglichen es dem Application Server Services wie Transaktionen, Pooling, Fehlerbehandlung und das Aufräumen von Verbindungen zu verwalten.

Der RA implementiert das `XAResource` Interface um verteilte Transaktionen zu unterstützen. Ein RA implementiert auch das `LocalTransaction` Interface, um EIS kontrollierte Transaktionen zu unterstützen.

4.2.3 Managed Application Szenario

Ein Applikationsentwickler spezifiziert Anforderungen an die ConnectionFactory im Deploymentdescriptor einer EJB wie folgt:

```
res-ref-name: eis/corso
res-type: javax.resource.cci.ConnectionFactory
res-auth: application oder container
```

Er lokalisiert eine ConnectionFactory Instanz via JNDI mit:

```
Context ctx = new InitialContext();

javax.resource.cci.ConnectionFactory cf =
      (javax.resource.cci.ConnectionFactory)
            ctx.lookup(„java.comp/env/eis/corso“);
```

Eine Applikation erhält eine Verbindung schlussendlich mit

```
javax.resource.cci.Connection con = cf.getConnection();
// Programmcode
cx.close();
```

4.2.4 Der Lookup einer ConnectionFactory mittels JNDI:

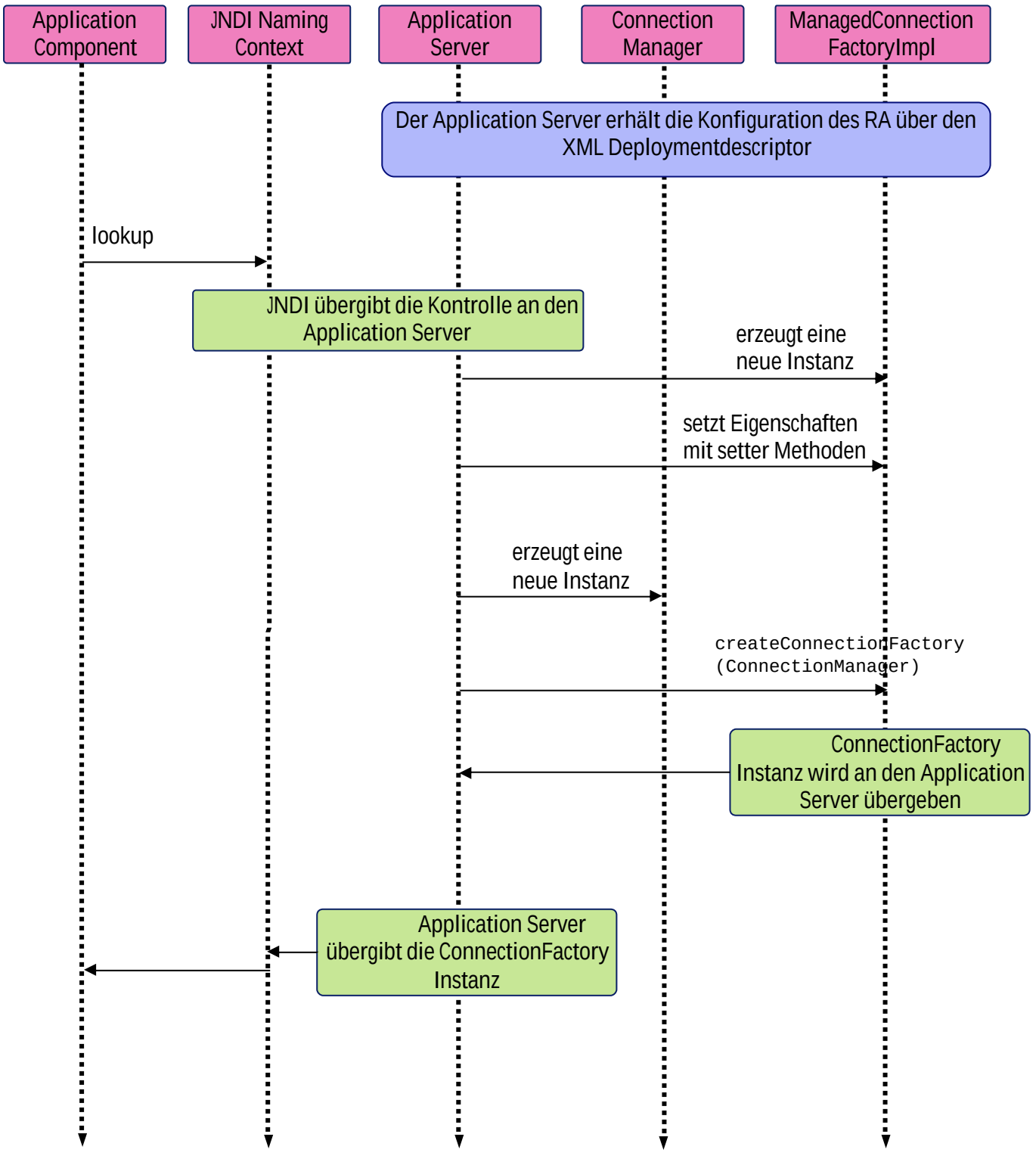
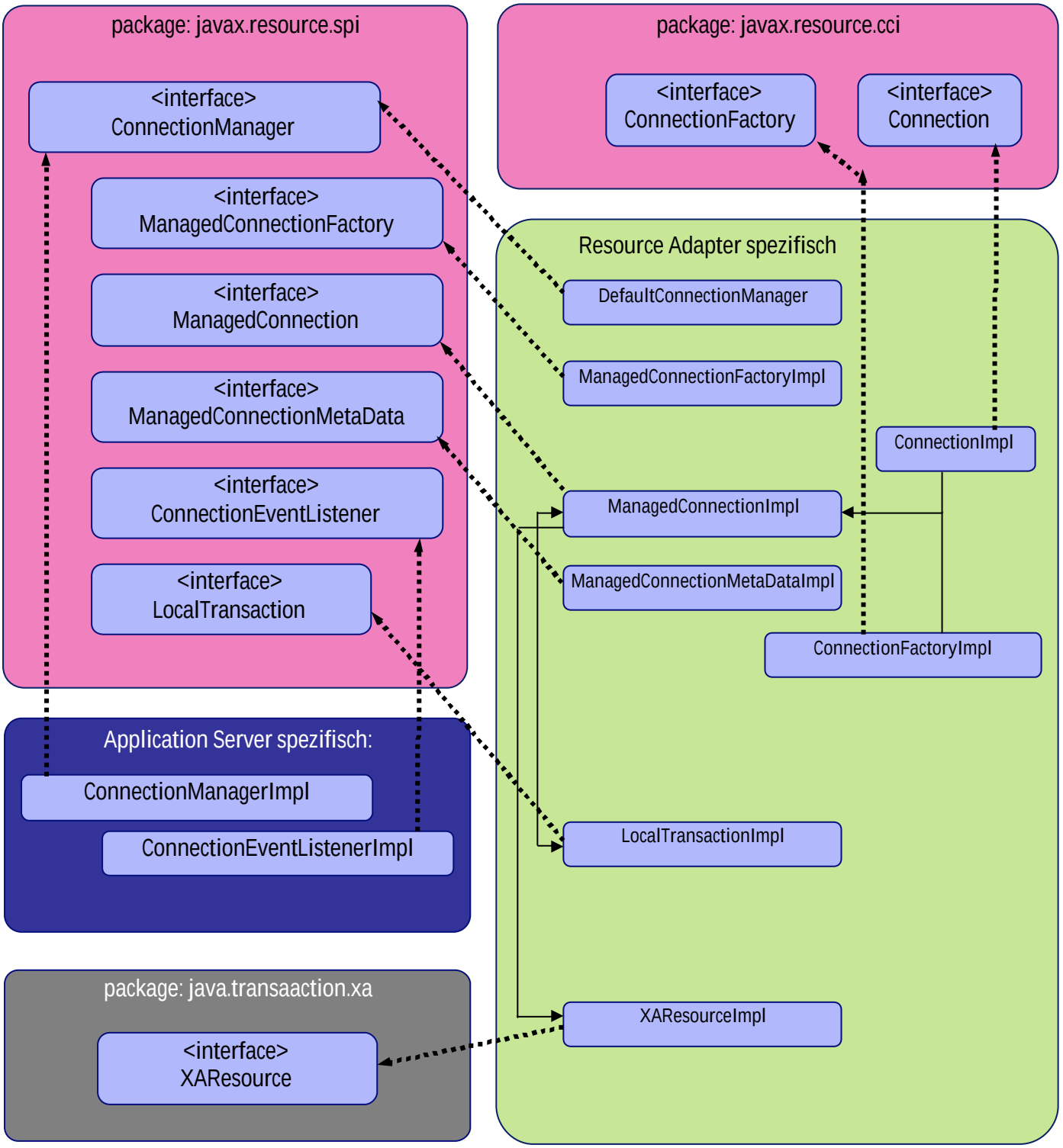


Abbildung 3

4.2.5 *Übersicht über die Interface/Klassen Spezifikation*



Legende:

implementiert

in Beziehung

4.2.6 Das ConnectionFactory Interface

```
public interface javax.resource.cci.ConnectionFactory extends java.io.Serializable,
        javax.resource.Referenceable
{
        public RecordFactory getRecordFactory() throws ResourceException;

        public Connection getConnection() throws ResourceException;
        public Connection getConnection(javax.resource.cci.ConnectionSpec properties)
                throws ResourceException;

        public ResourceAdapterMetaData getMetaData() throws ResourceException;
}
```

Eine Implementierung des `ConnectionFactory` Interfaces wird vom RA zur Verfügung gestellt. Sie implementiert **`java.io.Serializable`** und **`javax.resource.Referenceable`** um eine JNDI Registrierung zu ermöglichen. Eine Komponente lokalisiert eine ConnectionFactory via JNDI und erhält mit der Methode `getConnection` eine Verbindung zum EIS.

Die Variante der Methode `getConnection()` ohne Parameter wird verwendet, wenn der Container für das „sign-on“ zuständig ist. Die Variante mit dem Parameter `javax.resource.cci.ConnectionSpec` wird für die Übergabe von Verbindungsparametern wie z.B. Benutzername/Kennwort verwendet. Die übergebenen Informationen sollen client-spezifisch sein und keine Informationen bezüglich EIS Server Adresse/Port enthalten. Die ManagedConnectionFactory ist hierfür mit den notwendigen Parametern zum Verbindungsaufbau ausgestattet.

4.2.6.1 Die JcaCorsoConnectionFactory.getConnection Methode

Folgender Codeausschnitt zeigt die Delegation der `getConnection` Methode an den der ConnectionFactory assoziierten `ConnectionManager`. Die `ConnectionManager` Instanz wird zusammen mit der `ManagedConnectionFactory` Instanz bei Instanzierung der `ConnectionFactory` übergeben. Die Delegation erfolgt durch den Aufruf der Methode `allocateConnection` an der `ConnectionManager` Instanz:

```
public Connection getConnection(ConnectionSpec arg0)
                throws ResourceException {

                Connection con = null;

                if (connectionManager == null) {
                        throw new ResourceException("connectionManager instance is
                                null");
                }

                JcaCorsoConnectionSpec jcaCorsoConnectionSpec =
                        (JcaCorsoConnectionSpec) arg0;

                JcaCorsoConnectionRequestInfo jcaCorsoConnectionRequestInfo;

                try {
                        jcaCorsoConnectionRequestInfo =
                                new JcaCorsoConnectionRequestInfo(
                                        jcaCorsoConnectionSpec.getUserName(),
                                        jcaCorsoConnectionSpec.getPassWord());

                        con =
                                (Connection) connectionManager.allocateConnection(
                                        managedConnectionFactory,
                                        jcaCorsoConnectionRequestInfo);

                        return con;
                }
```

```
                catch (Exception e) {
                        e.printStackTrace();
                        throw new ResourceException(e.toString());
                }

        }
```

Voraussetzungen:

Eine Implementierung von `ConnectionFactory` muss einen default Konstruktor zur Verfügung stellen und `javax.resource.Referenceable` und `java.io.Serializable` implementieren

4.2.7 Das ConnectionSpec Interface

Das interface javax.resource.cci.ConnectionSpec wird von einer Applikation verwendet, um verbindungsrelevante Parameter beim Aufruf von getConnection zu übergeben. Die Verbindungsparameter des `ConnectionSpec` Interface beziehen sich auf Applikationsebene (definiert unter dem Rahmen von CCI), z.B. Benutzername und Kennwort während das `ConnectionRequestInfo` Interface sich auf die Ebene des System Kontraktes (SPI) bezieht (EIS Host, EIS Port).

4.2.7.1 Die JcaCorsoConnectionSpec Klasse

Corso Verbindungsparameter umfassen neben Benutzername und Kennwort auch die CorsoSite, den Port, die Strategy, die Aid, den Servicenamen und die Domain. Diese Verbindungsparameter sind jedoch Systemkontrakt bezogene Parameter und daher im `ConnectionRequestInfo` Interface verankert. Das Mapping des `ConnectionSpec` Interface zum `ConnectionRequestInfo` Interface erfolgt in obiger `getConnection` Methode.

```
public class JcaCorsoConnectionSpec implements ConnectionSpec {

        private String userName = null;
        private String passWord = null;

        public JcaCorsoConnectionSpec() {
                userName = "";
                passWord = "";

        }

        public JcaCorsoConnectionSpec(String userName, String passWord) {

                this.userName = userName;
                this.passWord = passWord;
        }


        getter/setter Methoden…
}
```

4.2.8 Das Connection Interface

```
public interface javax.resource.cci.Connection {
        public Interaction createInteraction() throws ResourceException;
        public ConnectionMetaData getMetaData() throws ResourceException;
        public ResultSetInfo getResultSetInfo() throws ResourceException;
        public LocalTransaction getLocalTransaction() throws ResourceException;
        public void close() throws ResourceException;
}
```

Eine Connection Instanz stellt einen applikationsbezogenen Handle auf eine Verbindung zum EIS dar. Die tatsächliche Verbindung ist durch `ManagedConnection` repräsentiert. Eine `Connection` Instanz kann (muss aber nicht) mit ein oder mehreren `Interaction` Instanzen assoziiert sein, wobei eine Instanz von `Interaction` einer Applikation den Zugriff auf EIS Daten und Funktionen ermöglicht.

Die Methode `getMetaData` liefert Informationen über die EIS Instanz die der `Connection` Instanz zugeordnet ist, z.B. Anzahl der max. Verbindungen, EIS Produktname und Version.

Die Methode `getResultSetInfo` liefert Information über die Funktionalität von Result Sets. Falls die Implementation des CCI keine Result Sets unterstützt muss eine `NotSupportedException` Exception geworfen werden.

Die `close` Methode initiiert das Schließen einer Verbindung. Die `ManagedConnection` Instanz notifiziert alle registrierten `ConnectionEventListener` mit einem `ConnectionEvent`: CONNECTION_CLOSED. Der Application Server ruft an der `ManagecConnection` Instanz die Methode `cleanup` auf und fügt die Verbindung zurück in den Connection Pool. Diese Methode muss vom RA implementiert werden.

Die Methode `getLocalTransaction` liefert eine LocalTransaction Instanz, die vom EIS kontorlliert wird. Falls dies nicht unterstützt wird, muss eine `NotSupportedException` Exception geworfen werden.

4.2.8.1 Die JcaCorsoConnection Klasse

```
public class JcaCorsoConnection implements Connection {

        static Category log = Category.getInstance(JcaCorsoConnection.class);

        private JcaCorsoManagedConnection jcaCorsoManagedConnection = null;

        private boolean isValidConnectionHandle;

        protected JcaCorsoConnection(JcaCorsoManagedConnection
                jcaCorsoManagedConnection) {
                this.jcaCorsoManagedConnection = jcaCorsoManagedConnection;
                isValidConnectionHandle = true;
        }


        public void close() throws ResourceException {

                log.debug("close() called on [" + this +"]");
                jcaCorsoManagedConnection.close(this);
        }


        public LocalTransaction getLocalTransaction() throws ResourceException {

                return new JcaCorsoLocalTransaction(jcaCorsoManagedConnection);
        }
```

```
        public CorsoTopTransaction getContainerManagedCorsoTopTransaction() {

                log.debug(
                        "getContainerManagedCorsoTopTransaction() was called on ["
                                + this
                                + "]");
                return jcaCorsoManagedConnection.getCorsoTopTransaction();
        }


        protected JcaCorsoManagedConnection getJcaCorsoManagedConnection() {
                return jcaCorsoManagedConnection;
        }

        // Used by ManagedConnection.associateConnection

        protected void setJcaCorsoManagedConnection(JcaCorsoManagedConnection
                connection) {
                jcaCorsoManagedConnection = connection;

        }

        protected boolean isHandleValid() {
                return isValidConnectionHandle;
        }

        protected void setHandleInvalid() {
                isValidConnectionHandle = false;
        }

        protected void setHandleValid() {
                isValidConnectionHandle = true;
        }

        ////////////////////////////////////////////////////////////////


        public CorsoConstOid createConstOid(CorsoStrategy arg0)
                throws CorsoException {

                log.debug("createConstOid() was called on [" + this +"]");

                return jcaCorsoManagedConnection.corsoConnection.createConstOid(arg0);
        }


        public static int NO_TIMEOUT = CorsoConnection.NO_TIMEOUT;
        public static int INFINITE_TIMEOUT = CorsoConnection.INFINITE_TIMEOUT;

}
```

Die `JcaCorsoConnection` ist ein Connection Handle auf die zugehörige physikalische Verbindung, die durch eine `ManagedConnection` Instanz repräsentiert ist. Die assoziierte `ManagedConnection` wird im Konstruktor übergeben. Ein Aufruf der Methode `close()` schließt nicht die physikalische Verbindung, sondern deligiert das Schließen an die `ManagedConnection`.

Die Methode `getLocalTransaction`() liefert eine `JcaCorsoLocalTransaction` Instanz, die das `javax.resource.cci.LocalTransaction` Interface implementiert. Eine `cci.LocalTransaction` repräsentiert eine Bean kontrollierte Transaktion im Gegensatz zur `spi.LocalTransaction`, die vom Container initiiert und commited wird. (Deployment Deskriptor `<transaction-type>Container</transaction-type>`).

Die Methode `getContainerManagedCorsoTopTransaction()` liefert bei containermanaged Transaktionen das `CorsoTopTransaction` Objekt einer durch den Container geöffneten `spi.LocalTransaction`.

Die `JcaCorsoConnection` „wrapped“ sämtliche Methoden der `corso.lang.CorsoConnection` wie z.B. die Methode `public CorsoConstOid createConstOid(CorsoStrategy arg0`. Sie delegiert den Methodenaufruf an die `corso.lang.CorsoConnection`, die die tatsächliche physikalische Verbindung zum Corso Kernel als Membervariable der ManagedConnection hält.

Richtlinien und Voraussetzungen

- Das `Connection` Interface muss die Methode `close` implementieren
- Ein RA muss das `Connection` und das `ConnectionFactory` Interface implementieren
- Die ConnectionFactory Implementierung delegiert den `getConnection` Methodenaufruf an den der Factory assoziierten ConnectionManager. Die `ConnectionManager` Instanz wird zum Zeitpunkt der Instanzierung zur `ConnectionFactory` assoziiert. Siehe Abbildung 3
- Die ConnectionFactory Instanz ist für das Übernehmen der Connection Request Informationen (`ConnectionRequestInfo`) verantwortlich und muss die Methode `ConnectionManager.allocateConnection` im selben Thread Context der `getConnection` Methode der Applikation aufrufen.

4.2.9 ConnectionRequestInfo

Ein RA implementiert das `ConnectionRequestInfo` Interface um die für das EIS spezifischen Connection Parameter als Datenstruktur für einen Connection Request zu kapseln.

Wenn `ConnectionRequestInfo` Instanzen die Methoden `createManagedConnection` oder `matchManagedConnection` der `ManagedConnectionFactory` erreichen, werden diese per-request bezogenen Informationen vom RA herangezogen um entweder eine neue Verbindung zu erzeugen bzw. eine bestehende Connection aus dem Connectionpool zu selektieren.

Ein RA muss die Methoden `equals` und `hashCode` implementieren. Die Gleichheit muss auf alle Eigenschaften der `ConnectionRequestInfo` bezogenen Parameter validiert werden. Ein Applicationserver verwendet diese Methoden um den Pool der Verbindungen zu verwalten.

4.2.9.1 Die JcaCorsoConnectionRequestInfo Klasse

```
public class JcaCorsoConnectionRequestInfo implements ConnectionRequestInfo {

	private String userName = null;
	private String passWord = null;
	private CorsoStrategy corsoStrategy = null;
	private int aid = 0;
	private String serviceName = null;
	private String corsoSite = null;
	private String domain = null;
	private int port = 0;

	public JcaCorsoConnectionRequestInfo() {
		userName = "";
		passWord = "";

		try {
			corsoStrategy = new CorsoStrategy(CorsoStrategy.PR_DEEP_EAGER);
		}
		catch (CorsoException e) {
			// ignore, Exception is never thrown!
		}

		aid = 0;
		serviceName = "";
		corsoSite = "localhost";
		domain = "";
		port = 5006;
	}

	public JcaCorsoConnectionRequestInfo(String userName, String passWord) {
		this();

		this.userName = userName;
		this.passWord = passWord;
	}

	public JcaCorsoConnectionRequestInfo(
		String userName,
		String passWord,
		CorsoStrategy corsoStrategy,
		int aid,
		String serviceName,
		String corsoSite,
		String domain,
		int port) {

		this.userName = userName;
		this.passWord = passWord;
		this.corsoStrategy = corsoStrategy;
		this.aid = aid;
		this.serviceName = serviceName;
		this.corsoSite = corsoSite;
		this.domain = domain;
		this.port = port;
	}



	public boolean equals(java.lang.Object other) {

		if ((other != null)
			|| (other instanceof JcaCorsoConnectionRequestInfo)) {
			JcaCorsoConnectionRequestInfo ri =
				(JcaCorsoConnectionRequestInfo) other;

			return (
				(ri.getAid() == getAid())
					&& (ri.getCorsoSite().equals(this.getCorsoSite()))
					&& (ri.getDomain().equals(this.getDomain()))
					&& (ri.getPassWord().equals(this.getPassWord()))
					&& (ri.getPort() == this.getPort())
					&&
				(ri.getServiceName().equals(this.getServiceName()))
					&& ri.getUserName().equals(this.getUserName()));
		}
```

```
                else {
                        return false;
                }
        }

        public int hashCode() {
                return this.getUserName().hashCode() + this.getCorsoSite().hashCode();
        }

        getter/setter Methoden …
}
```

4.2.10 ConnectionManager

Das `javax.resource.spi.ConnectionManager` Interface stellt ein Bindeglied für einen RA dar, einen Connection Request an den Applicationserver zu übergeben.

```
public interface javax.resource.spi.ConnectionManager extends java.io.Serializable {
        public Object allocateConnection(ManagedConnectionFactory mcf,
                ConnectionRequestInfo cxRequestInfo) throws ResourceException;
}
```

Die Methode `allocateConnection` wird von der RA `ConnectionFactory .getConnection` Methode aufgerufen.

Vorraussetzungen

- Ein Applicationserver implementiert das `ConnectionManager` Interface um Connection Requests an den Applicationserver zu delegieren
- Ein RA muss eine „default" Implementierung des ConnectionManager implementieren. Dieser ConnectionManager kommt in einem non-managed Szenario des RAs zum Einsatz. (non-managed bedeutet ohne Verwendung eines J2EE Applicationservers)
- In einem Applicationserver managed Szenario darf der RA nicht eine Klasse des default ConnectionManager verwenden.

4.2.11 ManagedConnectionFactory

Eine `javax.resource.spi.ManagedConnectionFactory` Instanz ist sowohl eine Factory für `ConnectionFactory` als auch für `ManagedConnection` Instanzen. Weiters unterstützt das Interface Connectionpooling des Applicationservers indem es eine Methode zum Vergleichen von Verbindungs- Request und bestehenden Verbindungen verlangt.

```
public interface javax.resource.spi.ManagedConnectionFactory extends
java.io.Serializable
{
	public Object createConnectionFactory(ConnectionManager connectionManager)
		throws ResourceException;

	public Object createConnectionFactory() throws ResourceException;

	public ManagedConnection createManagedConnection(javax.security.auth.Subject
		subject, ConnectionRequestInfo cxRequestInfo) throws ResourceException;

	public ManagedConnection matchManagedConnections(java.util.Set connectionSet,
		javax.security.auth.Subject subject,
		ConnectionRequestInfo cxRequestInfo) throws ResourceException;

	public boolean equals(Object other);
	public int hashCode();
}
```

Die Methode `createConnectionFactory` erzeugt eine Instanz einer `ConnectionFactory`, die mit dem `ConnectionManager` des Applicationservers und der Instanz der `ManagedConnectionFactory` (this) assoziiert wird.

Die Methode `createManagedConnection` erzeugt eine physische Instanz einer Verbindung zum EIS (`corso.lang.CorsoConnection`), die durch eine `ManagedConnection` gekapselt ist. Die Methode verwendet die sicherheits- bezogenen Informationen aus der `Subject` Instanz und optionale Request Informationen aus der `ConnectionRequestInfo` Instanz um die Verbindung zum Eis herzustellen.

Eine erzeugte `ManagedConnection` Instanz speichert typischerweise die sicherheits- und verbindungsbezogenen Informationen um ein Connectionpooling des Applicationservers zu ermöglichen. (Konstruktor `JcaCorsoManagedConnection`)

Der `matchManagedConnections` Methode werden als Parameter ein Set von Verbindungskandidaten, eine `Subject` und eine `ConnectionRequestInfo` Instanz übergeben, aus der die Methode den am besten geeignetsten Kandidaten auswählt. Falls kein geeigneter Kandidat gefunden werden kann muss `null` zurück gegeben werden.

4.2.11.1 Die JcaCorsoManagedConnectionFactory.createConnectionFactory Methode

```
public Object createConnectionFactory(ConnectionManager connectionManager)
		throws ResourceException {

		log.debug(
			"createConnectionFactory(ConnectionManager connectionManager)
			 was called.");

		return new JcaCorsoConnectionFactory(this, connectionManager);
}
```

4.2.11.2 Die JcaCorsoManagedConnectionFactory.createManagedConnection Methode

```
public ManagedConnection createManagedConnection(
		Subject subject,
		ConnectionRequestInfo connectionRequestInfo)
		throws ResourceException {

		log.debug("createManagedConnection was called.");

		JcaCorsoConnectionRequestInfo reqInfo =
			(JcaCorsoConnectionRequestInfo) connectionRequestInfo;

		if (reqInfo != null) {
```

```
                reqInfo.setCorsoStrategy(
                        defaultConnectionRequestInfo.getCorsoStrategy());
                reqInfo.setAid(defaultConnectionRequestInfo.getAid());
                reqInfo.setServiceName(
                        defaultConnectionRequestInfo.getServiceName());
                reqInfo.setCorsoSite(
                        defaultConnectionRequestInfo.getCorsoSite());
                reqInfo.setDomain(defaultConnectionRequestInfo.getDomain());
                reqInfo.setPort(defaultConnectionRequestInfo.getPort());
        }
        else {
                reqInfo =
                        new
                        JcaCorsoConnectionRequestInfo(
                                defaultConnectionRequestInfo);
        }

        CorsoConnection corsoConnection = new CorsoConnection();

        log.debug(
                "trying to connect to Corso Kernel with: "
                        + this.concatenateConnectionRequestInfo(reqInfo));

        try {
                corsoConnection.connect(
                        reqInfo.getUserName(),
                        reqInfo.getPassWord(),
                        reqInfo.getCorsoStrategy(),
                        reqInfo.getAid(),
                        reqInfo.getServiceName(),
                        reqInfo.getCorsoSite(),
                        reqInfo.getDomain(),
                        reqInfo.getPort());

                if (corsoConnection.isConnected()) {

                        log.debug("connected to Corso Kernel.");

                        return new JcaCorsoManagedConnection(
                                corsoConnection,
                                reqInfo,
                                this);
                }
                else {
                        throw new ResourceException("Could not connect to Corso.
                        (corsoConnection.isConnected() returned false)");
                }

        }
        catch (CorsoException e) {

                e.printStackTrace();
                throw new ResourceException(e.toString());
        }
}
```

Falls die übergebene Request Informationen `null` sind, d.h. in der `getConnection` Methode keine `connectionSpec` Instanz übergeben wurden, wird eine Verbindung zum Corso Kernel mit default Request Informationen aufgebaut (s.o. `JcaCorsoConnectionRequestInfo`). Andernfalls werden, die Parameter ausgenommen Benutzername und Kennwort mit den default Parametern gesetzt. Rückgabe ist eine neue Instanz einer ManagedConnection, die mit der erzeugten Corso Verbindung, den Requestparametern und der Factory (this) assoziiert ist.

Voraussetzungen

- Eine ManagedConnectionFactory muss die Methoden `hashCode` und `equals` implementieren
- Eine Implementierung muss eine Java Bean sein

4.2.11.3 Die JcaCorsoManagedConnectionFactory.matchManagedConnection Methode

```
public ManagedConnection matchManagedConnections(
        Set connectionSet,
        Subject subject,
        ConnectionRequestInfo connectionRequestInfo)
        throws ResourceException {

        log.debug("matchManagedConnections was called.");

        JcaCorsoConnectionRequestInfo requestInfo =
                (JcaCorsoConnectionRequestInfo) connectionRequestInfo;

        Iterator iterator = connectionSet.iterator();


        while ((iterator.hasNext()) && (requestInfo != null)) {
                Object element = iterator.next();

                if (element instanceof JcaCorsoManagedConnection) {

                        JcaCorsoManagedConnection mc =
                                (JcaCorsoManagedConnection) element;


                        if ((mc
                                .getJcaCorsoConnectionRequestInfo()
                                .getUserName()
                                .equals(requestInfo.getUserName()))
                                && (mc
                                        .getJcaCorsoConnectionRequestInfo()
                                        .getPassWord()
                                        .equals(requestInfo.getPassWord()))) {

                                log.debug(mc + " matched!");

                                return mc;
                        }

                }

        }

        log.debug("no ManagedConnection candidate matched -> returning null");
        return null;
}
```

Der Methode wird ein Set von Connections übergeben, aus der eine passende Instanz der ManagedConnection anhand der Requestparameter selektiert werden soll. Im diesem Fall werden nur Benutzername und Kennwort für den Vergleich herangezogen. Die restlichen Corso Connection Parameter (Corsosite, Port...) sind konfigurationsspezifisch für den RA (siehe `JcaCorsoConnectionSpec` Interface.

4.2.12 ManagedConnection

Eine ManagedConnection Instanz entspricht eine physischen Verbindung zu einem EIS.

```
public interface javax.resource.spi.ManagedConnection {

public Object getConnection(javax.security.auth.Subject subject, ConnectionRequestInfo
cxRequestInfo) throws ResourceException;

        public void destroy() throws ResourceException;
        public void cleanup() throws ResourceException;

        // Methoden für Connection and transaction event Notifikationen

        public void addConnectionEventListener(ConnectionEventListener listener);

        public void removeConnectionEventListener(
                ConnectionEventListener listener);

        public ManagedConnectionMetaData getMetaData() throws
                ResourceException;

// weitere Methoden
}
```

Die `getConnection` Methode erzeugt einen neuen Connection Handle, der an eine ManagedConnection Instanz gebunden ist. Der Connection Handle ist vom Typ `javax.resource.cci.Connection`.

Die `getConnection` Methode ermöglicht auch eine Re-Authentifikation auf einer bestehenden physikalischen Verbindung (Argumente `Subject` und `ConnectionRequestInfo`). Ein RA kann neuerliche Authentifikation unterstützen, wenn der Resource Manger des EIS's eine neuerliche Authentifikation unterstützt.

Die Methode `addConnectionEventListener` ermöglicht es, einen `EventListener` an einer `MangedConnection` Instanz zu registrieren. Die `ManagedConnection` Instanz notifiziert seine registrierten Listener bezüglich `close`, `error` und transaktionsrelevanten Ereignissen.

4.2.12.1 Die JcaCorsoManagedConnection.getConnection Methode

```
public Object getConnection(Subject arg0, ConnectionRequestInfo requestInfo)
                throws ResourceException {

                if (requestInfo == null) {
                        return this.createManagedConnectionHandle();
                }
                else {
                        JcaCorsoConnectionRequestInfo reqInfo =
                                (JcaCorsoConnectionRequestInfo) requestInfo;

                        if ((jcaCorsoConnectionRequestInfo
                                .getUserName()
                                .equals(reqInfo.getUserName()))
                                && (jcaCorsoConnectionRequestInfo
                                        .getPassWord()
                                        .equals(reqInfo.getPassWord()))) {

                                return this.createManagedConnectionHandle();

                        }
                        else {
                                throw new SecurityException("Reauthenication not
                                                                supported!");
```

```
                        }
                }
        }

private Connection createManagedConnectionHandle() {

                JcaCorsoConnection con = new JcaCorsoConnection(this);

                this.addConnectionHandle(con);
                return con;
}
```

4.2.12.2 Connectionsharing und mehrfache Connection Handles

Connectionsharing resultiert typischerweise in effizienter Verwendung von Ressourcen und besserer Performanz. Es entsteht durch mehrfachen Aufruf der `getConnection` Methode durch den Applicationserver an einer `ManagedConnection` Instanz.

Eine RA Implementierung kann

- eine thread- sichere Semantik für eine ManagedConnection Implementierung ausarbeiten (bevorzugt)
- sicherstellen, dass nur genau ein Connection Handle einer ManagedConnection Instanz zugeordnet ist. Der aktive Handle zur Connection ist die einzige Connection, die der ManagedConnection Instanz bis zum Zeitpunkt des application-level `close` Aufrufes zugeordnet ist

Eine Application kann die Fähigkeit, Verbindungen zu „sharen“ im Deploymentdescriptor mit den Schlüsselwörtern „shareable“ bzw. „unshareable“ im Descriptor Element <res-sharing-scope> angeben. Default ist „shareable“.

Ein RA muss Verletzungen der Bedingungen zum „Sharen“ einer Verbindung in folgenden Fällen durch eine SharingViolationException melden, wenn

- Operationen wie das Verändern der Verbindungsparameter, Sicherheitsparameter, Isolationlevelparameter auf `ManagedConnecton` Instanzen ausgeführt werden, die
    - mehr als einen Connection Handle assoziiert haben
    - mit einer lokalen oder einer XA Transaktion assoziiert sind

4.2.12.3 Connectionsharing im Szenario lokaler Transaktionen

Folgendes Szenario beschreibt den Mechanismus des Connectionsharing im Context einer lokalen Transaktion, die 2 EJB Komponenten Aufrufe überspannt. 2 EJB Komponenten (Stateful Session Beans) erhalten in ihren Businessmethoden Verbindungen zum selben EIS, deren Transaktions Attribut auf „Required“ gesetzt ist.

- Ein Client ruft eine Methode an der Session Bean A auf. Session Bean A erhält einen Connection Handle zum EIS mittel der `ConnectionFactory.getConnection` Methode.
- Der Container beginnt eine lokale Transaktion in dem er an der `ManagedConnection` die Methode `getLocalTransaction` aufruft und die Transaktion mit `LocalTransaction.begin()` eröffnet.
- Session Bean A ruft nun eine Methode an Session Bean B auf, die ebenfalls eine Verbindung zum EIS mit `ConnectionFactory.getConnection` herstellt. Der Application Server erkennt den transaktionalen Kontext beider Session Beans und liefert Session Bean B einen Connection Handle zur selben `ManagedConnection` wie Session Bean A
- Nach Beenden der Business Methoden beider Beans ruft der Container an der `LokalTransaction` Instanz die Methode `commit()` auf um die Transaktion zu beenden. Beide Methoden Aufrufe wurden im Kontext einer lokalen Transaktion an der selben physikalischen Verbindung ausgeführt.
- Nach Beendigung der lokalen Transaktion räumt der Container die `ManagedConnection` auf in dem er die Methode `cleanup()` aufruft.
- Der Container fügt die `ManagedConnection` zurück in den Pool.

```
public void cleanup() throws ResourceException {

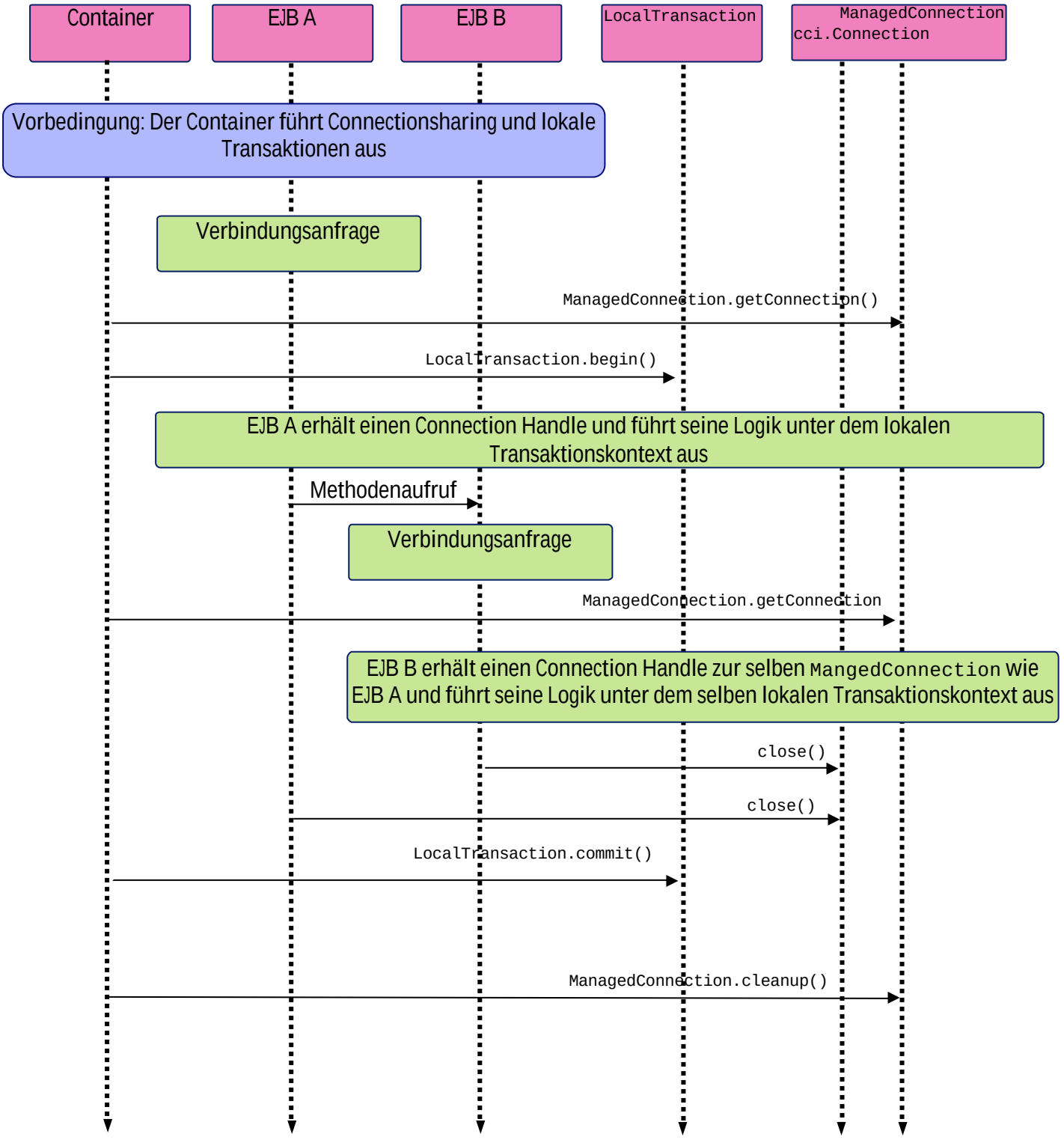
                Iterator it = connectionHandleVector.iterator();

                while (it.hasNext()) {
                        JcaCorsoConnection con = (JcaCorsoConnection) it.next();
                        con.setHandleInvalid();
                        log.debug("setting connection handle [" + con + "] invalid.");
                }

                connectionHandleVector.removeAllElements();

                jcaCorsoContainerManagedLocalTransaction = null;
}
```

Abbildung: Connectionsharing im Kontext lokaler Transaktion



4.2.12.4 Connection Association

In der Regel ist ein Connection Handle fix einer `ManagedConnection` zugeordnet. Es gibt jedoch Szenarien in denen eine Assoziation eines Handles zu einer anderen `ManagedConnection` stattfindet. Folgendes Beispiel beschreibt diese Situation

Eine Session Bean A ist Client einer Entity Bean C. Eine weitere Session Bean B ist Client der selben Entity Bean C. Sowohl A und C als auch B und C definieren einen Connection Sharing Kontext wie auch einen transaktionalen Kontext (AC, BC)

Entity Bean C erhält im Zuge ihrer Erzeugung einen Connection Handle zum EIS, den sie für ihre Lebensdauer behält. Es erfolgt ein Methodenaufruf in C durch A und ein Methodenaufruf von C durch B. In beiden Fällen wird der Connection Handle von C der `ManagedConnection` der aufrufenden Session Bean zugeordnet.

Die Methode `associateConnection()` soll den Handle, der als Parameter übergeben wird von seiner assoziierten ManagedConnection lösen und mit der Instanz der ManagedConnection (sich selbst) assoziieren.

```
public void associateConnection(Object connection)
              throws ResourceException {

              JcaCorsoConnection jcaCorsoConnection = null;


              if (connection instanceof JcaCorsoConnection) {
                     jcaCorsoConnection = (JcaCorsoConnection) connection;

                     jcaCorsoConnection
                            .getJcaCorsoManagedConnection()
                            .removeConnectionHandle(
                            jcaCorsoConnection);

                     jcaCorsoConnection.setJcaCorsoManagedConnection(this);
                     jcaCorsoConnection.setHandleValid();
                     this.addConnectionHandle(jcaCorsoConnection);

              }
              else {
                     log.debug("connection is not instanceof JcaCorsoConnection");
              }

}
```

4.2.12.5 Connection Event Notification und Connection Close

An jeder `ManagedConnection` Instanz registriert der Container einen Eventlistener um, close-, error- und Transaktionsevents empfangen zu können. Folgende Schritte werden bei einem Connection Handle Close ausgeführt.

- Bei einem `close()` Aufruf an einer Instanz von `javax.resource.cci.Connection` wird das Schließen des Handles an die assoziierte `ManagedConnection` delegiert.

- Die ManagedConnection notifiziert alle registrierten Listener bezüglich des Schließen des Handles mittels der `ConnectionEventListener.connectionClosed()` Methode. Eine Instanz eines

`ConnectionEvent` mit dem Typ CONNECTION_CLOSED wird als Parameter mit übergeben.

- Der Applicationserver führt ein transaktionsbezogenes Aufräumen der ManagedConnection Instanz aus.

- Der Applicationserver ruft abhängig vom Connectionsharing bzw. von der Existenz weiterer Connectionhandles die Methode `cleanup()` auf. Das Cleanup bereitet eine ManagedConnection für die Rückführung in den Connectionpool vor.

- Die ManagedConnection wird dem Pool rückgeführt.

```
public void close(JcaCorsoConnection jcaCorsoConnection)
                throws ResourceException {

                this.sendConnectionClosedEvent(jcaCorsoConnection);

                removeConnectionHandle(jcaCorsoConnection);

}

public void sendConnectionClosedEvent(JcaCorsoConnection jcaCorsoConnection) {

                ConnectionEvent conEvent =
                        new ConnectionEvent(this, ConnectionEvent.CONNECTION_CLOSED);


                conEvent.setConnectionHandle(jcaCorsoConnection);

                Iterator it = connectionEventListenerVector.iterator();

                while (it.hasNext()) {

                        Object element = it.next();

                        if (element instanceof ConnectionEventListener) {

                                ConnectionEventListener cEL = (ConnectionEventListener)
                                        element;

                                cEL.connectionClosed(conEvent);
                        }
                }

}
```

Dem Event wird die Instanz des Connection Handles, an dem der `close()` Aufruf erfolgte hinzugefügt.

4.2.12.6 Aufräumen von Managed Connections

Die Methode `ManagedConnection.cleanup()` leitet das Aufräumen einer `ManagedConnection` ein. Sämtliche Connectionhandles sollen invalidiert werden und Methodenaufrufe an invalidierten Handles in einer `Exception` resultieren. Der Applicationserver behält ein Wissen über erzeugte Connectionhandles, ruft nach einem transaktionalem Ende im Falle von Connection Sharing die Methode auf, um alle Connectionhandles zu invalidieren und die `ManagedConnection` wieder in den Pool zu geben. Der `cleanup()` Aufruf kann beispielsweise auch nach dem Schließen des letzten Connectionhandles erfolgen.

Der Aufruf von `cleanup()` einer aufgeräumten Verbindung sollte keine `Exception` werfen. Das `cleanup` darf die physische Verbindung nicht schließen. Ein Applicationserver ruft

`destroy()` an der `ManagedConnection` auf, um eine physikalische Verbindung zu zerstören und somit die Größe des Pools zu verwalten.

4.3 Transaction Management

Ein Resource Manager (in unserem Fall: Corso) kann in der Connector Architektur 2 Arten von Transaktionen unterstützen:

- Transaktionen, die durch einen Transaktionsmanager gesteuert und kontrolliert werden, der nicht Teil des Resource Manager (Corso) ist. In diesem Zusammenhang werden diese Transaktionen als JTA bzw. XA Transaktionen bezeichnet.

- Transaktionen, die intern durch den Resource Manager verwaltet werden. Diese Art der Transaktion wird als lokale Transaktion bezeichnet

Ein Transaktionsmanager koordiniert Transaktionen über verteilte Resource Manager. Die Connector Architektur definiert folgende Transaktion's Kontrakte zwischen einem Applicationserver, einem RA und Resource Manager:

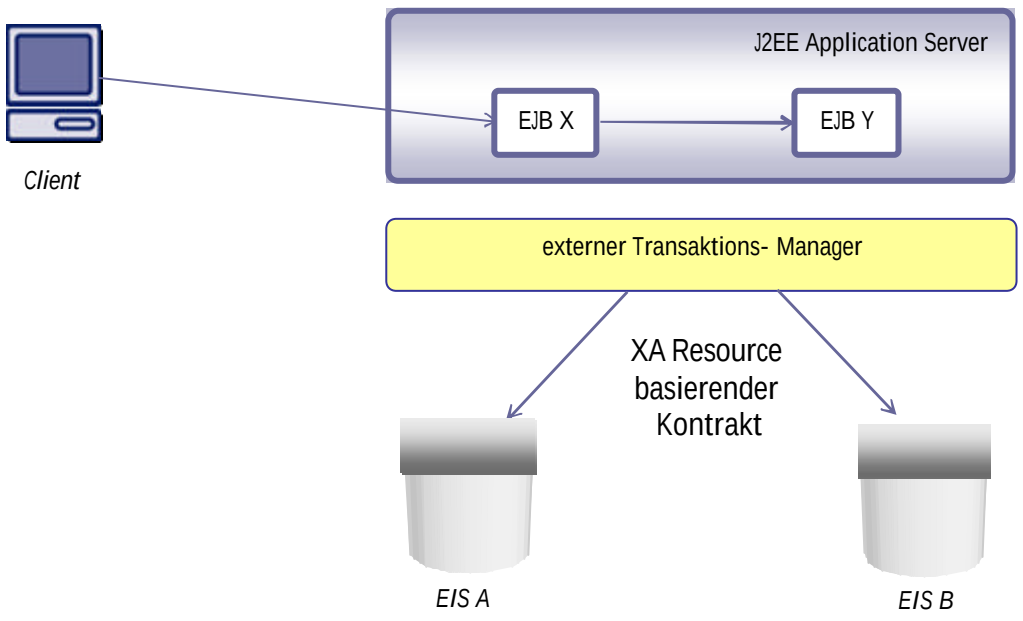
- einen JTA `javax.transaction.xa.XAResource` basierenden Kontrakt zwischen einem Resource Manager und Transaction Manager

- einen lokalen Transaktion's Management Kontrakt

4.3.1 Transaktions Management Szenarien

4.3.1.1 Transaktionen verteilt über mehrere Resource Manager

In folgender Abbildung ruft ein Client eine EJB auf, die auf das EIS-A zugreift. EJB X ruft wiederum EJB Y auf, die auf EIS-B zugreift. Der Applicationserver verwendet einen externen Transaktions- Manager, um transaktionalen Zugriff auf verteilte EIS Resource Manager zu gewährleisten. Sowohl EIS-A als auch EIS-B implementieren in ihren Resource Adaptern das XAResource Interface, um JTA Transaktionen zu unterstützen.

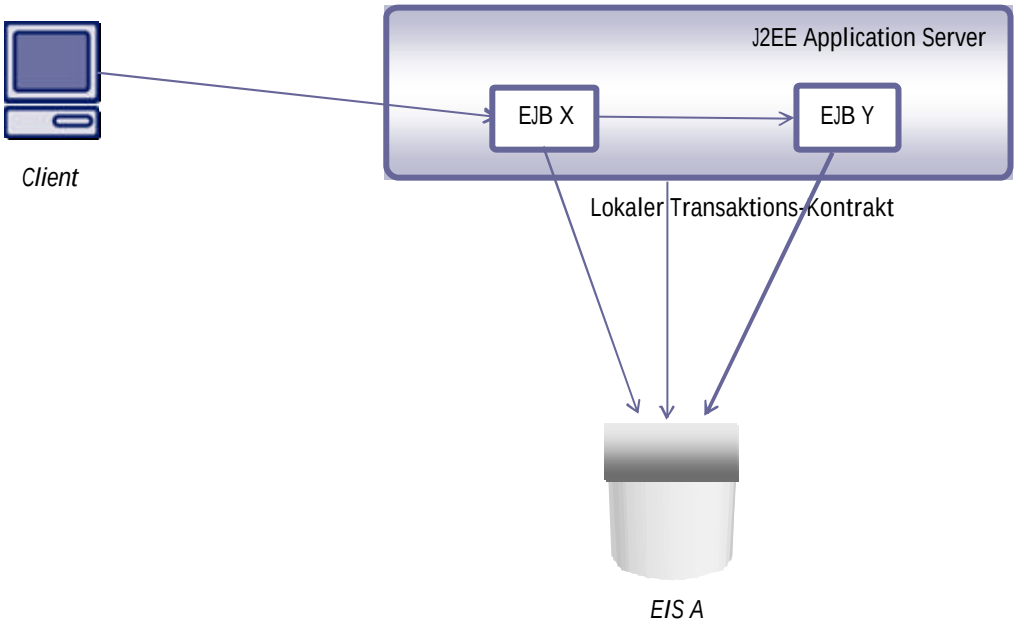


4.3.1.2 Lokales Transaktions- Management

Lokale Transaktionen sind entweder durch den Container oder durch EJB Komponenten gesteuert. Im Komponenten Falle, kann der Programmierer entweder das JTA `UserTransaction` Interface oder die Resource Manager API verwenden. (z.B.: JDBC Transaktionen bezüglich `java.sql.Connection` oder `JcaCorsoLocalTransaction` und `CorsoToptransaction` zu `JcaCorsoConnection`)

Falls ein einzelner Resource Manager (Corso) an einer Transaktion teilnimmt, gibt es 2 Möglichkeiten für den Container

- Verwenden eines Transaktionsmanager zusammen mit „One-phase-Commit“ Optimierung
- Der Container überlässt die Kontrolle der Transaktion dem Resource Manager



4.3.2 Der LocalTransaction Management Kontrakt

Der lokale Transaktions- Kontrakt besteht aus 2 Teilen

- Das `javax.resource.spi.LocalTransaction` Interface für container managed local Transactions, in dieser Arbeit implementiert durch die Klasse `JcaCorsoContainerManagedLocalTransaction`
- Das `javax.resource.cci.LocalTransaction` Interface falls der Resource Manager eine Transaktions- API anbietet, implementiert durch die Klasse `JcaCorsoLocalTransaction`

4.3.3 Interfaces und Klassen javax.resource.spi.LocalTransaction

4.3.3.1 Transactions- Interface javax.resource.spi.LocalTransaction

Ein RA implementiert das LocalTransaction Interface, um vom Resource Manager kontrollierte Transaktionen zu unterstützen

```
public interface javax.resource.spi.LocalTransaction {
	public void begin() throws ResourceException;
	public void commit() throws ResourceException;
	public void rollback() throws ResourceException;
}
```

4.3.3.2 Die JcaCorsoContainerManagedLocalTransaction Klasse

```
public class JcaCorsoContainerManagedLocalTransaction
	implements javax.resource.spi.LocalTransaction {

	private CorsoTopTransaction corsoTopTransaction = null;
	private JcaCorsoManagedConnection jcaCorsoManagedConnection = null;

	public JcaCorsoContainerManagedLocalTransaction(JcaCorsoManagedConnection con)
		throws ResourceException {
		jcaCorsoManagedConnection = con;
	}

	public void begin() throws ResourceException {
		try {
			corsoTopTransaction =
				jcaCorsoManagedConnection.corsoConnection.
					createTopTransaction();
		}
		catch (Exception e) {
			e.printStackTrace();
			throw new ResourceException(e.toString());
		}
	}

	public void commit() throws ResourceException {
		try {
			corsoTopTransaction.commit(CorsoConnection.INFINITE_TIMEOUT);
		}
		catch (CorsoTimeoutException e) {
			throw new ResourceException(e.toString());
		}
		catch (CorsoTransactionException e) {
			throw new ResourceException(e.toString());
		}
		catch (CorsoException e) {
			throw new ResourceException(e.toString());
		}
	}

	public void rollback() throws ResourceException {
		try {
			corsoTopTransaction.abort();
		}
		catch (CorsoTransactionException e) {
			throw new ResourceException(e.toString());
		}
		catch (CorsoException e) {

			throw new ResourceException(e.toString());
		}
	}

	protected CorsoTopTransaction getCorsoTopTransaction() {
		return corsoTopTransaction;
	}

}
```

4.3.3.3 Transaktions- Interface der ManagedConnection

```
public interface javax.resource.spi.ManagedConnection {
	public XAResource getXAResource() throws ResourceException;
	public javax.resource.spi.LocalTransaction getLocalTransaction() throws
		ResourceException;
...
}
```

Der Container ruft an der `ManagedConnection` Instanz `getLocalTransaction` auf, um eine Instanz einer vom Resource Manager kontrollierten Transaktion zu erhalten.

4.3.3.4 Die JcaManagedConnection.getLocalTransaction Methode

```
public javax.resource.spi.LocalTransaction getLocalTransaction()
	throws ResourceException {

		if (jcaCorsoContainerManagedLocalTransaction == null) {
			jcaCorsoContainerManagedLocalTransaction =
				new JcaCorsoContainerManagedLocalTransaction(this);

		}

		return jcaCorsoContainerManagedLocalTransaction;
}
```

4.3.4 Interfaces und Klassen der javax.resource.cci.LocalTransaction

4.3.4.1 Transactions- Interface javax.resource.cci.LocalTransaction

Das LocalTransaction Interface ist identisch mit dem der `spi.LocalTransaction`. Die Implementierung ist im wesentlichen auch identisch mit der `JcaCorsoContainerManagedLocalTransaction` Klasse. Der Unterschied besteht in der Event Notifizierung an den Container für jede Methode, die der Benutzer am Transaktions-Objekt aufruft.

4.3.4.2 Die JcaCorsoLocalTransaction Klasse

```
public class JcaCorsoLocalTransaction implements javax.resource.cci.LocalTransaction {

	private CorsoTopTransaction corsoTopTransaction = null;
	private JcaCorsoManagedConnection jcaCorsoManagedConnection = null;

	protected JcaCorsoLocalTransaction(JcaCorsoManagedConnection
		jcaCorsoManagedConnection)
		throws ResourceException {

		this.jcaCorsoManagedConnection = jcaCorsoManagedConnection;
	}

	public void begin() throws ResourceException {
		try {
			corsoTopTransaction =

	jcaCorsoManagedConnection.corsoConnection.createTopTransaction();

			jcaCorsoManagedConnection.sendLocalTransactionStartedEvent();
		}
		catch (Exception e) {
			e.printStackTrace();
			throw new ResourceException(e.toString());
		}
	}
```

Folgender Codeausschnitt zeigt die Notifizierung über den Begin einer lokalen Transaktion aller an der ManagedConnection registrierten Listener.

```
public void sendLocalTransactionStartedEvent() {

              ConnectionEvent conEvent =
                     new ConnectionEvent(this,
                            ConnectionEvent.LOCAL_TRANSACTION_STARTED);

              Iterator it = connectionEventListenerVector.iterator();

              while (it.hasNext()) {

                     Object element = it.next();

                     if (element instanceof ConnectionEventListener) {

                            ConnectionEventListener cEL = (ConnectionEventListener)
                                                          element;
                            cEL.localTransactionStarted(conEvent);
                     }
              }
}
```

5 „Proof of Concept“ anhand eines Beispiels

5.1 Übersicht

Dieses Kapitel beschreibt die Migration einer bestehenden Nativ-Corso Applikation zur einer kompletten J2EE Applikation unter Verwendung der in den Kapiteln 3 und 4 vorgestellten Technologien. Die existierende Nativ-Corso Applikation modelliert mit Corso und der Anbindungssprache an Corso „Java&Co“ einen Flughafen, der Landebahnen und Flugzeuge verwaltet, die auf diesen Landebahnen konkurrierend starten und landen. Es wird auf die Unterschiede der Nativ Version zur j2EE Version eingegangen. Die J2EE Implementierung behält den Präsentation Layer (GUI) der Nativ-Corso Applikation bei und kapselt die Funktionalität in Business Methoden von Enterprise Java Beans.

5.2 Beschreibung der Corso Flughafen Applikation

5.2.1 Aufgabenstellung

Ein Flughafen soll mit Corso Prozessen und Datenstrukturen modelliert werden. Ein Flughafen ist durch seinen Flughafennamen eindeutig identifiziert und kann je nach Wetterbedingungen für Flugzeuge zur Landung und zum Starten geöffnet bzw. gesperrt werden. Dem Flughafen sind eine frei wählbare Anzahl von Landebahnen zugeordnet, auf denen Flugzeuge starten und landen. Befindet sich ein Flugzeug im Landeanflug oder im Begriff zu starten, ist die Landebahn für dieses Flugzeug exklusiv reserviert. Nach Beenden des Start/Landevorganges ist die Bahn erneut für andere Flugzeuge nutzbar. Ein Flughafen kann zu jedem Zeitpunkt geöffnet bzw. gesperrt werden. Eine Anzeigetafel informiert zu jedem Zeitpunkt über den Status des Flughafens, über Landebahnen und auf Landebahnen befindlichen Flugzeugen.

5.2.2 Prozesse der Flughafen Applikation

- Flughafen: Ein oder mehrere Flughäfen, eindeutig identifiziert durch einen Namen (Named Object) mit beliebiger Anzahl von Landebahnen.
- Flugzeug: Ein oder mehrer Flugzeuge, identifiziert durch eine FlugzeugID
- Anzeigetafel: Ein Board, das einem Flughafen zugeordnet ist und in near-time über den Status des Flughafens und der Landebahnen informiert

5.2.3 *Use Cases der Flughafen Applikation*

- Flughafen: Flughafen mit eindeutigem Namen anlegen, Flughafen öffnen und sperren
- Flugzeug: Flugzeug mit eindeutiger ID anlegen, starten und landen lassen
- Anzeigetafel: Information des Status des Flughafens, der Landebahnen und der Flugzeuge

5.2.4 *Interaktion der Prozesse*

- Transaktion des Flugzeuges: Ein Flugzeug, das landet oder startet führt folgende Aktionen im Kontext einer `CorsoToptransaction` aus: Es werden Landebahnen nach ihrem Status befragt und eine ausgewählt, die frei ist. Im Anschluss wird überprüft, ob der Flughafen geöffnet ist. Wenn ja, wird die Landebahn dem Flugzeug exklusiv zur Verfügung gestellt, die Transaktion „comitted".

- Flughafen: Ein neu angelegter Flughafen ist default gesperrt, kann geöffnet und gesperrt werden. Gerade in Landung bzw. Startvorgang befindliche Flugzeuge dürfen nach Sperren des Flughafens ihren Vorgang jedoch beenden

- Informationen über alle Landebahnen und den Flughafen in near-time ohne Polling. Angezeigt werden die Landebahnen, den Status der Bahn und die der Landebahn zugeordnete FlugzeugID, falls Landebahn nicht frei.

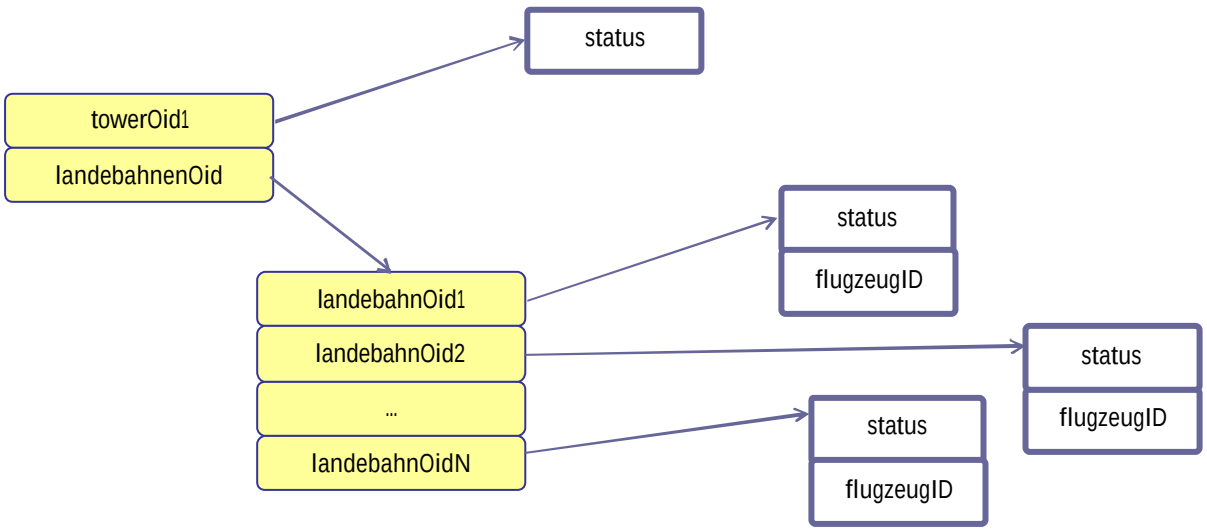
5.2.5 *Corso Datenstrukturen der Flughafen Applikation*

- FlughafenInfo Objekt: Implementiert Corso Shareable. Die Anzahl der Landebahnen, eine Referenz auf ein Landebahnen Objekt und der Status des Flughafens sind Members dieses Objektes

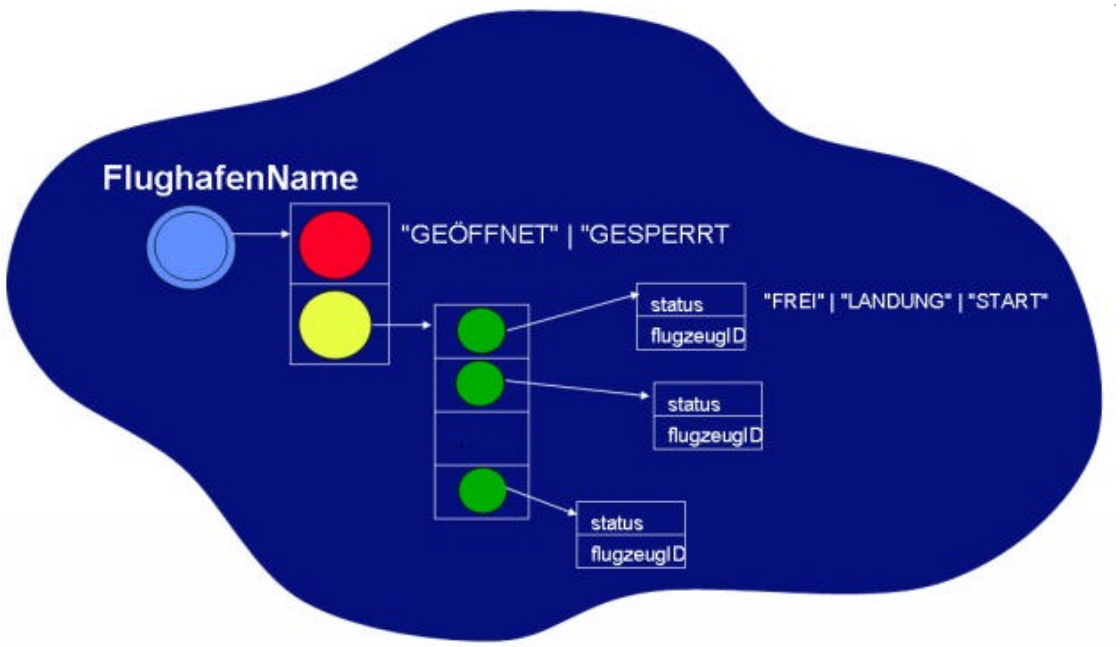
- Landebahnen: Implementiert Corso Shareable. Referenzen auf einzelne Landebahn Objekte werden als Liste (Vector) in diesem Objekt gespeichert

- Landebahn: Implementiert Corso Shareable. Der Status (Frei, Landung, Start) und die der Landebahn assoziierte FlugzeugID werden gespeichert.

Datenstruktur des FlughafenInfo, der Landebahn und der Landebahnen Objekte:



Alle Datenstrukturen im Space:



5.3 Codeauszüge aus der Nativ Java&Co Flughafen Applikation

5.3.1 Die Flughafen Klasse

Die Flughafen Klasse (Applikation) stellt eine Verbindung zum Corso Space her und erzeugt oder ließt die NamedVarOid mit dem Flughafennamen. Eine Instanz der FlughafenInfo wird über die NamedVarOid erzeugt und im Space gespeichert (writeShareable):

```
namedOid =
        con.createNamedVarOid(
                con.getCurrentStrategy(),
                flughafenName,
                null);

flughafenInfo = new FlughafenInfo(nLandebahnen, con);

namedOid.writeShareable(
        flughafenInfo,
        CorsoConnection.INFINITE_TIMEOUT);
```

Im Konstruktor der FlughafenInfo Klasse werden auch die Oid des Towers (speichert den Status des Flughafens) und die Oid, der Landebahnen erzeugt und im Space gespeichert:

```
towerOid = con.createVarOid(con.getCurrentStrategy());
landebahnenOid = con.createVarOid(con.getCurrentStrategy());

Landebahnen landebahnen = new Landebahnen(nLandebahnen, con);

try {
        landebahnenOid.writeShareable(
                landebahnen,
                CorsoConnection.INFINITE_TIMEOUT);
        towerOid.writeString(
                CourseUtil.GESPERRT,
                CorsoConnection.INFINITE_TIMEOUT);
```

Die gesamte Datenstruktur ist persistent (überlebt Systemabstürze) im Corso Space mit erstmaliger Instanzierung des FlughafenInfo Objektes aufgebaut und über den Flughafennamen (String) jederzeit erreichbar.

Nach Erzeugen/Lesen der Datenstrukturen kann der Status des Flughafens (TowerOid) über Buttons im GUI gesperrt bzw. geöffnet werden.

```
void oeffnenButton_actionPerformed(ActionEvent e) {
        System.out.println("OEFFNEN BUTTON PRESSED");
        status = CourseUtil.GEOEFFNET;
        flughafenInfo.setTowerStatus(status);
}

public void setTowerStatus(String status) {
        try {
                towerOid.writeString(status, CorsoConnection.INFINITE_TIMEOUT);
        ...
```

5.3.2 Die Flugzeug Klasse

Die Flugzeug Klasse (Applikation) ist ebenfalls mit einem GUI ausgestattet und operiert auf den Datenstrukturen im Space eines existierenden Flughafens. Ein Flugzeug ist in diesem Beispiel kein persistentes Objekt im Corso-Space, es ist nur für die Dauer eines Start/Landevorganges mit einer Landebahn assoziiert (String FlugzeugID zu Landebahn). Wie im Falle der Flughafen Applikation wird eine Corso Connection hergestellt, ein FlughafenInfo Objekt über den Flughafennamen und ein Landebahnen Objekt über die Landebahnen Oid instanziert.

Der Vorgang des Starten eines Flugzeuges geht wie folgt vor sich. Es wird über die Oids aller verfügbaren Landebahnen iteriert und der Status der Bahn abgefragt:

```
for (int i = 0; i < LandebahnOids.size(); i++) {
        oid = (CorsoVarOid) LandebahnOids.get(i);

        CorsoTopTransaction tx = con.createTopTransaction();
        landebahn = new Landebahn(oid, tx);

        if (!(landebahn.getStatus()).equals(CourseUtil.FREI)) {
        …
```

Falls die Landebahn frei ist wird geprüft, ob der Flughafen geöffnet bzw. gesperrt ist und im positiven Falle, die Landebahn mit der FlugzeugID beschrieben:

```
Landebahn lb = new Landebahn(CourseUtil.START, flugzeugID);
oid.writeShareable(lb, tx);
tx.commit(CorsoConnection.INFINITE_TIMEOUT);
```

Der Zustand des Flugzeuges wird auf „STARTEND“ gesetzt.

5.3.3 Die Anzeigetafel Klasse

Die Anzeigetafel zeigt den Status des Flughafens und der Landebahnen in near-time an. Über Notifikationen werden Änderungen am Flughafen (geöffnet/gesperrt) und den Landebahnen Oids registriert. Die Anzeigetafel pollt daher nicht periodisch die Belegungen der Flughafen Datenstruktur.

Folgender Code-Ausschnitt zeigt den Corso Notification Mechanismus:

Sowohl für die Tower OID als auch für jede Landebahn OID werden `NotificationItem` Instanzen instanziert und in einem Java Vector abgelegt.

```
Vector vect = new Vector();

CorsoNotificationItem towerItem =
                              new CorsoNotificationItem(
                                     flughafenInfo.getTowerOid(),
                                     0,
                                     // timestamp; INITIALIZE_WITH_CURRENT_TIME_STAMP
                                     CorsoNotificationItem.CURRENT_TIMESTAMP);
vect.add(towerItem);

for (int i = 0; i < nLandebahnen; i++) {

                              CorsoNotificationItem nItem =
                                     new CorsoNotificationItem(
                                            (CorsoVarOid) v.get(i),
                                            0,
                                            // timestamp
                                            CorsoNotificationItem.CURRENT_TIMESTAMP);
                                            vect.add(nItem);
}
```

Der Vector mit den NotificationItems wird der Methode createNotification als Parameter übergeben.

```
CorsoNotification notification =
               con.createNotification(vect,
                                       new CorsoStrategy(CourseUtil.STRATEGY));
```

In einer Endlosschleife wird auf das „Feuern“ der NotificationItems gewartet und die Änderung des Status des betreffenden Objektes auf der Anzeigetafel aktualisiert.

```
for (;;) {
               CorsoNotificationItem fired =
                   notification.start(CorsoConnection.INFINITE_TIMEOUT, null);

               if (fired != null) {
                      CorsoVarOid firedOid = fired.varOid();
                      if (firedOid.equals(flughafenInfo.getTowerOid())) {
                             towerTextField.setText("Flughafen "
                                                    + flughafenName
                                                    + " ist "
                                                    + flughafenInfo.getStatus());
...
```

5.4 Die J2EE Corso Flughafen Applikation

Dieser Abschnitt beschreibt die Migration der Nativ-Corso Java&Co Flughafen Applikation zur J2EE Version. Es werden die konzeptionellen Unterschiede beider Varianten aufgezeigt und anhand kurzer Codebeispiele detaillierter beschrieben.

EJB sind serverseitige Komponenten, die Businesslogik implementieren und sie in Form von remote aufrufbaren Methoden ausführbar machen. In der Flughafen Applikation lassen sich folgende Businessmethoden analog zu oben genannten Use-Cases angeben:

- Flughafen anlegen
- Flughafen öffnen
- Flughafen sperren
- Flugzeug landen
- Flugzeug starten
- Flugzeug fertig (Diese Methode signalisiert nur das Beenden eines Start/Landevorganges, um das Verstreichen einer Zeitspanne eines Start/Landevorganges auf einer Landebahn zu simulieren)

Die EJB Komponenten sind der Flughafen und das Flugzeug, die beide als stateful Session-Beans implementiert wurden. Stateful Session- Beans speichern den clientbezogenen Status im Zuge einer Passivierung (Auslagerung) ab, was z.B. hinsichtlich des Status eines Flugzeuges („AM_BODEN“ bzw. „STARTEND“ oder „LANDEND“) von Bedeutung ist. Auch in der Flughafenkomponente ist der clientbezogene Status durch den Flughafennamen und die Anzahl der Landebahnen relevant. Eine Implementierung mit stateless Session- Beans wäre im Falle des Flughafens möglich gewesen, da die Applikation das Öffnen eines bereits offenen Flughafens fehlerfrei gestattet. Jedoch müsste der Client in jedem Request den Flughafennamen zusätzlich zum Aktionsparameter mitgeben. Die Wahl fiel dennoch auf eine stateful Session - Bean, weil sich der Passivierungs-/Aktivierungs- Vorgang von Beans im Zusammenhang mit Corso als sehr elegant und einfach herausstellte.

5.4.1 Die Flughafen Komponente

Aus der bestehenden Nativ Java&Co Flughafen Applikation konnte die Businesslogik weitgehend übernommen und mit geringen Änderungen in eine EJB untergebracht werden. Übrig blieb der Programmcode des Präsentationlayer's (GUI), der nun Client der Flughafen Session- Bean ist und die Returnwerte der `setTowerStatus()` Methode in den Textboxen des GUIs darstellt. Implizit erfolgte in diesem Schritt eine Trennung des Präsentation- vom Businesslayer.

Die J2EE Flughafen EJB setzt sich aus folgenden Klassen/Interfaces zusammen

- Das Remote Interface: Flughafen.java
- Das Home Interface: FlughafenHome.java
- Die Stateful- Session Bean: FlughafenBean.java
- Der GUI Applikation: FlughafenApplication.java
- ejb-jar.xml und jboss.xml Deployment Descriptor Dateien

5.4.1.1 Das Flughafen Remote Interface

Das Remote Interface leitet von `EJBObject` ab und wird vom Container implementiert. Es spezifiziert die einzige Business Methode der Flughafen EJB zum Öffnen und Sperren des Flughafens

```
public interface Flughafen extends EJBObject {

	public void setTowerStatus(String status) throws RemoteException;

}
```

Die FlughafenApplication Applikation erhält eine Instanz der Flughafen EJB über die Lokalisierung des Home Objektes via JNDI und dem nachfolgenden `create` Aufruf an der Home Instanz. Bei Aufruf der `create` Methode wird die korrespondierende `ejbCreate` Methode an der Bean aufgerufen, die die Flughafen Datenstruktur im Corso-Space erzeugt oder einließt, falls schon existent (Instanzierung des FlughafenInfo Objektes).

```
try {
                        Context ctx = getInitialContext();
                        Object obj = ctx.lookup("FlughafenBean");

                        FlughafenHome flughafenHome =
                                (FlughafenHome) PortableRemoteObject.narrow(
                                        obj,
                                        FlughafenHome.class);

                        Flughafen flughafen =
                                flughafenHome.create(flughafenName, nLandebahnen);
}
```

5.4.1.2 Das Flughafen Home Interface

Das Home Interface sieht so dazu aus:

```
public interface FlughafenHome extends EJBHome {

	Flughafen create(String flughafenName, int nLandebahnen)
		throws CreateException, RemoteException;

}
```

5.4.1.3 Die Flughafen Stateful Session Bean

Die Stateful Session - Bean implementiert die Businesslogik zu den Methoden `setTowerStatus(String status)` und `ejbCreate(String flughafenName, int nLandebahnen)`

Die ejbCreate Methode erzeugt oder ließt die Flughafen Datenstruktur im Corso-Space auf folgende Weise.

- Es erfolgt eine Verbindungsanfrage zum Corso-Space über den im Kapitel 4 entwickelten Resource Adapter, der beim Starten des Applicationserver instanziert wird. Wie in Kapitel 4.2.6.1 beschrieben erhält man über einen JNDI Lookup und den Mechanismus der `ConnectionFactory` eine Instanz der `JcaCorsoConnectionFactory` an der die Methode `getConnection()` aufgerufen wird:

```
try {
                        Context ctx = new InitialContext();
                        Object obj =
                                ctx.lookup("java:comp/env/eis/CorsoConnectionFactory");

                        JcaCorsoConnectionFactory cf = (JcaCorsoConnectionFactory) obj;
                        JcaCorsoConnection con =
                                (JcaCorsoConnection) cf.getConnection();

}
```

- Es wird versucht eine NamedVarOid mit dem Flughafennamen aus dem Space zu lesen, falls diese nicht existiert (d.h. der Flughafen wurde noch nie angelegt) wird eine neue NamedVarOid im Space erzeugt:

```
try {
                        namedOid =
                                con.getNamedVarOid(
                                        this.flughafenName,
                                        CourseUtil.SITE,
                                        null,
                                        false,
                                        JcaCorsoConnection.NO_TIMEOUT);

                        flughafenInfo = new FlughafenInfo(namedOid);
}
catch (CorsoException e1) {

                        //-----------------------------
                        // nicht gefunden -> neu anlegen
                        //-----------------------------

                        try {
                                namedOid =
                                        con.createNamedVarOid(
                                                con.getCurrentStrategy(),
                                                this.flughafenName,
                                                null);

                                this.flughafenInfo = new
                                                FlughafenInfo(this.nLandebahnen, con);

                                namedOid.writeShareable(
                                        this.flughafenInfo,
                                        JcaCorsoConnection.INFINITE_TIMEOUT);
…
```

- In beiden Fällen wird eine Instanz der FlughafenInfo Klasse erzeugt. Die Instanzierung eines FlughafenInfo Objektes erzeugt/ließt in weiterer Folge die komplette Flughafen Datenstruktur im Space

Die `setTowerStatus` Methode delegiert das Setzen des Status (Öffnen/Sperren des Flughafen) einfach an die eben erzeugte `flughafenInfo` Instanz

```
public void setTowerStatus(String status) throws RemoteException {

                flughafenInfo.setTowerStatus(status);
}
```

Die `ejbActivate` Methode ist bis auf das Setzen der Membervariablen `flughafenName` und die Anzahl der Landebahnen (`nLandebahnen`)mit der `ejbCreate` Methode identisch. Der Typ des `flughafenName` (String) leitet von `java.lang.object` ab und die Anzahl der Landebahnen ist ein primitiver Datentyp. Somit sind beide Members Teil des Status der Session Bean und bei Aktivierung mit den korrekten Werten belegt. Die Methode ejbActivate sieht daher einfach wie folgt aus:

```
public void ejbActivate() throws EJBException, RemoteException {
                getOrCreateNamedVarOid();
}
```

Die Methode `getOrCreateNamedVarOid()`kapselt den oben beschriebenen Vorgang des Lesens/Erzeugens der `NameVarOid` und der Instanzierung des `flughafenInfo` Objektes. Der `catch` Block der Exception von `con.getNamedVarOid` wird in diesem Falle nie erreicht, da `ejbCreate` in jedem Fall vor `ejbActivate` aufgerufen wird und das Named Object bereits existiert.

Die `ejbPassivate` Methode setzt das `flughafenInfo` Objekt `null`, da es nicht Bestandteil eines Status einer Session Bean sein kann. Der Status des Flughafen ist im Corso-Space persistent gespeichert und über den Flughafennamen jederzeit lesbar.

5.4.1.4 Die FlughafenApplication Applikation

Diese Klasse ist in Ihrer Funktion auf das GUI, das Instanzieren der EJB und dem Aufrufen der Business Methode setTowerStatus reduziert. Das Lokalisieren und Instanzieren der EJB erfolgt wie bei 5.4.1.2 beschrieben.

Der Aufruf der Businessmethode der EJB erfolgt über Methoden, die durch Events vom GUI aufgerufen werden:

```
void oeffnenButton_actionPerformed(ActionEvent e) {

                status = CourseUtil.GEOEFFNET;
                try {
                        flughafen.setTowerStatus(status);
                }
                catch {
                ...
```

5.4.1.5 Die Deploymentdescriptor Dateien der Flughafen EJB

Die ejb-jar.xml Datei:

```
<session>
      <description>FlughafenBean</description>
      <ejb-name>FlughafenBean</ejb-name>

      <home>examples.flughafen.ejb.FlughafenHome</home>
      <remote>examples.flughafen.ejb.Flughafen</remote>
      <ejb-class>examples.flughafen.ejb.FlughafenBean</ejb-class>
      <session-type>Stateful</session-type>
      <transaction-type>Bean</transaction-type>

      <resource-ref>
         <res-ref-name>eis/CorsoConnectionFactory</res-ref-name>
         <res-type>at.tecco.corso.jca.JcaCorsoConnectionFactory</res-type>
         <res-auth>Container</res-auth>
      </resource-ref>

</session>
```

Die XML Elemente sind in Kapitel 3.3.4 beschrieben. Das `<resource-ref>` Element spezifiziert die Java Klasse der ConnectionFactory und den Referenznamen der im JNDI Lookup Verwendung findet:

```
ctx.lookup("java:comp/env/eis/CorsoConnectionFactory");
```

Die jboss.xml Datei löst den in der ejb-jar.xml auftretenden Referenznamen auf die im jBoss registrierte JNDI Lokation des Resource Adapters auf.

Die jboss.xml Datei:

```
<session>
         <ejb-name>FlughafenBean</ejb-name>
         <jndi-name>FlughafenBean</jndi-name>

         <resource-ref>
            <res-ref-name>eis/CorsoConnectionFactory</res-ref-name>
            <jndi-name>java:/eis/CORSO-RA</jndi-name>
         </resource-ref>

</session>
```

5.4.2 Die Flugzeug Komponente

Auch aus der bestehenden Nativ Java&Co Flugzeug Applikation konnte die Businesslogik weitgehend übernommen und mit geringen Änderungen in eine EJB untergebracht werden. Wie auch im Falle der Flughafen Komponente blieb der Programmcode des Präsentationlayer's (GUI) übrig, der nun Client der Flugzeug Session - Bean ist um die Returnwerte der Methoden `fertig`, `landen` und `starten` in den Textboxen des GUIs darzustellen.

Implizit erfolgte auch hier in diesem Schritt eine Trennung des Präsentation - vom Businesslayer.

Die Flugzeug EJB setzt sich aus folgenden Klassen/Interfaces zusammen

- Das Remote Interface: Flugzeug.java
- Das Home Interface: FlugzeugHome.java

- Die Stateful- Session Bean: FlugzeugBean.java
- Der GUI Applikation: FlugzeugApplication.java
- ejb-jar.xml und jboss.xml Deployment Descriptor Dateien

Die Technik zur Umsetzung in EJBs ist analog zur Flughafen Bean, sodass hier nicht näher auf Details eingegangen wird.

6 Bewertung

6.1 Skalierbarkeit

Der Corso JCA Ressource Adapter ist das Bindeglied der J2EE Serversoftware Komponenten Architektur zur Welt des „space based Computing“ mit Corso. Die Kommunikation von Prozessen (Applikationen) zum Corso- Space erfolgt jedoch immer über UDP/TCP Connections, sodass für das Ausführen von Operationen im Space eine geöffnete Verbindung zum CoKe (Corso Coordination Kernel) Vorraussetzung ist.

EJBs dürfen und können solch UDP/TCP Connections nach Belieben öffnen und schließen, jedoch können von EJB geöffnete Verbindungen zum CoKe nicht sinnvoll an andere Komponenten übergeben bzw. wiederverwendet werden. Eine Parameterübergabe einer Connection Instanz an eine andere Bean ist zwar denkbar, jedoch würde der nicht vorhersehbare Passivierungsvorgang der aufrufenden Bean die Verbindung der aufrufenden und somit auch die der aufgerufenden Bean schließen und den Methodenaufruf fehlerhaft machen.

Das Wiederverwenden von geöffneten TCP/IP Verbindungen (Connection- Pooling) ist ein wichtiges Kriterium zur Skalierbarkeit und Performance von Business Applikationen. Der in dieser Arbeit implementierte Resource Adapter implementiert die in JCA vorgesehenen Kontrakte zum Connectionpooling eines Applicationservers mit dem RA. Das Akquirieren von UDP/TCP Verbindungen erfolgt daher nicht im Programmcode der EJBs sondern im Connection Management des Resource Adapters, was die Wiederverwendung von Verbindungen zum CoKe über das Konzept der “ConnectionHandles“ möglich macht.

Der Resource Adapter stellt somit ein wichtiges Bindeglied für Verbindungen zum CoKe dar und ermöglicht das Connection- Pooling eines Applicationservers.

6.2 Verbinden heterogener Welten

Mit dem RA wird es möglich heterogene Welten wie z.B. J2EE und .NET zu verbinden. Die direkte Interaktion von J2EE und .NET Komponenten scheitert bis dato an unterschiedlichen Schichtenmodellen und Kommunikationsprotokollen beider Hersteller. Mit einem zusätzlichen Connector im Framework von .NET können die Welten von .NET und J2EE im Bereich der Applicationserver verbunden werden. Der Corso Space kann als zentraler, persistenter, fehlersicherer Datenhaushalt und als Synchronisationsmechanismus von serverseitigen Komponenten beider Welten gesehen werden, die Daten und Informationen über den Space austauschen.

6.3 Transaktionen

Business Prozesse laufen nahezu immer in einem transaktionalem Kontext ab und umfassen dabei in der Regel auch mehrere Methodenaufrufe. Im J2EE Framework können EJB Komponenten die Methoden anderer Komponenten im Kontext einer Transaktion aufrufen,

was durch eine Transaktionskontrolle des Applicationservers mit dem zugrundeliegenden Resource Manager oder einem externen Transaktionsmanager ermöglicht wird. Transaktionen sind in ihrer technischen Implementierung jedoch immer an eine zugrunde liegende Connection gebunden, sodass auch hier, das vom RA kontrollierte Connection-Management Vorraussetzung für den verteilten Aufruf von EJB Komponenten innerhalb einer Transaktion ist.

7 Weiterführende Arbeiten

7.1 Implementierung des XA Transaktions Management

Die JCA Spezifikation sieht im Transaction Management sowohl Kontrakte für lokale Transaktionen vor, die durch einen Resource Manager (in unserem Fall Corso) verwaltet werden, als auch Kontrakte des Resource Adapters für ein XA basierendes externes Transaktionsmanagement.

Mit Implementierung des `XAResource` Interfaces wäre Corso mittels des RAs in der Lage an Transaktionen teilzunehmen, die von einem externen Transaktionsmanager verwaltet werden. Corso unterstützt das 2 Phasen Commit Protokoll, somit sind die notwendigen Voraussetzungen für externes Management von Corsotransaktionen gegeben.

7.2 Implementierung der JAAS Spezifikation im RA

JAAS ist ein Java Package, das Authentifikation und Autorisation ohne detailliertes Wissen über die plattformspezifischen Implementierungen der genannten Dienste ermöglicht. Mit JAAS ist möglich, fast jedes Security System, sei es User/Passwort Listen, Directory Services oder Zertifikat basierende Authentifikation zu unterstützen.

Die vorliegende RA Version könnte um die Konzepte von JAAS erweitert werden, um die unterschiedlichen Security Systeme, die JAAS unterstützt, zur Authentifikation zum Corso und zur Authorisierung zu nützen.

Eine Erweiterung des RAs wäre auch, den Zugriff auf Corso Named Objects (`corso.lang.Connection.get/createNamedVarOid(strategy, authorizationkey`) automatisch mit den Eigenschaften des Users (Passwort/Credential) zu verknüpfen, um so einen Teilraum des Corso-Space zu erzeugen, der nur den authentifizierten und autorisierten Benutzer(n) zugänglich ist.

7.3 Weitere Möglichkeiten mit JCA V. 1.5

Mit der Version V.1.5 erhält ein RA weitere Funktionalität, die insbesondere für Corso von Interesse sind

- Eingehende Kommunikation: Corso wäre über den Ressource Adapter in der Lage beliebige EJBs (Session-, Enity-, Messagedriven- Beans) aufzurufen

- Work Management: Ein Kontrakt zwischen Application Server und Resource Adapter, der es dem Adapter ermöglicht „Work" Instanzen dem Application Server zu übergeben. Der Application Server ist für Threadding, Thread-Pooling und Security zuständig.
  Der RA könnte beispielsweise Corso System Services (Namingservice, Corsokernel Discovery Service, Monitoringservice) im Bereich des Applicationservers zur Ausführung bringen.

- Kontrakt für eingehende Nachrichten: Einem Adapter ist es möglich, beliebige Nachrichten asynchron an Nachrichtenendpunkte im Application Server zu senden

7.4 Ein Corso Load Balancer für Application Server

Ein applikatorischer Corso Loadbalancer ist denkbar, der Requests aus dem Space ließt, gegebenenfalls Security Policies auf sie anwendet und sie anschließend an einen Pool von Applicationserver weiterreicht. Die Verteilung der Last könnte über einen Pool von Requestobjekten realisiert werden, an dem eine beliebige Anzahl von Applicationserver konkurrierend Requests abarbeiten. Ein andere Ansatz wäre der direkte Aufruf von EJBs über den Inbound Communication Kontrakt der Version 1.5 von JCA .

Literaturverzeichnis

[1] E. Kühn: Virtual Shared Memory for Distributed Architecture. Nova Science Publishers 2001

[2] Tecco Coordination Systems: Corso Tutorial. http://www.tecco.at

[3] Ed Roman, Scott Ambler: Matering Enterprise Java Beans. Wiley Computer Publishing 2002

[4] J2EE Connector Architecture Specification, Version 1.0
http://java.sun.com/j2ee/connector/download.html

[5] J2EE Connector Architecture Specification, Version 1.5
http://java.sun.com/j2ee/connector/download.html