

Analyse der Enterprise-Java-Beans Spezifikation Version 2.0 als Basis für FlexiTRUST

6. August 2002

Markus Winkler

Studienarbeit an der
Technischen Universität Darmstadt
Fachbereich Theoretische Informatik
Kryptographie und Computeralgebra
Prof. Dr. Johannes Buchmann
Alexander Wiesmaier

Markus Winkler, Ludwigsplatz 8 a, 64283 Darmstadt, markus.winkler@gmxpro.de

Inhaltsverzeichnis

1	Einleitung	6
2	EJB-Architecture (EJBA)	7
2.1	Stateful-Session-Bean	7
2.1.1	Life-Cycle	7
2.2	Stateless-Session-Beans	8
2.2.1	Life-Cycle	9
2.2.2	Stateless-Session-Bean als Front-End-Façade	9
2.3	Entity-Bean	11
2.3.1	Life-Cycle	11
2.4	Message-Driven-Beans	12
2.4.1	Exkurs: Was ist Messaging?	12
2.4.2	Betrachtung der Message-Driven-Beans	13
2.4.3	Life-Cycle	14
2.4.4	MDBs in FlexiTRUST?	15
3	Persistenz	16
3.1	Bean-Managed-Persistence	17
3.2	Container-Managed-Persistence	17
3.2.1	Container-Managed-Fields	17
3.2.2	EJB-QL und ejbSelect	18
3.3	BMP oder CMP?	19

<i>INHALTSVERZEICHNIS</i>	3
4 Clustering	21
4.1 Cluster Implementierung	21
4.1.1 Unabhängige JNDI Bäume	22
4.1.2 Zentralisierter JNDI Baum	22
4.1.3 Verteilter globaler JNDI Baum	22
5 Designvorschläge für FlexiTRUST	24
5.1 Design: DAO, Façade, Value-Object	24
5.2 Design: DAO, Façade, Value-Object, MDB, BMP	24

Abbildungsverzeichnis

2.1	Life-Cycle - Stateful-Session-Bean	8
2.2	Life-Cycle - Stateless-Session-Bean	9
2.3	Front-End-Façade Struktur	10
2.4	Sequence-Diagram	10
2.5	Life-Cycle - Entity-Bean	12
2.6	Life-Cycle - Message-Driven-Bean	14
5.1	Design - Synchrone Realisierung	25
5.2	Design - Asynchrone Realisierung	26

Abkürzungsverzeichnis

API	Application-Programming-Interface
BMP	Bean-Managed-Persistence
BO	Business-Object
CMP	Container-Managed-Persistence
CORBA	Common-Object-Request-Broker-Architecture
DAO	Data-Access-Object
EJB	Enterprise-Java-Bean
IOR	Interoperable-Object-Reference
J2EE	Java-2-Platform-Enterprise-Edition
JDBC	Java-Database-Connectivity
JDK	Java-Development-Kit
JMS	Java-Message-Service
JNDI	Java-Naming-and-Directory-Interface
JVM	Java-Virtual-Machine
MDB	Message-Driven-Bean
MOM	Message-Oriented-Middleware
ORB	Object-Request-Broker
PTP	Point-To-Point
Pub	Publish
RR	Request-Reply
SQL	Structured-Query-Language
Sub	Subscribe
XML	EXtensible-Markup-Language

Kapitel 1

Einleitung

Bei Enterprise-Java-Beans (EJB) handelt es sich um Software-Komponenten, die der EJB-Spezifikation der Firma Sun Microsystems folgen. Die Veröffentlichung der ersten Spezifikation im Jahre 1998 erregte viel Aufsehen in der Java Szene. Bedeutende Softwarehersteller wie Oracle, Borland, Symantec oder Sybase brachten schon kurze Zeit später Produkte auf den Markt, die der Spezifikation folgten. Mittlerweile liegt sie in der Version 2.0 aus dem Jahre 2001 vor.

Die Zielsetzung dieser Studienarbeit ist die Analyse der EJB Spezifikation 2.0 im Hinblick auf eine Neuentwicklung der Certification-Authority für FlexiTRUST des Fachbereichs Theoretische Informatik - Kryptographie und Computeralgebra an der Technischen Universität Darmstadt. FlexiTRUST ist ein in JAVA implementierter Trustcenter. [Mic01a]

Kapitel 2

EJB-Architecture (EJBA)

Entsprechend der EJB Spezifikation 2.0 werden vier verschiedene Beantypen betrachtet:

- Stateful-Session-Beans
- Stateless-Session-Beans
- Entity-Beans
- Message-Driven-Beans

2.1 Stateful-Session-Bean

Stateful-Session-Beans sind zustandsabhängig. Dadurch wird es möglich, diese Beanart für eine Folge von Aufgaben des Servers verwenden zu können. Das könnten beispielsweise ein komplexer Business-Prozess oder ein Multi-Step-Workflow sein. Normalerweise werden die Schritte nach und nach abgearbeitet und benötigen mehrfachen Client-Input.

Stateful-Session-Beans werden im Hauptspeicher gehalten, können jedoch vom Container bei zwischenzeitlicher Nichtverwendung inaktiviert werden, um sie in sekundären Speicher auszulagern.

2.1.1 Life-Cycle

Abbildung 2.1 zeigt den Lebenszyklus einer Stateful-Session-Bean. Zu Beginn des Zyklus befindet sich die Bean im Zustand <Does-Not-Exist>. Initiiert der

Client den Zyklus durch Aufruf der Create-Methode, instantiiert der Container die Bean und ruft die Methoden `<setSessionContext>` und `<ejbCreate>` auf. Die Bean befindet sich nun im Zustand `<Ready>`.

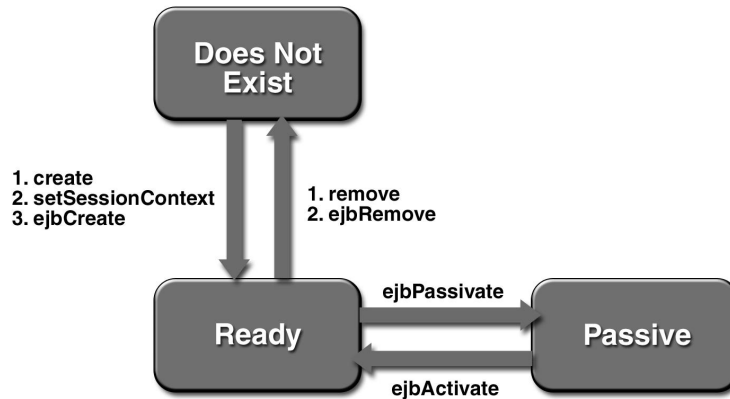


Abbildung 2.1: Life-Cycle - Stateful-Session-Bean

Eine Bean im Zustand `<Ready>` kann durch den Container passiviert werden, sofern sie im Augenblick nicht benötigt wird. Hierdurch wird sie bis zur nächsten Verwendung in den Sekundärspeicher ausgelagert und Arbeitsspeicher freigegeben. Vor der Passivierung wird die Methode `<ejbPassivate>` ausgeführt, sofern diese definiert wurde. Erfolgt ein erneuter Zugriff auf die Bean muss diese nur noch aktiviert werden, d.h. aus dem Sekundär- in den Primärspeicher verschoben werden. Unmittelbar nach der Aktivierung ruft der Container die Methode `<ejbActivate>` auf. Das Ende des Lebenszyklus wird durch den Aufruf der Remove-Methode durch den Client eingeleitet. Der Container führt `<ejbRemove>` aus, bevor die Instanz der Bean durch den Garbage-Collector aufgeräumt werden kann. [Mic02a, Mic01a]

2.2 Stateless-Session-Beans

Stateless-Session-EJBs sind Komponenten, die einen unabhängigen Single-Use-Service implementieren. Stateless-Session-EJBs sind kurzlebig und zustandslos. Für einen Client stellen sie eine Untermenge der Business-Logik dar.

Da Stateless-Session-Beans keinen Zustand besitzen, sind sie für den Server leicht zu handhaben. Dadurch werden sie einfacher skalierbar als die Stateful-Variante oder als Entity-Beans. Stateless-Session-Beans werden in der Praxis als

Front-End-Façade¹ für mehrere Entity-Beans eingesetzt, als Middle-Tier-Proxy für direkte JDBC-Aufrufe oder als einzelner Business-Process mit nur einem Schritt.

2.2.1 Life-Cycle

Abbildung 2.2 zeigt den Lebenszyklus einer Stateless-Session-Bean. Da sie im Gegensatz zu ihrer Stateful-Variante nie passiviert werden kann, besitzt sie nur zwei Zustände: <Does-Not-Exist> und <Ready>. [Mic02a, Mic01a]

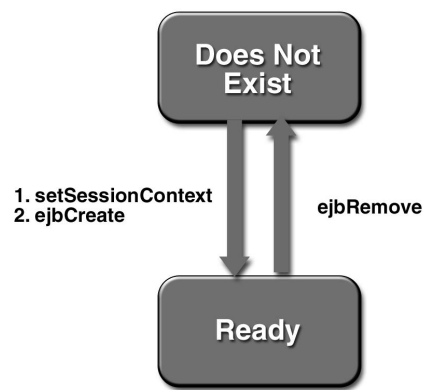


Abbildung 2.2: Life-Cycle - Stateless-Session-Bean

2.2.2 Stateless-Session-Bean als Front-End-Façade

Session-Beans, insbesondere Stateless-Session-Beans, dienen als Façade. Sie nutzen Entity-Beans, können jedoch auch selbstständig über Data-Access-Objects auf die Datenbank zugreifen. Die Aktualisierung der Entities wird von den Entity-Beans selbstständig durchgeführt. Die Session-Beans delegieren an die Entity-Beans und implementieren die Logik, die mehrere Entity-Bean-Instanzen betrifft, beispielsweise Mengenoperationen.

Durch das Session-Façade-Pattern werden folgende Probleme adressiert:

- Zu enge Kopplung mit direkter Abhängigkeit der Clients von den Business-Objects.
- Zu viele Methodenaufrufe zwischen Client und Server, was zu Netzwerk-Performance-Problemen führen kann.

¹ vgl. Kapitel 2.2.2

- Fehlen einer einheitlichen Client-Zugriffs-Strategie und damit Exponierung der Geschäftsobjekte für möglichen Missbrauch.

Der gewählte Ansatz ist eine Session-Bean als Façade um die Komplexität der Interaktionen zwischen den Business-Objects (BO) im Workflow zu kapseln. Die Session-Façade verwaltet die BOs und stellt den Clients eine einheitliche, grobkörnige Schnittstelle (Service-Access-Layer) zur Verfügung.

Die folgenden Abbildungen 2.3 und 2.4 zeigen die Struktur des Session-Façade-Patterns und stellen die Interaktionen einer Session-Façade mit zwei Entity-Beans, einer Session-Bean und einem Data-Access-Object dar.

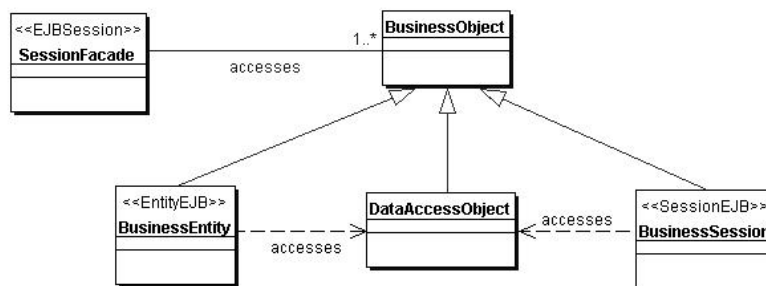


Abbildung 2.3: Front-End-Façade Struktur

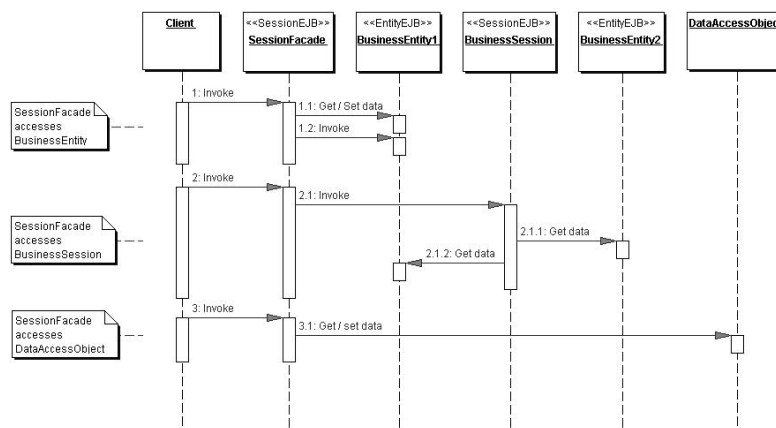


Abbildung 2.4: Sequence-Diagram

Im Rahmen des FlexiTRUST sollte dieses Pattern Anwendung bei der Realisierung der Worker und Workbenches finden. [Mic02c, Mic02b]

2.3 Entity-Bean

Entity-Beans sind Darstellungen persistenter Daten. Beim Erzeugen einer Entity werden die Daten in einem persistenten Speicher eingetragen. Dies geschieht im Fall einer Datenbank typischerweise durch SQL-Insert. Gleichzeitig wird eine Kopie der Daten im Speicher als Teil der EJB oder als Aggregation von Objekten, die von der EJB verwaltet werden, gehalten. Wenn die Attribute der Objekte im Speicher geändert werden, wird vom Container die Aktualität der gespeicherten Werte sichergestellt. Da die Attribute der EJB persistent gespeichert sind, können viele Clients auf unterschiedliche Art und Weise Zugriff auf die selben Daten erhalten. Der Container kann den Zugriff durch ein einzelnes Objekt serialisieren, oder er hält mehrere Kopien der Daten im Speicher. Er sorgt in diesem Fall für den Abgleich der Daten im Arbeitsspeicher.

Entity-Beans sind primär für das Speichern, Laden, Aktualisieren und Löschen von Daten zuständig. Die eigentliche Business-Logik wird in davor geschalteten Session-Beans implementiert². [Mic02a, Mic01a]

2.3.1 Life-Cycle

Abbildung 2.5 zeigt den Lebenszyklus einer Entity-Bean. Im Gegensatz zum Zyklus von Session-Beans wird dieser ausschließlich durch den Container kontrolliert.

Zu Beginn des Zyklus befindet sich die Bean im Zustand `<Does-Not-Exist>`. Nachdem der Container eine Instanz der Bean erzeugt hat, ruft er die Methode `<setEntityContext>` auf, in der gegebenenfalls ein Data-Access-Object instanziiert wird und Referenzen auf andere EJBs angelegt werden.

Die Bean befindet sich nun in einen Pool verfügbarer Instanzen, ist jedoch noch nicht an eine bestimmte Identität gebunden. Alle Instanzen im Pool sind identisch. Erst durch den Übergang in den Zustand `<Ready>` erhält die Bean ihre Identität. Dieser Übergang von `<Pooled>` in `<Ready>` kann auf zwei Arten erfolgen. Der Client ruft die Create-Methode auf, was den Container dazu veranlasst, `<ejbCreate>` und `<ejbPostCreate>` der Bean durch zu führen, oder die Bean wurde im Vorfeld passiviert und wird wieder aktiviert.

Am Ende des Lebenszyklus einer Entity-Bean entfernt der Container die Bean aus dem Pool und ruft die Methode `<unsetEntityContext>` der Bean auf.

²vgl. Kapitel 2.2.2

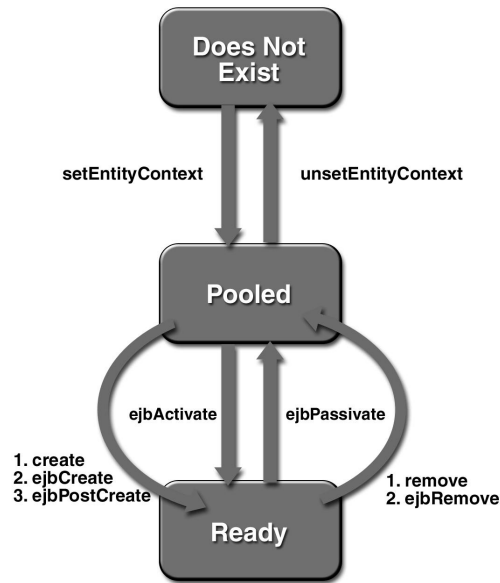


Abbildung 2.5: Life-Cycle - Entity-Bean

2.4 Message-Driven-Beans

Durch EJB 2.0 wird eine neue, dritte Bean-Art neben Session- und Entity-Beans eingeführt: die Message-Driven-Bean.

2.4.1 Exkurs: Was ist Messaging?

Messaging ist ein Mechanismus der es Programmen erlaubt, miteinander *asynchron* zu kommunizieren. Prinzipiell können drei Kategorien der Message-Oriented-Middleware (MOM) unterschieden werden, die definieren, *welcher* Client die Message erhält:

1. Publish/Subscribe (Pub/Sub)
2. Point-To-Point (PTP)
3. Request-Reply (RR)

zu 1.:

Publish/Subscribe kann als Event-Driven-Messaging betrachtet werden.

Pub/Sub-Domain: Die Clients (Konsumenten/Abonnenten) registrieren sich bei einem Publisher/Producer und erhalten die Messages, sobald er diese verteilt. Die Messages stellen hier einen Event dar, der Publisher ist die Quelle und die Clients sind die Event-Listener.

Pub/Sub-Topic: Hier sind die Producer (Erzeuger) und Clients voneinander getrennt. Die Clients melden sich nun bei der MOM mit einem so genannten Topic an und erhalten entsprechende Messages. Die MOM ist für die Verteilung zuständig, die Producer sind von dieser Aufgabe entlastet.

zu 2.:

PTP-Domain: In einem PTP-System versendet ein Client/Producer die Message direkt an einen weiteren Client. Diese Art des Messagings kann uni- oder bidirektional erfolgen.

PTP-Queues: Bei einer PTP-Queue können mehrere Producer ihre Messages in eine Warteschlange einstellen. Die Queue serialisiert die Messages und mehrere Clients können von der Queue die Messages abholen. Diese Queue arbeitet nach dem First-In-First-Out Prinzip, das heißt, die älteste Message wird zuerst entnommen.

zu 3.:

RR-Domain: Bei einer Request/Reply-Domain sendet ein Client eine Message, hier ein Request, und erwartet im Gegenzug eine Message als Reply.

Bei JMS werden nur die Varianten PTP-Domain und Pub/Sub-Domain unterstützt! [BS01]

2.4.2 Betrachtung der Message-Driven-Beans

Diese Beans besitzen weder Local- noch Remote-Interfaces. Die Verwendung der Message-Driven-Beans hat gegenüber dem herkömmlichen Message-Listener den Vorteil, dass eingehende Nachrichten parallel von mehreren Bean-Instanzen bearbeitet werden können, was üblicherweise nur durch Eigenimplementierung zu realisieren wäre. Der EJB-Container hält zu diesem Zweck einen Pool von Instanzen der Bean-Klasse bereit. Dabei ist zu beachten, dass eine Message-Driven-Bean wie ein Stateless-Session-Bean keinen Zustand besitzt.

Eine Message-Driven-Bean (MDB) unterscheidet sich von Session- und Entity-Beans vor allem dadurch, dass die Clients sie nicht durch Interfaces ansprechen. Eine MDB hat ausschließlich eine Bean-Klasse. Wenn MDB eine Event-Notification erhält, ruft der Container die Methode `<onMessage>` auf, um die

Abarbeitung anzustoßen. Innerhalb der `onMessage`-Methode können Hilfsmethoden mit der Business-Logik aufgerufen werden, oder es wird eine Instanz einer Session oder Entity-Bean erzeugt, welche die Daten verarbeiten.

Mit dem Versenden einer Message an eine MDB beginnt eine Transaktion, die bis zum Ende der `onMessage`-Methode erhalten bleibt. Im Falle eines Rollbacks wird die Message vom Container aus erneut versendet. Tritt jedoch serverseitig, beispielsweise innerhalb einer an der Transaktion beteiligten Session- oder Entity-Bean eine Exception auf, wird der Client davon nicht benachrichtigt! Folglich kann bei komplexer Business-Logik die Verwendung von Message-Beans problematischer sein, als die Benutzung herkömmlicher synchroner Aufrufe. [MH00, Ma102]

2.4.3 Life-Cycle

Abbildung 2.6 zeigt den Lebenszyklus einer Message-Driven-Bean. Beim Start des Servers erzeugt der Container einen Pool von Message-Driven-Bean-Instanzen. Für jede der Instanzen ruft der Container die Methoden `<setMessageDrivenContext>` und `<ejbCreate>` auf. Wie die Stateless-Session-Bean³ wird eine MDB niemals passiviert und besitzt nur die beiden Zustände `<Does-Not-Exist>` und `<Ready>`.

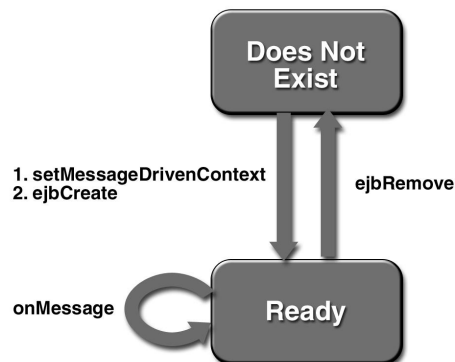


Abbildung 2.6: Life-Cycle - Message-Driven-Bean

Am Ende des Lebenszyklus der Bean ruft der Container die Methode `<ejbRemove>` auf. [Mic02a]

³vgl. Kapitel 2.2

2.4.4 MDBs in FlexiTRUST?

Vorteile

- Der Client muss sich nicht mit Home- und Remote-Interfaces auseinander setzen, um mit dem Server kommunizieren zu können. Lediglich die JMS-API muss hier bekannt sein.
- Der Client ist von der Geschäftslogikschicht völlig entkoppelt. Die einzige Koppelung besteht durch den Namen der Queue und der Message.
- Durch die gegebene Asynchronität der Kommunikation mit der MDB braucht der Client nicht zu warten, bis der Vorgang beendet ist. Er kann mit seiner Arbeit sofort fortfahren.

Nachteile

- Die Kommunikationsschnittstelle zu MDBs ist zu allgemein. Die Art der übergebenen Messages wird erst zur Laufzeit überprüft. Das kann zu Laufzeitproblemen wie z.B. `java.lang.ClassCastExceptions` führen. Ein „intelligenter“ Code wird erforderlich.
- Durch die Asynchronität lassen sich nur Prozesse durchführen, die lediglich aus dem Request- und nicht auch aus einem Response-Teil bestehen.
- Fehlende Benachrichtigung des Clients im Falle von serverseitigen Exceptions
- Für die MDB kann keine Security-Identity eingetragen werden.

Einsatzmöglichkeiten für FlexiTRUST

- Als Ersatz einer Queue für die Eingangs-Queues der Workbenches.
- Für die Kommunikation mit dem Stock.

In Bezug auf FlexiTRUST, insbesondere bei der Versendung eines Workers, ist besonderes Augenmerk auf das Exceptionhandling zu legen. [Bie02]

Kapitel 3

Persistenz

Persistenz einer EJB bedeutet, dass die Daten der EJB persistent sind bzw. existieren unabhängig davon, ob die Entity zur Zeit im Speicher ist oder nicht.

Wenn eine Entity-Bean erzeugt wird, müssen die Daten, welche die Entity repräsentieren, persistent gespeichert werden. Gleichzeitig wird eine Kopie der Daten im Speicher als Teil der EJB gehalten. Immer wenn Attribute der Entity geändert werden, müssen die Daten der darunter liegenden Persistenzschicht automatisch aktualisiert werden.

Da die Daten persistent gespeichert werden, beispielsweise in einer SQL-Datenbank, können mehrere Clients auf die selben Daten zur gleichen Zeit zugreifen. Der Container hat zwei Möglichkeiten, den Zugriff mehrerer Clients zu implementieren. Er kann die Zugriffe durch ein einzelnes Objekt serialisieren, welches nicht für mehrere Clients gleichzeitig benutzbar ist. Alternativ können zwei In-Memory Kopien der Daten angelegt werden deren Synchronisation vom Container durchgeführt wird.

Zwei Varianten der Persistenz werden unterschieden:

1. Bean-Managed-Persistence (BMP)
2. Container-Managed-Persistence (CMP)

Beide verwenden die gleichen Mechanismen und zeigen ein identisches Clientverhalten auf. Der einzige Unterschied ist der Datenbankzugriff. Bei BMP wird er durch den Bean-Entwickler implementiert, im Fall von CMP wird dieser durch den Container generiert. [MH00]

3.1 Bean-Managed-Persistence

Bei der Bean-Managed-Persistence (BMP) ist die Entity selbst für die Logik der persistenten Speicherung und der Datensynchronisation zuständig. Die vier Kernfunktionalitäten, die die Entity durchführen können muss, sind:

1. Einen Eintrag dem persistenten Speicher hinzufügen (SQL-INSERT).
2. Einen Eintrag aus dem persistenten Speicher entfernen (SQL-DELETE).
3. Einen Eintrag entsprechend den momentanen Attributwerten der Entity im persistenten Speicher aktualisieren (SQL-UPDATE).
4. Einen Eintrag entsprechend den Werten des persistenten Speichers innerhalb der Entity aktualisieren (SQL-SELECT).

Folglich ist der Entwickler für die Datenbankstatements verantwortlich. Diese werden entweder direkt in der EJB oder in einem von ihr verwendetem Data-Access-Object (DAO) implementiert. Ein DAO ist ein Objekt, welches von der Entity verwendet wird und den Zugriff auf den persistenten Speicher durchführt. Die Implementierung der DAOs ist austauschbar wobei die EJBs beim Tausch des DAOs nicht geändert werden müssen. Sie können unabhängig vom Container optimiert und getestet werden, wodurch die Entwicklung erleichtert wird. Weiterhin bieten sie komplexere Zugriffsmöglichkeiten auf Datenbanken, als dieses mit Container-Managed-Persistence möglich wäre¹.

3.2 Container-Managed-Persistence

Bei Container-Managed-Persistence (CMP) ist die gesamte Logik der persistenten Speicherung der Entity im Container enthalten. Der Container koordiniert alle Teile automatisch anhand von Regeln, die der Entwickler durch die Systemkonfiguration definiert hat. Er erzeugt alle notwendigen Tabellen und SQL-Statements. Die Aufgaben des Entwicklers beschränken sich auf die Deklaration der Felder und Finder-Methoden im Deployment-Descriptor.

3.2.1 Container-Managed-Fields

Es werden zwei Arten von Container-Managed-Fields einer Entity unterschieden: Persistent-Fields und Relation-Fields. Persistent-Fields sind direkt einem

¹vgl. Kapitel 3.2

persistent gespeicherten Wert zugeordnet, typischerweise einer Spalte innerhalb einer Datenbanktabelle. Sie werden nicht als Attribute der implementierenden Klasse gespeichert. Der Zugriff auf diese Felder ist ausschließlich durch abstrakte Accessor-Methoden möglich.

Bean-Implementierung:

```
public abstract class WorkerEJB implements EntityBean {
    public abstract String getWorkerID();
    public abstract void setWorkerID(String workerID);
    ...
}
```

Deployment-Descriptor:

```
<ejb-jar><enterprise-beans><entity>
  <ejb-name>WorkerEJB</ejb-name>
  <cmp-field><field-name>workerID</field-name></cmp-field>
</entity></enterprise-beans></ejb-jar>
```

Relation-Fields referenzieren eine andere Entity oder hängen von Entity-Objekten ab. Ihre Deklaration erfolgt ebenfalls innerhalb des Deployment-Descriptors, wobei hier auf eine genauere Betrachtung verzichtet wird. [Mic01a, SJG02a]

3.2.2 EJB-QL und ejbSelect

Die EJB-Query-Language wurde definiert, um Finder- und ejbSelect-Methoden plattformunabhängig definieren zu können. Die ejbSelect-Methoden enthalten Abfragen innerhalb der Implementierung einer Entity. Anders als Finder können diese Methoden Entities oder einzelne Felder einer Entity liefern.

Die EJB-QL ist eine Teilmenge von SQL 92 mit Erweiterungen für die Navigation über Beziehungen. [SJG02b]

Home-Interface:

```
public interface WorkerHome implements EJBHome {
    Collection findExpiredWorkers(long expireDate)
        throws FinderException;
}
```

Deployment-Descriptor:

```

<ejb-jar><enterprise-beans><entity>
  <ejb-name>WorkerEJB</ejb-name>
  <query>
    <query-method>
      <method-name>findExpiredWorkers</method-name>
      <method-params><method-param>
        long
      </method-param></method-params>
    </query-method>
    <ejb-ql><![CDATA[
      SELECT OBJECT(w)
      FROM worker w
      WHERE w.expireDate < ?1
    ]]></ejb-ql>
  </query>
</entity></enterprise-beans></ejb-jar>

```

3.3 BMP oder CMP?

Die größte Problematik für den Container bei Verwendung von CMP stellen effiziente Datenbankzugriffe dar. Verdeutlichen kann man sich dies anhand folgender Beispiele:

Es existiere eine Tabelle CD

```
CD: PK(id), title, artist, year
```

wobei die einzelnen CDs durch Entity-Beans realisiert sind. Sollen nun alle CDs gefunden werden, deren Artist „chopin“ enthält würde ein SQL-Statement folgendermaßen aussehen:

```
SELECT FROM CD WHERE artist LIKE "%chopin%"
```

Im Umfeld der CMP würde der Container durch den Aufruf der Methode "findAll()" des Home-Interfaces der CDEJB alle derzeitigen Instanzen von CD selektieren

```
SELECT FROM CD;
```

und dann für jede Zeile eine Instanz der Entity erzeugen. Dann müssen alle Objekte anhand des Auswahlkriteriums überprüft werden, wofür die jeweiligen

Attribute aus der Datenbank gelesen werden. Dieses geschieht nach und nach für jede einzelne Entity, da der Container versucht, den Arbeitsspeicherbedarf zu minimieren. Für jedes zu untersuchende Objekt wird folgendes Statement ausgeführt:

```
SELECT FROM CD WHERE ID=xxxx
```

Folglich werden für jede in der Datenbank vorhandene Zeile eine Entity instanziiert und ein eigenes SQL-Statement ausgeführt. Im Vergleich hierzu würde bei BMP nur ein Datenbankzugriff für die Suche der Entity erfolgen und anschließend nur Instanzen „passender“ Objekte erzeugt werden.

Eine weitere Problematik der Container-Managed-Persistence erschließt sich, wenn man sich die Vorgehensweise des Containers bei der Aktualisierung eines oder mehrerer Attribute verdeutlicht. Hat sich ein Attribut der Entity geändert, wird der Container ein SELECT auf die ganze Zeile durchführen, die Attribute zwischenspeichern, um die Zeile anschließend zu löschen und durch einen INSERT wieder einzufügen. Werden nun nacheinander Attribute der Entity geändert, wird für jedes ein DELETE gefolgt von einem INSERT durchgeführt.

Im Vergleich hierzu ist bei Bean-Managed-Persistence nur ein einziges UPDATE erforderlich. Dieses lässt sich dadurch erklären, dass Änderungen in Attributen im Normalfall durch Business-Methoden der Entity durchgeführt werden. Bei Aufruf einer solchen Methode, im Beispiel

```
void updateBeanData(BeanDetailData bdd)
```

beginnt der Container eine Transaktion, führt die Methode aus und beendet anschließend die Transaktion. Als letzter Schritt wird dann die Methode <ejb-Store> aufgerufen, die über das Data-Access-Object die Daten persistent in die DB einbringt.²

Die Entscheidung, ob CMP oder BMP verwendet werden sollte, hängt im wesentlichen von der Komplexität des Systems ab. Bei CMP übernimmt der Container die meiste Arbeit, der Entwickler muss nur die persistenten Felder definieren. Bei der Entwicklung ist jedoch darauf zu achten, dass diese Technologie durchdacht angewendet wird, da das Endsystem ansonsten sehr schnell ineffizient und schlecht zu warten sein wird. BMP erscheint zuerst unnötig kompliziert und fehleranfällig, bietet jedoch bei einem guten Design bei weitem mehr Flexibilität, Portabilität, Effizienz und Übersichtlichkeit. [Boo02]

²Der Application-Server JBoss in der Version 3.0 besitzt Erweiterungen der EJB-QL, welche die meisten oben beschriebenen Probleme entkräftet. Allerdings ist man bei ihrer Verwendung auf JBoss festgelegt. Für nähere Informationen sei auf [SJG02b] verwiesen.

Kapitel 4

Clustering

Ein Cluster ist eine Ansammlung von Computern oder Serverinstanzen. Clustering wird verwendet um Fehlertoleranz und Lastverteilung zu erreichen.

Es werden zwei Varianten des Clusterings unterschieden:

- Shared-Nothing und
- Shared-Disk

In einem Shared-Nothing-Cluster verfügt jeder Application-Server über ein eigenes Filesystem mit einer eigenen Kopie der Applikation. Dadurch müssen bei Updates des Codes alle Server aktualisiert werden. Im Gegensatz dazu verwendet ein Shared-Disk-Cluster ein gemeinsames Plattensystem, wodurch ein Aktualisierungsprozess vereinfacht wird. Diese Variante besitzt jedoch mit ihrem einzigen Filesystem einen Single-Point-Of-Failure. [Mic01b, Kan01a, Kan01b, Lab02]

4.1 Cluster Implementierung

Das JNDI stellt den Kern einer jeden J2EE Application dar. Im Umfeld eines Clusters gestaltet sich seine Implementierung jedoch schwierig, da nicht mehrere Objekte an einen einzelnen Namen gebunden werden können.

Zur Zeit existieren drei verschiedene Implementierungen:

- Unabhängige JNDI Bäume (Independent-JNDI-Tree)
- Zentralisierter JNDI Baum (Centralized-JNDI-Tree)

- Verteilter globaler JNDI Baum (Shared-Global-JNDI-Tree)

Die Art der Implementierung wird durch den verwendeten Application-Server festgelegt.

4.1.1 Unabhängige JNDI Bäume

Jeder der Application-Server verwaltet einen unabhängigen JNDI-Baum. Die einzelnen Server in einem JNDI-Tree-Cluster wissen nicht um die Existenz anderer Server im Cluster. Ein Vorteil dieses Verfahrens ist die einfache Skalierbarkeit durch einfaches Hinzufügen eines weiteren Servers.

Die Ausfallsicherung (Failover) liegt im Verantwortungsbereich des Entwicklers. Da hier unabhängige JNDI-Bäume verwendet werden, sind die Remote-Proxies an den Server gebunden, auf dem der Lookup durchgeführt wurde. Wenn nun ein Methodenaufruf an eine EJB fehl schlägt, muss der Entwickler zusätzlichen Code implementiert haben, der die Adresse eines weiteren aktiven Servers bestimmt und einen zusätzlichen JNDI-Lookup durchführt.

4.1.2 Zentralisierter JNDI Baum

In dieser Variante wird ein zentralisierter JNDI-Baum auf einem oder mehreren Name-Servern verwaltet, an den sich die Objekte beim Start eines Servers einbinden. Um die Referenz auf eine EJB zu erhalten, muss der Client einen Look-Up auf das Home-Interface über den Nameserver durchführen, welcher eine Interoperable-Object-Reference (IOR) zurückliefert. Eine IOR zeigt auf mehrere aktive Server innerhalb des Clusters, die dieses Objekt zur Verfügung stellen. Der Client wählt im nächsten Schritt die erste Referenz und erhält das gewünschte Home und Remote. Für den Fehlerfall ist im CORBA-Stub eine Logik enthalten, die die Objekte von einem alternativen Server liefert.

Im Fehlerfall der Name-Server kann jedoch der gesamte Cluster zusammenbrechen, obwohl die eigentlichen Application-Server funktionsfähig wären.

4.1.3 Verteilter globaler JNDI Baum

Beim Start eines Server im Cluster meldet er sich und seinen JNDI-Baum bei allen anderen Servern an. Jeder Server bindet den neuen Baum in seine eigene Struktur ein. Durch dieses Vorgehen besitzt jeder der Server einen kompletten

Datenbestand. Ein Cluster nach diesem Verfahren zeichnet sich durch eine hohe Skalier- und Verfügbarkeit aus. Versagt einer der Server, bleibt der Cluster trotzdem funktionsfähig.

Nachteilig bei diesem Verfahren ist jedoch der hohe Datenverkehr für den Abgleich der JNDI Bäume.

Kapitel 5

Designvorschläge für FlexiTRUST

Die folgenden Designvorschläge verwenden Bean-Managed-Persistence bei synchroner-, bzw. asynchroner Kommunikation zwischen den Workbenches.

5.1 Design: DAO, Façade, Value-Object

Die Abbildung 5.1 zeigt ein Klassendiagramm einer möglichen Worker-Workbench-Struktur, umgesetzt nach den Sun J2EE Design-Patterns Session-Façade, Data-Access-Object und Value-Object.

Die Workbench ist als Stateless-Session-Bean realisiert und stellt die Façade für den Zugriff auf die Worker dar. Die Worker werden als Entity-Beans umgesetzt, da sie die eigentlichen Daten repräsentieren. Im Package Domain liegen die Value-Objects, welche direkt von den Data-Access-Objects verwendet werden, um die SQL-Statements zu vervollständigen. [Mic02b, Mic02c, Mic02d, Car02, Hau01]

5.2 Design: DAO, Façade, Value-Object, MDB, BMP

Die Abbildung 5.2 zeigt ein Klassendiagramm einer möglichen Worker-Workbench-Struktur, umgesetzt nach den Sun J2EE Design-Patterns Session-Façade,

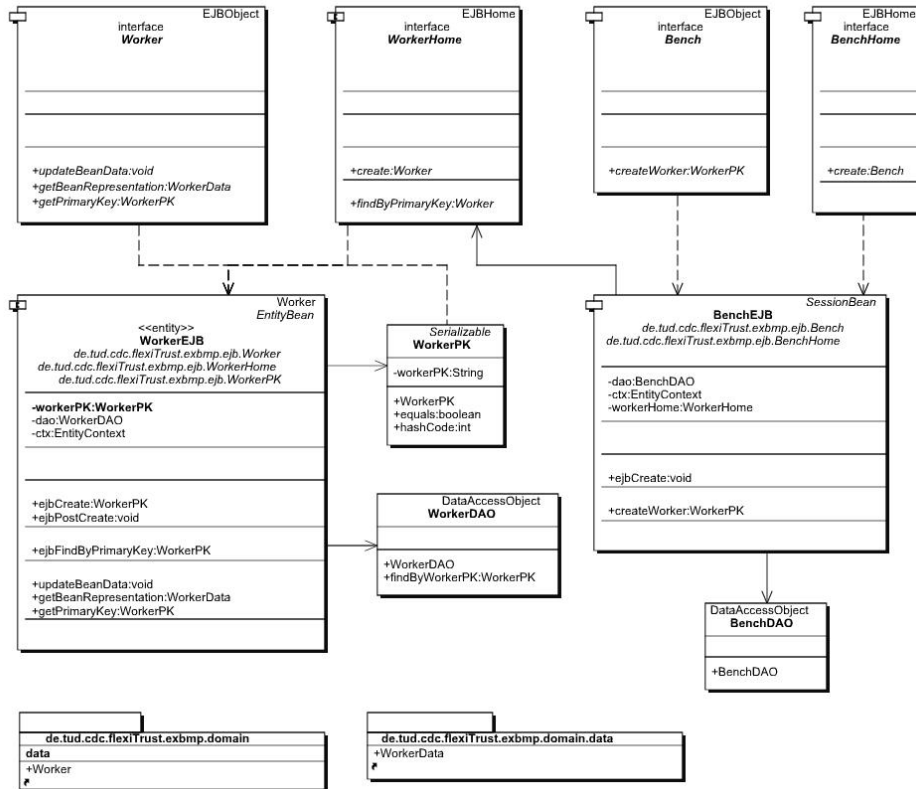


Abbildung 5.1: Design - Synchrone Realisierung

Data-Access-Object und Value-Object unter Verwendung einer Message-Driven-Bean als Einstiegspunkt der Workbench. [Mic02b, Mic02c, Mic02d]

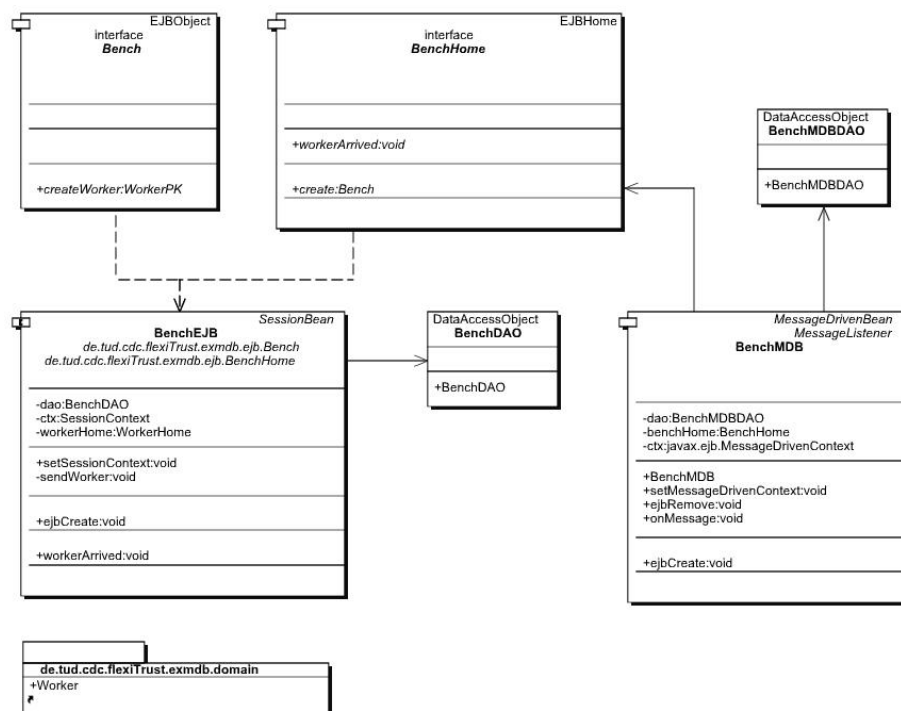


Abbildung 5.2: Design - Asynchrone Realisierung

Literaturverzeichnis

- [Bie02] Kap. 5 Die Integrationsschicht In: BIEN, Adam: *J2EE Patterns, Entwurfsmuster für die J2EE*. München : Addison-Wesley Longman Verlag GmbH, 2002. – ISBN 3-8273-1903-X
- [Boo02] Kap. 3 Using container-managed persistence In: BOONE, Kevin: *JBoss 2.4+ Documentation*. <http://www.jboss.org/online-manual/HTML/index.html>, Juni 2002. – JBoss Online Manual
- [BS01] BEA SYSTEMS, Inc.: *BEA Education Services - Developing Enterprise Applications Using EJB*. 2315 North First Street, San Jose, CA 95131 : BEA Systems, Inc., 2001. – <http://www.bea.com>
- [Car02] CARMAN, James: *Write once, persist anywhere*. http://www.javaworld.com/javaworld/jw-03-2002/jw-0301-dao_p.html. März 2002. – JavaWorld
- [Hau01] HAUCK, Peter: Fundsachenmanagement mit EJBs Erfahrungen und Patterns, TLC GmbH - Deutsche Bahn Gruppe, September 2001
- [Kan01a] KANG, Abraham: *J2EE clustering, Part 1*. <http://www.javaworld.com/javaworld/jw-02-2001/jw-0223-extremescale.html>. Februar 2001. – JavaWorld
- [Kan01b] KANG, Abraham: *J2EE clustering, Part 2*. http://www.javaworld.com/javaworld/jw-08-2001/jw-0803-extremescale2_p.html. August 2001. – JavaWorld
- [Lab02] Kap. 11 HOWTO - Clustering in JBoss 3.0 alpha In: LABOUREY, Sacha: *JBoss 2.4+ Documentation*. <http://www.jboss.org/online-manual/HTML/index.html>, 2002. – JBoss Online Manual
- [Mal02] MALANI, Prakash: *Transaction and redelivery in JMS*. http://www.javaworld.com/javaworld/jw-03-2002/jw-0315-jms_p.html. März 2002. – JavaWorld

- [MH00] MONSON-HAEFEL, Richard: *Read all about EJB 2.0*. http://www.javaworld.com/javaworld/jw-06-2000/jw-0609-ejb_p.html. Juni 2000. – JavaWorld
- [Mic01a] MICROSYSTEMS, Sun. *Enterprise JavaBeans Specification, Version 2.0*. <http://java.sun.com/products/ejb/docs.html#\#specs>. 2001
- [Mic01b] MICROSYSTEMS, Sun. *Migrate Your Application from a Single Machine to a Cluster, the Easy Way*. <http://developer.java.sun.com/developer/technicalArticles/J2EE/clusteri%ng/>. September 2001
- [Mic02a] MICROSYSTEMS, Sun. *The J2EE Tutorial - Enterprise Beans*. http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/EJBConcepts.html. 2002
- [Mic02b] MICROSYSTEMS, Sun. *Sun Java Center J2EE Patterns - Data Access Object*. <http://developer.java.sun.com/developer/restricted/patterns/DataAccess0%bject.html>. 2002
- [Mic02c] MICROSYSTEMS, Sun: *Sun Java Center J2EE Patterns - Session Facade*. <http://developer.java.sun.com/developer/restricted/patterns/SessionFaca%de.html>. 2002. – <http://developer.java.sun.com/developer/restricted/patterns/J2EETPattern%AtAGlance.html>
- [Mic02d] MICROSYSTEMS, Sun: *Sun Java Center J2EE Patterns - Value Object*. <http://developer.java.sun.com/developer/restricted/patterns/ValueObject%.html>. 2002. – <http://developer.java.sun.com/developer/restricted/patterns/J2EETPattern%AtAGlance.html>
- [SJG02a] Kap. 3 CMP-Fields In: SUNDSTROM, Dain ; THE JBOSS GROUP: *JBossCMP*. 2520 Sharondale Dr., Atlanta, GA 30305 USA : JBoss Group, LLC, Juni 2002
- [SJG02b] Kap. 5 Queries In: SUNDSTROM, Dain ; THE JBOSS GROUP: *JBossCMP*. 2520 Sharondale Dr., Atlanta, GA 30305 USA : JBoss Group, LLC, Juni 2002