

# Die neue Version von EJB (EJB 3.0)

von

Robert Switzer

## 1 Präambel

Nach meinem Vortrag über Enterprise JavaBeans (EJB) in Februar protestierte Franz-Josef (sinngemäß) “Das ist leider alles überholt, denn jetzt gibt es EJB 3.0 und das soll viel einfacher sein”.

Einerseits hatte Franz-Josef da Recht und andererseits lag er mit seinem Einwand ganz falsch:

- Viele Entwickler haben erkannt, daß EJB viel zu kompliziert war. In meinem Vortrag haben wir gesehen, daß man für jedes Enterprise Bean mehrere Artefakte entwickeln muß:
  - Außer der Beanklasse selbst, um die es eigentlich geht, braucht man einige Interfaces.
  - Man braucht etliche XML-Deskriptordokumente, die nicht leicht zu schreiben sind.
  - In der Beanklasse werden diverse Callbackmethoden benötigt, die entweder trivial (und darum nur lästig) oder aber relativ knifflig zu programmieren sind.
- Diese kritischen Entwickler haben Ihre Erkenntnisse in den Ruf nach einem neuen Java Specification Request (JSR) fließen lassen. So entstand JSR 220, der zu einem ganz neuen EJB führen sollte.

Und hier ist der Punkt, wo Franz-Josef falsch lag; bis vor wenigen Tagen gab es JSR 220 erst als Early Draft Review. Heute liegt allerdings die Public Review Version vor und die meisten Mitglieder der Expertengruppe haben diese Version bereits abgenickt. Eine Final Draft Version ist zu erwarten, so bald die von dem JCP vorgeschriebene Frist abgelaufen ist.

Man erwartet, daß sich niemand trauen würde, in diesem frühen Stadium eine Implementation von EJB 3.0 vorzulegen. Aber es gibt erstaunlicherweise eine solche Implementation; die JBoss-Gruppe hat ihren Anwendungsserver mit einer vorläufigen Version von EJB 3.0 ausgestattet, damit die Entwickler das bisher Spezifizierte testen und ihre Erkenntnisse dazu in den Spezifizierungsprozeß einfließen lassen können. Die JBoss-Leute sind selbst aktive Mitglieder der JSR220-Expertengruppe.

In diesem Vortrag will ich über meine Erfahrungen mit dieser vorläufigen Version von EJB 3.0 referieren.

*Bemerkung.* Seit Ende Juni gibt es auch von Oracle eine Implementation von EJB 3.0, die die Persistenz mit TopLink implementiert – genauso wie die JBoss-Implementation auf Hibernate basiert.

## 2 Was bringt uns EJB 3.0?

Die Expertengruppe nennt folgende Ziele für die neue Spezifikation:

- Elimination der Deskriptoren.
- Kapselung der Abhängigkeiten von der Umgebung und dem JNDI-Zugriff mit Hilfe der in Java 5 neu eingeführten Annotationen sowie durch Dependency Injection.
- Enterprise Beans sollten so weit vereinfacht werden, daß sie eine größere Ähnlichkeit mit POJOs bzw. JavaBeans besitzen.
- Elimination der speziellen Komponenteninterfaces (remote und local); hierfür sollten normale Javainterfaces ausreichen.
- Elimination der Homeinterfaces.
- Elimination von Callbacks. Nicht alle Callbacks waren lästig bzw. überflüssig; man hat solche übriggelassen (bzw. neue eingeführt), die besonders sinnvoll erscheinen.
- Verbesserte Möglichkeiten für das Testen (z. B. Unit Tests) außerhalb des Anwendungsserver.

Ferner hat die TrailBlazer-Dokumentation [tb] von JBoss folgendes über EJB 3.0 zu sagen:

EJB 3.0 ist nicht nur eine standardisierte J2EE Programmierschnittstelle, sondern auch eine neue Herangehensweise an die Anwendungsentwicklung. Die Essenz von EJB 3.0 ist der Gebrauch von POJOs, von Java-Annotationen, sowie vom Entwurfsmuster Dependency Injection, um rasch und einfach lose gekoppelte und testfähige Anwendungen zu konstruieren.

Ich will heute einige Beispiele für die vereinfachten Enterprise Beans bringen. In meinem Vortrag in Februar war die Beanentwicklung so komplex, daß ich leider keine Zeit hatte, zu den recht wichtigen Entity Beans zu kommen. Heute will ich mindestens ein Beispiel eines Entity Bean bringen.

## 3 Unser erstes Enterprise Bean mit EJB 3.0

Wir beginnen mit einem zustandslosen Session Bean. Ich könnte mit dem leidigen “Hello World”-Beispiel beginnen, damit der Leser sieht, wie viel einfacher die Beanentwicklung geworden ist. Aber ich habe keine Lust, die Welt nochmal zu grüßen. Darum rechnet unser erstes Bean Dollar in Euro um.

Wir beginnen mit der Komponentenschnittstelle (ein ganz normales Java Interface):

```
package convert.bean;

import javax.ejb.Remote;
```

```
import java.math.BigDecimal;

@Remote
public interface Converter
{
    public BigDecimal dollarToEuro(BigDecimal dollars);
}
```

*Kommentar:*

1. Es handelt sich um die Schnittstelle, mit der ein entfernter Client (remote) übers Internet auf unser Bean zugreifen könnte. Das deklarieren wir mit der Annotation `@Remote`. Der Import

```
import javax.ejb.Remote;
```

wird lediglich dafür gebraucht, um diese Annotation verfügbar zu machen.

2. Hier haben wir nur eine einzige Anwendungsmethode, die wir entfernten Clients verfügbar machen wollen: `dollarToEuro`.
3. Diese Schnittstelle unterscheidet sich von “stinknormalen” Java Interfaces lediglich durch die Annotation `@Remote`.

Und nun kommt unsere Beanklasse:

```
package convert.bean;

import javax.ejb.Stateless;

import java.math.*;

@Stateless
public class ConverterBean implements Converter
{
    BigDecimal euroExchangeRate = new BigDecimal("0.8134");

    public BigDecimal dollarToEuro(BigDecimal dollars)
    {
        BigDecimal result = dollars.multiply(euroExchangeRate);

        return result.setScale(2, BigDecimal.ROUND_HALF_UP);
    }
}
```

*Kommentar:*

1. Unsere Beanklasse muß keine speziellen Interfaces implementieren – etwa

`SessionBean`

Sie implementiert lediglich ihre Komponentenschnittstelle `Converter`, was dem erfahrenen Javaprogrammierer ganz logisch erscheint – unter EJB 2.1 jedoch falsch gewesen wäre!

2. Wir charakterisieren unser Bean als zustandslosen Session Bean mit der Annotation `@Stateless`. Auch in diesem Fall dient der Import

```
import javax.ejb.Stateless;
```

dazu, diese Annotation verfügbar zu machen.

3. Angenehm fällt hier auf, daß wir keine Callbackmethoden implementieren müssen – nicht einmal als Stubs.

Und nun kommen wir zu unserem Client, wo wir einige erfreulichen Vereinfachungen beobachten werden.

```
package convert.client;
```

```
import convert.bean.Converter;  
import javax.naming.InitialContext;  
import java.math.BigDecimal;
```

```
public class Client  
{  
    public static void main(String[] args)  
    {  
        try  
        {  
            InitialContext initial = new InitialContext();  
            Converter converter = (Converter)  
                initial.lookup(Converter.class.getName());  
  
            BigDecimal dollars = new BigDecimal("100.00");  
            BigDecimal euros = converter.dollarToEuro(dollars);  
  
            System.out.println("$100.00 are " + euros + " Euro");  
  
            System.exit(0);  
        }  
        catch (Exception ex)
```

```

        {
            System.err.println(ex);
        }
    }
}

```

*Kommentar:*

1. Diejenigen mit einem guten Gedächtnis werden sich erinnern, daß wir damals an vergleichbarer Stelle in meinem Februarvortrag folgende Dinge tun mußten:

- (a) das Ergebnis der `lookup`-Methode war nur ein `Object` und mußte umständlich mit einem Objekt vom Typ

`PortableRemoteObject`

gecastet werden.

- (b) das Ergebnis unserer damaligen Typkonversion war ein Objekt vom Typ

`HelloHome`

Mit der `create`-Methode dieses Homeobjekts konnten wir uns schließlich ein Objekt des gewünschten Typs (`Hello`) verschaffen.

Nun mit EJB 3.0 ist das alles viel einfacher, wie man oben sieht:

- (a) jetzt kommen wir mit einem schlichten Cast aus.
- (b) wir brauchen keine Homeobjekte (die gibt es nicht einmal mehr). Wir bekommen gleich ein Objekt des gewünschten Typs (`Converter`).

2. Die JBoss-Dokumentation verrät, daß Komponenten defaultmäßig unter ihrer qualifizierten (vollen) Namen im JNDI-Baum eingetragen werden. So können wir eine solche Komponente stets mit einer Instruktion wie in unserer Clientklasse finden:

```

Converter converter = (Converter)
    initial.lookup(Converter.class.getName());

```

Es bleibt abzuwarten, ob diese Lösung von JSR 220 übernommen wird.

## 4 Beispiele für Entity Beans

Vorweg möchte ich wieder aus der JBoss-TrailBlazer-Dokumentation [tb] zitieren:

EJB 3.0 Entity Beans werden benutzt, um Tabellen einer relationalen Datenbank zu modellieren und auf diese zuzugreifen. Dieses Persistenzframework basiert ausschließlich auf POJOs und sollte den meisten Java-Programmierern natürlich vorkommen.

Wenn Sie mit Hibernate oder TopLink vertraut sind, müßten Sie die EJB 3.0 Entity Beans intuitiv finden, da deren Entwurfsmodell von dem Zugang von Hibernate und TopLink zur objekt-relationalen Abbildung beeinflusst wurde. Selbst wenn Sie wenig Erfahrung mit komplexer Datenbankprogrammierung in Java haben, werden Sie EJB 3.0 Entity Beans leicht erlernbar finden. Sie müssen nur Ihre Anwendungsdaten in einfachen POJOs modellieren. Dann an Hand einer Menge von leicht zu merkenden Annotationen, ermittelt der Container, wie diese Java-Objekte auf Reihen einer Tabelle in einer relationalen Datenbank abzubilden sind. Mit dem `EntityManager`-API können Sie mit der relationalen Datenbank umgehen, als ob diese Java-Objekte anstatt Reihen von relationaler Daten speichern würde.

Nun kommen wir zu der eigentlichen Krönung dieses Vortrags: einem Beispiel, das ich von dem JBoss-Tutorium abgekupfert und leicht nostrifiziert habe. Dieses Beispiel könnte das Herz eines Online-Ladens bilden; es besteht aus:

- zwei Entity Beans

`Order`  
`LineItem`

wobei `Order` eine Bestellung repräsentieren könnte und `LineItem` einen Einzelposten in der Bestellung darstellt.

Dieses Beispiel ist besonders lehrreich, weil es eine 1-zu-N-Beziehung zwischen `Order` und `LineItem` gibt: zu einer Bestellung gibt es i. Allg. mehrere Einzelposten.

- einem zustandsbehafteten Session Bean `ShoppingCart`. Dieses Session Bean ist das einzige Bean, das dem entfernten Client zugänglich ist. Zu den allgemein akzeptierten Entwurfsmustern bei der EJB-Entwicklung (best practices) gehört es, daß man seine Entity Beans den entfernten Clients nicht direkt zugänglich macht, sondern ein Session Bean als Mittler (facade) dazwischen setzt.

*Bemerkung.* Unter EJB 3.0 in der gegenwärtigen Gestalt wäre es nicht einmal möglich, dieses Muster auszuhebeln und Entity Beans entfernten Clients direkt verfügbar zu machen, da Entity Beans keine Komponentenschnittstellen besitzen – insbesondere kein Remote Interface.

Im Einklang mit den eingangs genannten Zielen von JSR 220 werden Entity Beans mit POJOs implementiert. Ich zeige beide POJOs und kommentiere dann beide zusammen. Erst `Order`:

```
package shopping.entity;  
  
import javax.persistence.CascadeType;  
import javax.persistence.Entity;  
import javax.persistence.FetchType;
```

```

import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.Table;
import java.util.ArrayList;
import java.util.Collection;

@Entity
@Table(name = "PURCHASE_ORDER")
public class Order implements java.io.Serializable
{
    private static final long serialVersionUID = 8940882223672036299L;
    private int id; // PK
    private double total;
    private Collection<LineItem> lineItems;

    @Id(generate = GenerationType.AUTO)
    public int getId()
    {
        return id;
    }

    public void setId(int id)
    {
        this.id = id;
    }

    public double getTotal()
    {
        return total;
    }

    public void setTotal(double total)
    {
        this.total = total;
    }

    public void addPurchase(String product, int quantity, double price)
    {
        if (lineItems == null) lineItems = new ArrayList<LineItem>();
        LineItem item = new LineItem();
        item.setOrder(this);
        item.setProduct(product);
        item.setQuantity(quantity);
        item.setSubtotal(quantity * price);
    }
}

```

```

        lineItems.add(item);
        total += quantity * price;
    }

    @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER, mappedBy="order")
    public Collection<LineItem> getLineItems()
    {
        return lineItems;
    }

    public void setLineItems(Collection<LineItem> lineItems)
    {
        this.lineItems = lineItems;
    }
}

```

Und nun noch LineItem:

```

package shopping.entity;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;

@Entity
public class LineItem implements java.io.Serializable
{
    private static final long serialVersionUID = -2399254359179553429L;
    private int id; // PK
    private double subtotal;
    private int quantity;
    private String product;
    private Order order;

    @Id(generate = GenerationType.AUTO)
    public int getId()
    {
        return id;
    }

    public void setId(int id)
    {
        this.id = id;
    }
}

```

```

public double getSubtotal()
{
    return subtotal;
}

public void setSubtotal(double subtotal)
{
    this.subtotal = subtotal;
}

public int getQuantity()
{
    return quantity;
}

public void setQuantity(int quantity)
{
    this.quantity = quantity;
}

public String getProduct()
{
    return product;
}

public void setProduct(String product)
{
    this.product = product;
}

@ManyToOne
@JoinColumn(name = "order_id")
public Order getOrder()
{
    return order;
}

public void setOrder(Order order)
{
    this.order = order;
}
}

```

*Kommentar:*

1. Ein POJO wird mit der Annotation `@Entity` als Entity Bean bezeichnet. Optional kann man mit einer Annotation `@Table` angeben, wie die Tabelle heißen soll, in der POJOs von diesem Typ persistent gemacht werden, Defaultmäßig werden POJOs mit Namen `foo` in einer Tabelle mit Namen `FOO` gespeichert. Bei unserem Typ `Order` mußten wir die Annotation `@Table` einsetzen, da `ORDER` in SQL ein reserviertes Wort ist; bei dem Typ `LineItem` dagegen konnten wir den Defaultnamen akzeptieren.
2. Ein POJO, das ein Entity Bean implementieren soll, muß das Interface `Serializable` implementieren. Die Konstanten mit Namen

`serialVersionUID`

die ich in beiden POJOs vorgesehen habe, haben nichts mit Entity Beans zu tun; sie sind nur deswegen da, damit Eclipse nicht meckert, daß sie fehlen. In der Javadoc-Ausgabe für `Serializable` steht, daß es *dringend* empfohlen wird, eine solche Konstante zu definieren. Die scheinbar phantasievollen Werte, die ich diesen Konstanten gegeben habe, habe ich von dem Tool `serialver` errechnen lassen.

3. Jedes Entity Bean muß einen eindeutigen Identifikator besitzen, der in der entsprechenden Tabelle als Primärschlüssel dienen kann. Grundsätzlich kann der Programmierer diesen Identifikator so nennen, wie er will. Die einzige Bedingung ist, daß die zugehörige `get`-Methode mit einer `@Id`-Annotation versehen werden sollte. Zusätzlich kann man veranlassen, daß dieser Identifikator/Primärschlüssel automatisch generiert wird; hierzu verwendet man das `generator`-Member der `@Id`-Annotation. Der Wert `AUTO` bedeutet, daß die verwendete Datenbank die für sie natürliche Generierungsmethode verwenden soll.
4. Zwischen unseren Entities `Order` und `LineItem` besteht eine bidirektionale 1-zu-N-Beziehung.

*Bemerkung.* Diese Beziehung müßte nicht zwangsläufig bidirektional sein; es gibt keinen wirklich einleuchtenden Grund dafür, daß jedes `LineItem` den `Order` kennt, zu dem es gehört. Ich habe mich für eine bidirektionale Beziehung entschieden, um zu demonstrieren, wie so etwas gemacht wird. Man könnte die Beziehung relativ leicht in eine unidirektionale umwandeln.

[ejb3] erläutert, daß eine bidirektionale Beziehung sowohl ein *besitzendes* (engl. *owning*) Ende als auch eine *inverses* Ende hat. Das besitzende Ende bestimmt, wann Updates in der Datenbank vorgenommen werden. Dabei gelten folgende Regeln:

- das inverse Ende der Beziehung muß das besitzende Ende referenzieren und zwar mittels des `mappedBy`-Member der `@OneToOne`-, der `@OneToMany`- oder der `@ManyToMany`-Annotation.
- das N-Ende einer 1-zu-N-Beziehung muß das besitzende Ende sein. Darum kann das `mappedBy`-Member nicht in in der `@ManyToOne`-Annotation vorkommen.

- Bei 1-zu-1-Beziehungen entspricht das besitzende Ende der Tabelle, die den entsprechenden Fremdschlüssel enthält.
- bei bidirektionalen N-zu-M-Beziehungen kann jedes der beiden Enden das besitzende Ende sein.

Inspektion unserer beiden POJOs zeigt, daß sie diesen Regeln entsprechen. `LineItem` (das N-Ende) ist das besitzende Ende – obwohl das nach der Semantik unseres Anwendungsbereiches seltsam erscheint. Der POJO `Order` repräsentiert die zu ihm gehörenden `LineItems` mit einem Attribut

```
Collection<LineItem> lineItems;
```

Die zugehörige `get`-Methode bekommt die Annotation

```
@OneToMany(..., mappedBy="order")
```

Der Wert des `mappedBy`-Member ist der Name des Attributs im Besitzer, das das inverse Ende referenziert (in unserem Fall `order`). Die `get`-Methode `getOrder` bekommt selbstverständlich eine `@ManyToOne`-Annotation.

*Bemerkung.* Erst dachte ich, die Annotation `@JoinColumn` sei entweder optional oder etwas JBoss-proprietäres; in der vorläufigen Version von [ejb3] wurde diese Annotation nur in Verbindung mit der Annotation `@SecondaryTable` erwähnt. Anscheinend haben sich die JBoss-Mitglieder der Expertengruppe in diesem Punkt durchsetzen können; im Final Draft von [ejb3] ist `@JoinColumn` vorgesehen.

Nun wollen wir uns der Fassade zuwenden, über die unser Client auf die Entity Beans zugreifen kann (bzw. muß). Wir beginnen mit der Schnittstelle, die ein entfernter Client verwenden darf:

```
package shopping.facade;

import javax.ejb.Remote;
import javax.ejb.Remove;

import shopping.entity.Order;

@Remote
public interface ShoppingCart
{
    void buy(String product, int quantity, double price);

    Order getOrder();

    @Remove void checkout();
}
```

*Kommentar:* Das meiste hier dürfte vom `Converter`-Beispiel her bekannt sein. Neu ist lediglich die Annotation `@Remove`. Diese Annotation hat mit dem Lebenszyklus eines zustandsbehafteten Session Bean zu tun. Die Ausführung einer mit dieser Annotation ausgezeichneten Methode führt dazu, daß das Bean nach Terminierung der Methode (erfolgreich oder fehlerhaft) aus dem Container entfernt wird.

Und nun müssen wir noch die Beanklasse `ShoppingCartBean` kennenlernen, die unsere Schnittstelle `ShoppingCart` implementiert. Sie illustriert einige neuen und interessanten Aspekte:

```
package shopping.facade;

import javax.persistence.*;
import javax.ejb.Remove;
import javax.ejb.Stateful;

import shopping.entity.Order;

@Stateful
public class ShoppingCartBean implements ShoppingCart
{
    @PersistenceContext (unitName="shopping")
    private EntityManager manager;
    private Order order;

    public void buy(String product, int quantity, double price)
    {
        if (order == null) order = new Order();
        order.addPurchase(product, quantity, price);
    }

    public Order getOrder()
    {
        return order;
    }

    @Remove
    public void checkout()
    {
        manager.persist(order);
    }
}
```

*Kommentar:*

1. Als erstes möchte ich hier das Interface `EntityManager` herauspicken. Das neue API `EntityManager` löst ein Problem, das mir sofort in den Sinn sprang, als ich unter den Zielen von JSR 220 las, daß die Home-Interfaces verschwinden sollten. Denn gerade bei den Entity Beans spielten die Home-Interfaces bisher eine herausgehobene Rolle: mit Ihnen konnte man nicht nur neue Instanzen eines Entity Bean erzeugen, sondern man konnte damit auch bereits existente Instanzen suchen. Wie sollte man unter EJB 3.0 Instanzen von Entity Beans suchen?

Die Antwort auf diese Frage lautet: mit dem `EntityManager`. Da dieses API eine so zentrale Rolle in EJB 3.0 spielt, will ich hier einen Teil dieser Schnittstelle wiedergeben:

```
public interface EntityManager
{
    /**
     * Check if the instance belongs to the current
     * persistence context.
     */
    public boolean contains(Object entity);

    /**
     * Make an instance managed and persistent, using the
     * unqualified class name as the entity name.
     */
    public void persist(Object entity);

    /**
     * Merge the state of the given entity into
     * the current persistence context.
     */
    public <T> T merge(T entity);

    /**
     * Remove the instance.
     */
    public void remove(Object entity);

    /**
     * Synchronize the persistence context
     * with the underlying database.
     */
    public void flush();

    /**
     * Find by primary key.
```

```

    */
    public <T> T find(Class<T> entityClass, Object primaryKey);

    /**
     * Create an instance of Query for executing an
     * EJBQL query.
     */
    public Query createQuery(String ejbqlString);

    ...
}

```

Die hier verwendeten Begriffe *managed entity* und *persistence context* werden später noch erklärt.

2. Ich muß eingehen auf das hier benutzte Entwurfsmuster Dependency Injection. Unsere Bean-Klasse `ShoppingCartBean` hätte natürlich eine Instanz der Schnittstelle `EntityManager` erzeugen können – etwa so:

```

private EntityManager manager =
    new org.jboss.ejb3.entity.EntityManagerImpl();

```

Wir sind uns wohl einig, daß das keine besonders gute Idee wäre.

Wir können eine Abhängigkeit unserer Klasse vom Interface `EntityManager` natürlich nicht vermeiden. Aber wir sollten sie nach Möglichkeit nicht auch noch von einer konkreten Implementation dieses Interface abhängig machen, wenn es irgend zu vermeiden ist. Am besten läßt man eine Instanz seiner Klasse (hier `ShoppingCartBean`) mit einer Instanz einer konkreten Implementation des fraglichen Interface durch einen äußeren Agenten “impfen” – z. B. durch ein Framework wie Spring oder durch den Anwendungsserver (in unserem Fall JBoss).

EJB 3.0 unterstützt dieses Entwurfsmuster direkt mit der Annotation

```
@PersistenceContext
```

die wir in `ShoppingCartBean` sehen.

*Bemerkung.* Wenn der Leser mehr über das Entwurfsmuster Dependency Injection erfahren möchte, kann ich den Article [mf] wärmstens empfehlen.

3. Da unsere Entity Beans schlichte POJOs sind, können wir sie ohne große Umstände erzeugen wie in der Methode `buy`. Da sie außerdem noch `Serializable` implementieren, können sie ohne weiteres an einen entfernten Client übergeben werden wie in der Methode `getOrder`. Dabei sollte einem bewußt sein, daß die Kopie, die der Client erhält nicht mit der Kopie im Server synchron bleibt.
4. Unsere `Order`-Beans werden erst in der Datenbank persistent gemacht, wenn unser `ShoppingCart`-Bean in `checkout` sein Abschied nimmt.

## 5 Der Persistenzkontext

JBoss ist so konfiguriert, daß es für die Persistenz defaultmäßig das eingebaute Datenbankmanagementsystem (DBMS) HypersonicSQL verwendet. In einer Produktionsumgebung würde man gern mehrere verschiedene DBMS einsetzen – evtl. für jede Anwendung eine andere. Um diese verschiedenen Datenquellen unterstützen zu können, braucht man für jede einen anderen `EntityManager`.

Diese `EntityManager` haben je einen Namen, mit dem sie voneinander unterschieden werden. Das Attribut `unitName`, das wir in unserer Annotation `PersistenceContext` gesehen haben, hat den Namen des gewünschten `EntityManager` als Wert. JBoss erfährt alles über den zu verwendenden `EntityManager` aus einem sog. *persistence archive*. Das ist eine `jar`-Datei mit Extension `.par` und folgendem Inhalt:

- die `.class`-Dateien aller in diesem Kontext befindlichen Entities (POJOs).
- ein Folder `META-INF` mit einem Deskriptor

```
persistence.xml
```

der den zugehörigen `EntityManager` beschreibt.

*Beispiel.* Der Deskriptor für unser Beispiel könnte dann wie folgt aussehen:

```
<?xml version="1.0" encoding="UTF-8"?>

<entity-manager>
  <name>shopping</name>
  <jta-data-source>java:/DefaultDS</jta-data-source>
  <properties>
    <property name="hibernate.hbm2ddl.auto"
              value="create-drop"/>
  </properties>
</entity-manager>
```

*Kommentar:*

1. Dieses Dokument hat ein Wurzelement mit Namen `entity-manager`.
2. Das Subelement mit Name `name` gibt dem `EntityManager` natürlich einen Namen.
3. Das `jta-data-source`-Element spezifiziert den JNDI-Namen der Datenquelle, die von diesem `EntityManager` verwaltet wird. Hier könnten wir anstelle von

```
java:/DefaultDS
```

z. B.

```
java:/PostgresDS
```

setzen, wenn wir JBoss so konfiguriert hätten, daß es PostgreSQL als Datenquelle einsetzen könnte.

*Bemerkung.* Der Begriff *persistence context*, von dem hier gesprochen wurde, ist nicht jbosspezifisch; wir werden ihm in Abschnitt 7 nochmal begegnen.

## 6 Der Client für unser Beispiel

Und nun können wir einen Blick auf den Client für unsere Anwendung werfen:

```
package shopping.client;

import shopping.entity.LineItem;
import shopping.entity.Order;
import shopping.facade.ShoppingCart;

import javax.naming.InitialContext;

public class Client
{
    public static void main(String[] args) throws Exception
    {
        InitialContext ctx = new InitialContext();
        ShoppingCart cart = (ShoppingCart)
            ctx.lookup(ShoppingCart.class.getName());

        System.out.println("Buying 2 memory sticks");
        cart.buy("Memory stick", 2, 500.00);
        System.out.println("Buying a laptop");
        cart.buy("Laptop", 1, 2000.00);

        System.out.println("Print cart:");
        Order order = cart.getOrder();
        System.out.println("Total: $" + order.getTotal());
        for (LineItem item : order.getLineItems())
        {
            System.out.println(item.getQuantity() +
                "      " + item.getProduct() +
                "      $" + item.getSubtotal());
        }

        System.out.println("Checkout");
    }
}
```

```
        cart.checkout();
    }
}
```

Diese Klasse bedarf wohl keines Kommentars.

## 7 Die Semantik von Entity Beans unter EJB 3.0

Ich finde, daß es der Expertengruppe gelungen ist, die Entwicklung von Entity Beans erheblich zu vereinfachen. Es waren die Deskriptoren, die das Leben des Entwicklers unter EJB 2.1 so schwer machten. Nun fehlen diese gänzlich.

Aber der Volksmund sagt “Umsonst ist nur der Tod”. Irgendwo muß ein versteckter Preis zu zahlen sein. Möglicherweise ist der Preis bei der für mein Empfinden recht komplizierten Semantik der Entity Beans unter EJB 3.0. Aber vielleicht empfinde ich das nur so, weil ich glaubte, die Entity Beans unter EJB 2.1 recht gut zu verstehen.

[ejb3] definiert den Begriff *persistence context*, dem wir bereits in Abschnitt 5 begegnet sind, wie folgt:

A persistence context is a set of managed entity instances, and is managed by the EntityManger. A persistence context is co-extensive with a transaction scope. The set of entities that can be managed by a given EntityManager is defined by a persistence unit.

*A persistence unit defines the set of all classes that are related or grouped by the application and which must be colocated in their mapping to a single database.*

Alles klar? Ich hätte diesen Text ins Deutsche übersetzt, aber dazu hätte ich ihn verstehen müssen! Ich hoffe, der Final Draft bringt hier etwas mehr Klarheit.

Später werden folgende Erklärungen hinzugefügt, die für mein Gefühl verständlicher sind:

A new entity bean instance has no persistent identity, and is not yet associated with a persistence context.

A managed entity bean instance is an instance with a persistent identity that is currently associated with a persistence context.

A detached entity bean instance is an instance with a persistent identity that is not (or no longer) associated with a persistence context.

A removed entity bean instance is an instance with a persistent identity, associated with a persistence context, that is scheduled for removal from the database.

## 8 Nachrichtengesteuerte Komponenten

In unserem ersten EJB-Vortrag haben wir gelernt, daß es drei Sorten von Enterprise Beans gibt:

- Session Beans.
- Entity Beans.
- Message Driven Beans.

von denen wir bisher allerdings nur die ersten beiden erlebt haben. Die Message Driven Beans reagieren, wie der Name nahelegt, auf Nachrichten und nicht auf Methodenaufrufe wie bei den anderen beiden Sorten. So gestatten sie auf optimale Weise die asynchrone Bearbeitung: entweder signalisiert eine Antwortnachricht, daß der Auftrag erledigt ist – oder es gibt gar keine Antwort.

Die Klasse eines Message Driven Bean (MDB) erweitert ein `Listener`-Interface; welches Interface das ist, hängt von der eingesetzten Nachrichtentransporttechnologie ab. Mit JMS ist dieses Interface `MessageListener` – ein Interface, das nur eine einzige Methode besitzt:

```
void onMessage(Message m);
```

Die Message-driven Beans waren schon immer wesentlich einfacher als die anderen beiden Beansorten – vorausgesetzt man beherrscht JMS und das tun wir ja, nicht wahr? Aber sie brauchten immer noch einen deployment descriptor. Unter EJB 3.0 wird dieser Deskriptor durch geeignete Annotationen ersetzt.

Unser drittes Beispiel demonstriert den Gebrauch der Message-driven Beans. Ähnlich wie beim zweiten Beispiel wurde dieses vom JBoss TrailBlazer abgekupfert und leicht nostrifiziert.

Unser MDB berechnet den Ertrag in einem Rentenkonto, wenn die Zinseszinsen monatlich bei fester Verzinsung und einem vereinbarten monatlichen Sparbetrag berechnet werden.

Am interessantesten für uns jetzt ist das MDB `CalculatorBean`, das wir jetzt anschauen wollen:

```
package savings;

import javax.ejb.*;
import javax.jms.*;
import java.sql.Timestamp;

@MessageDriven(activateConfig =
{
    @ActivationConfigProperty(propertyName = "destinationType",
        propertyValue = "javax.jms.Queue"),
```

```

        @ActivationConfigProperty(propertyName = "destination",
                                propertyValue = "queue/savings") })
public class CalculatorBean implements MessageListener
{
    public void onMessage(Message msg)
    {
        try
        {
            ObjectMessage omsg = (ObjectMessage) msg;
            Timestamp sent = new Timestamp(omsg.getLongProperty("sent"));
            MessageBean bean = (MessageBean) omsg.getObject();

            int start = bean.getStart();
            int end = bean.getEnd();
            double growthrate = bean.getGrowthrate();
            double saving = bean.getSaving();

            // Pause to simulate a long running task
            Thread.sleep(1000);

            double result = calculate(start, end, growthrate, saving);
            RecordManager.addRecord(sent, result);

            System.out.println("onMessage() called");
        }
        catch (Exception e)
        {
            System.err.println(e);
        }
    }

    private double calculate(int start, int end, double growthrate,
                             double saving)
    {
        double tmp = Math
            .pow(1.0 + growthrate / 12.0, 12.0 * (end - start) + 1);
        return saving * 12.0 * (tmp - 1) / growthrate;
    }
}

```

*Kommentar:*

1. Genau wie unter EJB 2.1 muß unsere MDB-Klasse das Interface `MessageListener` sowie dessen einzige Methode `onMessage` implementieren.

2. Unter EJB 2.1 mußte ein MDB immer noch in einem deployment descriptor beschrieben werden: von welcher Art ist der Nachrichtenendpunkt, von dem das MDB seine Nachrichten bezieht, bzw. wie heißt der JNDI-Name dieses Endpunkts, etc. Unter EJB 3.0 kann man auf den Deskriptor verzichten; stattdessen wird die MDB-Klasse mit der Annotation `@MessageDriven` versehen. Die `@MessageDriven`-Annotation hat ein Element `activateConfig`, dessen Wert eine kommasetrennte Liste von `@ActivationConfigProperty`-Annotationen ist. Diese Annotation wiederum besitzt zwei Elemente `propertyName` und `propertyValue`.

Auf diese Art werden für unser MDB `CalculatorBean` die nötigen Informationen spezifiziert: es bezieht seine Nachrichten von einer `Queue` mit JNDI-Namen `queue/savings`.

3. Wie man im Code unserer Methode `onMessage` sieht, geht unser MDB davon aus, daß die Nachrichten, die es erhält, folgende Eigenschaften besitzen:
  - sie haben ein anwendungsspezifisches Attribut `sent` vom Typ `long`, dessen Wert die Zeit ist, zu der die Nachricht abgeschickt wurde.
  - die Nachricht ist vom Typ `ObjectMessage` und dessen Inhalt ist ein Java-Bean `MessageBean` und dessen Attribute sind die Argumente für die Methode, die den Ertrag des Rentenkontos berechnet.

Diese Argumente sind:

- Alter des Sparers zu der Zeit, als er mit Sparen beginnt (`start`).
- Alter, in dem der Sparer in Rente gehen will (`end`).
- Festzinsrate (`growthrate`).
- Betrag, der monatlich gespart werden soll (`saving`).

Rein theoretisch müßte es möglich sein, daß unser MDB das Ergebnis seiner Berechnung mit einer Antwortnachricht an den Client übermittelt.

Nur weiß ich leider nicht, wie das geht. Die Autoren des TrailBlazer-Beispiels ziehen einen anderen Weg vor, der auf den ersten Blick so aussieht, als ob sie hinten durch die Brust ins Auge wollten.

Das MDB legt das Ergebnis seiner Berechnung in einem zentralen Cache ab, von wo der Client es abholen kann sobald es dort ankommt. Diese scheinbar überkandidelte Lösung hat den Vorzug, daß Client und Berechnungsdienst (MDB) weiter entkoppelt werden.

Der Cacheverwalter ist ein Objekt vom Typ `RecordManager` und hat eine Methode

```
public static void addRecord(Timestamp sent, double result)
```

die unsere Methode `onMessage` oben verwendet, um ihr Ergebnis abzulegen. Dabei ist das erste Argument `sent` der Schlüssel, mit dem Datensätze im Cache gesucht werden können. Als Wert dieses Schlüssels nehmen wir den Wert des anwendungsspezifischen Attributs `sent`, das mit dem Auftragsnachricht verbunden war.

*Bemerkung.* Diese Lösung funktioniert nur so lange gut, wie das Verkehrsvolumen für unseren Berechnungsdienst relativ klein bleibt. Sollte diese Lösung anfangen, problematisch zu werden, könnte man als Schlüssel den JMS-definierten Headereintrag `JMSMessageID` aus der Auftragsnachricht einsetzen.

Das Frontend (unser Client, wenn man so will) besteht aus zwei JavaServer Pages (JSP):

- `calculator.jsp`: diese Seite konstruiert das Objekt vom Typ `MessageBean` und schickt es als `ObjectMessage` an den Nachrichtenendpunkt.
- `check.jsp`: diese Seite bekommt den Schlüssel `sent` von `calculator.jsp` übermittelt und wartet so lange bis ein zu diesem Schlüssel passender Datensatz im Cache vorhanden ist. Dann stellt sie das Ergebnis dar.

## 9 Schlußbemerkungen

In den Übungen zur Vorlesung über EJB, die ich im vergangenen Wintersemester hielt, habe ich das Leben der Übungsteilnehmer mit folgenden Werkzeugen vergleichsweise einfach gemacht:

- XDoclet
- JBoss-IDE

XDoclet ist (verständlicherweise) noch nicht an EJB 3.0 angepaßt worden. JBoss-IDE ist angeblich an EJB 3.0 angepaßt worden, aber ich habe den Eindruck, daß dieses Plugin noch eine Baustelle ist. So mußte ich bei meinen Versuchen mit EJB 3.0 auf diese beiden angenehmen Werkzeuge verzichten.

EJB 3.0 ist jedoch so viel einfacher als EJB 2.1, daß ich diese Hilfsmittel kaum vermißt habe. Ich finde, daß die Ziele, die sich die JSR220-Expertengruppe gesetzt hat, bereits weitgehend erreicht sind.

Die Probeversion der JBoss-Gruppe ist sicherlich ein wichtiger Beitrag zu diesem Unternehmen. Probleme, die die Expertengruppe evtl. übersehen hat, können nun leichter aufgedeckt und gemeldet werden.

Andererseits muß man feststellen, daß es seine Tücken hat, mit einer experimentellen Version einer Software zu arbeiten, die eine Spezifikation implementiert, die ihrerseits noch nicht in endgültiger Form vorliegt. Bei jedem neuen Release dieser Software kann es sein, daß sich vieles geändert hat, so daß der eigene Code neu angepaßt werden muß. Wer jedoch starke Nerven hat, kann es aufregend finden, an einer solchen Entwicklung beteiligt zu sein – und sei es nur als Anwender.

## Literatur

[ejb3] <http://www.jcp.org/en/jsr/detail?id=220>. Hier findet man die jeweils aktuellen JSR220-Dokumente.

- [mf] <http://www.martinfowler.com/articles/injection.html>. Ein sehr guter Artikel zum Thema “Dependency Injection”.
- [tb] <http://www.jboss.com/docs/demos>. Hier findet man die jeweils aktuelle Dokumentation zur EJB-3.0-Entwicklung unter JBoss.
- [or] <http://www.oracle.com/technology/tech/java/ejb30.html> Hier findet man alles über die neue Oracle-Implementation von EJB 3.0.