Workshop Java EE 7

Ein praktischer Einstieg in die Java Enterprise Edition mit dem Web Profile

von
Marcus Schießer, Martin Schmollinger

2., akt. u. erw. Aufl.

dpunkt.verlag 2014

Verlag C.H. Beck im Internet:
www.beck.de
ISBN 978 3 86490 195 9

Zu Inhaltsverzeichnis

schnell und portofrei erhältlich bei beck-shop.de DIE FACHBUCHHANDLUNG

8.3 Transaktionssteuerung

Im vorherigen Kapitel über JPA haben wir in Abschnitt 7.4.2 bereits erklärt, was eine Transaktion ist. Außerdem wurde die Transaktionssteuerung von EJBs genutzt, indem wir unsere Services in EJBs umgewandelt haben. Dies war äußerst einfach: Wir mussten lediglich die Annotation `@RequestScoped` durch die Annotation `@Stateless` austauschen. Dadurch nutzten wir das Standardverhalten von EJBs, wobei ein Methodenaufruf eine neue Transaktion startet und bei Beendung der Methode die Transaktion ebenfalls beendet wird. Wirft die Methode eine Ausnahme, so wird die Transaktion zurückgerollt, andernfalls werden die Daten in der Datenbank nach Methodenende festgeschrieben. Bei dieser Standardkonfiguration kümmert sich der EJB-Container um die Transaktionssteuerung. Der Container unterstützt dabei verschiedene Strategien, die im folgenden Abschnitt erläutert werden.

Alternativ kann sich der Entwickler auch selbst programmatisch um die Transaktionssteuerung kümmern; diese Variante wird in Abschnitt 8.3.2 näher behandelt.

8.3.1 Transaktionssteuerung durch den Container

Standardmäßig kümmert sich der EJB-Container um die Transaktionssteuerung. In der englischen Literatur spricht man daher auch von *Container-Managed Transactions* (CMT). Der Container unterstützt dabei verschiedene Strategien für die Steuerung der Transaktion, die sich hauptsächlich damit beschäftigen, was passieren soll, wenn bereits eine Transaktion zum Zeitpunkt des Methodenaufrufs existiert. Diese Situation kann beispielsweise auftreten, wenn eine EJB eine weitere aufruft. Die verschiedenen Strategien werden in Tabelle 8–1 erläutert und können pro Methodenaufruf festgelegt werden.

Transaktionsstrategie	Erklärung
`REQUIRED`	Existiert bereits eine Transaktion, so wird diese von der aufzurufenden Methode verwendet. Ansonsten wird bei Methodenstart eine neue Transaktion angelegt und diese bei Beendigung der Methode beendet.
`MANDATORY`	Zur Ausführung der Methode ist eine Transaktion erforderlich (engl. *mandatory*), es wird jedoch keine neue Transaktion gestartet. Wird die Methode aufgerufen, ohne dass eine Transaktion aktiv ist, wird eine Ausnahme geworfen.
`REQUIRES_NEW`	Für die Ausführung der Methode ist eine neue Transaktion erforderlich, die nach Beendigung der Methode beendet wird. Existiert bereits eine Transaktion vor Aufruf der Methode, so wird diese angehalten und nach Methodenende wieder aktiviert. EJB unterstützt keine verschachtelten Transaktionen!

→

Transaktionsstrategie	Erklärung
`SUPPORTS`	Ist bereits eine Transaktion aktiv, so wird diese verwendet. Falls nicht, so wird aber auch keine neue Transaktion gestartet. Existiert bereits eine Transaktion, so kann die neue Methode diese auch durch das Werfen einer Ausnahme zurückrollen.
`NOT_SUPPORTED`	Ist bereits eine Transaktion aktiv, so wird diese angehalten und nach Methodenende weitergeführt. Wirft die aktuelle Methode eine Ausnahme, so wird die angehaltene Transaktion nicht zurückgerollt.
`NEVER`	Eine Methode, die diese Strategie verwendet, läuft nie in einer Transaktion. Ist bereits eine Transaktion aktiv, so wird eine Ausnahme geworfen. Es wird keine neue Transaktion gestartet.

Tab. 8–1 *Strategien für die Container-Transaktionssteuerung*

Wenn Sie die Erklärung für die Strategie *REQUIRED* lesen, wird sie Ihnen bekannt vorkommen: Es handelt sich um die Strategie, die von unseren Services verwendet wird. Dies liegt daran, dass EJB 3.2 per Konvention diese Strategie für Methodenaufrufe verwendet, ohne dass sie explizit festgelegt werden muss.

Wollen wir eine andere Strategie für unsere Methode verwenden, so müssen wir diese mit der Annotation `@TransactionAttribute` aus dem Paket `javax.ejb` versehen und die gewünschte Strategie als Parameter angeben. Die Strategie wird über den Aufzählungstyp `@TransactionAttributeType` aus demselben Paket festgelegt. Dieser Aufzählungstyp hat als mögliche Werte die Namen der Strategien aus Tabelle 8–1. Möchte man beispielsweise für eine Methode die Strategie *MANDATORY* festlegen, so muss man sie mit folgender Annotation versehen:

```
@TransactionAttribute(TransactionAttributeType.MANDATORY)
```

Alternativ kann man auch einer ganzen Klasse eine solche Annotation geben, dann gilt die gewählte Strategie für alle Methoden der Klasse.

Zurückrollen einer Transaktion

Wir hatten bereits erwähnt, dass eine Transaktion beim Auftreten einer Ausnahme zurückgerollt wird. Genau genommen muss es sich dabei um eine Ausnahme des Typs `RuntimeException` handeln. Dies sind die Ausnahmen, die während der Laufzeit auftreten und nicht explizit durch einen `try-catch`-Block behandelt werden müssen.

Wenn jedoch eine überprüfte Ausnahme (engl. *checked exception*) auftritt, also eine, die vom Typ `Exception` abgeleitet ist und explizit vom Programmcode behandelt werden muss, so wird die Transaktion nicht automatisch zurückgerollt. Möchte der Entwickler dennoch die Transaktion zurückrollen, so kann er in dem `catch`-Block der Ausnahmebehandlung eine `EJBException` werfen. Da diese vom Typ `RuntimeException` abgeleitet ist, wird die Transaktion in diesem Fall ebenfalls zurückgerollt.

Alternativ kann der Entwickler auch die Methode `setRollbackOnly` der Klasse `SessionContext` aufrufen. Dadurch wird der Anwendungsserver angewiesen, die Transaktion zurückzurollen.

Unterschiedliche Transaktionsstrategien einsetzen

Um den Einsatz von verschiedenen Strategien besser zu verstehen, gehen wir beispielhaft ein Entwurfsmuster durch, das in der Praxis oft zum Einsatz kommt.

Das Sequenzdiagramm in Abbildung 8–3 zeigt hierzu eine mit `REQUIRED` annotierte `TransactionBean`, wodurch deren Methoden ausnahmslos diese Transaktionsstrategie einsetzen. Diese Bean kümmert sich ausschließlich um die Transaktionssteuerung, während die eigentliche Geschäftslogik durch die `BusinessBean` implementiert wird, die wiederum mit `MANDATORY` annotiert ist. Dadurch stellt der Anwendungsserver sicher, dass die Methoden der Bean in einer Transaktion laufen – ist dies nicht der Fall, wird eine Ausnahme geworfen.

In dem Beispiel gehen wir davon aus, dass im Kontext des *Methodenaufrufers* keine Transaktion existiert. Durch den Aufruf der Methode `doBusiness` der `TransactionBean` wird daher eine neue Transaktion gestartet. Diese wiederum delegiert die eigentliche Aufgabe an die Methode `doBusinessDelegate` der `BusinessBean`. Sobald die Methode beendet wurde, übernimmt die `TransactionBean` wieder die Ausführung, die die laufende Transaktion beendet.

Würde der *Methodenaufrufer* hingegen direkt eine Methode der `BusinessBean` starten, so würde der Anwendungsserver eine Ausnahme werfen. Durch dieses Entwurfsmuster wird für die Businessmethoden sichergestellt, dass sie in einer Transaktion laufen, sie müssen sich allerdings nicht um die eigentliche Transaktionssteuerung kümmern. Diese Aufgabe wird von der vorgelagerten `TransactionBean` übernommen. Dieses Vorgehen hat den Vorteil, dass man die `BusinessBean` nicht anpassen muss, um die Transaktionssteuerung zu ändern.

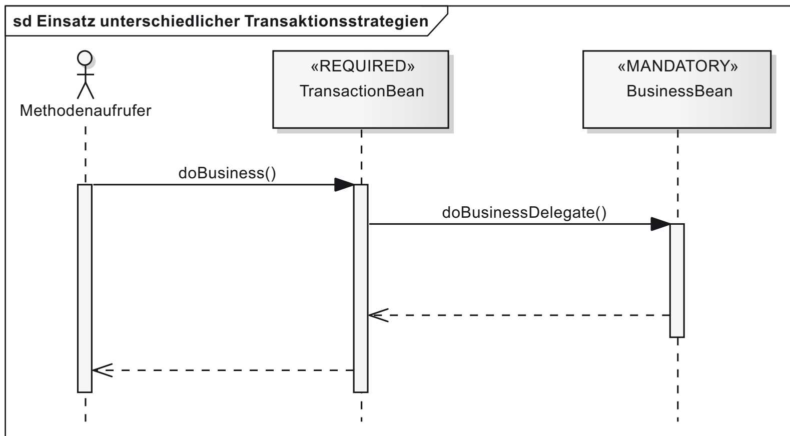

Abb. 8–3 *Beispiel für den Einsatz unterschiedlicher Transaktionsstrategien*

In unserem Beispiel verzichten wir der Einfachheit halber auf dieses Entwurfsmuster: Unsere Services implementieren die Geschäftslogik und kümmern sich ebenso um die Transaktionssteuerung. Sie sind daher mit keiner zusätzlichen Annotation versehen und verwenden dadurch die Standardstrategie *REQUIRED*.

8.3.2 Transaktionssteuerung über die Bean

Neben der Möglichkeit, die Transaktionssteuerung dem Container zu überlassen, kann man sich als Entwickler alternativ auch selbst um die Transaktionssteuerung kümmern. In diesem Fall spricht man in der englischen Fachliteratur von einer *Bean-Managed Transaction* (BMT), also davon, dass die Transaktionssteuerung von der Bean übernommen wird.

Da per Konvention die Transaktionssteuerung jedoch Aufgabe des Containers ist, muss die Bean zunächst mit folgender Annotation versehen werden:

```
@TransactionManagement(TransactionManagementType.BEAN)
```

Dadurch teilen wir dem Anwendungsserver mit, dass wir uns selbst um die Transaktionssteuerung kümmern wollen. Beide Typen (`TransactionManagement` und `TransactionManagementType`) befinden sich im Paket `javax.ejb`.

Führen Sie diese Änderung nun beispielhaft für die `CampaignServiceBean` durch. Wenn wir anschließend die Anwendung neu deployen und starten, erscheint beim Hinzufügen einer Aktion eine `javax.persistence.TransactionRequiredException`. Diese Ausnahme hatten wir bereits in Abschnitt 7.4.2 kennengelernt. Um sie zu beheben, hatten wir den Service in eine EJB umgewandelt, für die sich per Konvention der Container um die Transaktionssteuerung kümmert (siehe Abschnitt 8.3.1). Nachdem wir dem Anwendungsserver über obige Annotation jedoch mitgeteilt haben, dass wir selbst für die Transaktionssteuerung sorgen möchten, jedoch noch keine Transaktion gestartet haben, erscheint bei einer Datenbankmanipulation konsequenterweise wieder diese Ausnahme.

Um die Transaktion steuern zu können, benötigen wir ein Objekt, über das wir eine Transaktion starten und beenden können. Die in Java EE 7 enthaltene *Java Transaction API* (JTA) 1.2 liefert hierzu in dem Package `javax.transaction` das Interface `UserTransaction`. Wir können daher das Interface `UserTransaction` aus dem Paket `javax.transaction` in unseren Beans injizieren und verwenden. In unserem Beispiel fügen wir es der `CampaignServiceBean` hinzu:

```
@Resource
private UserTransaction userTransaction;
```

Anschließend kann die Transaktion über dieses Interface in der Bean gesteuert werden. Natürlich steht hinter dem Interface eine konkrete Implementierung, die vom Anwendungsserver zur Verfügung gestellt wird. Diese Implementierungsdetails sind für unsere Zwecke jedoch ohne Bedeutung.

Verwenden wir das Interface nun, indem wir die Methode `addCampaign` der `CampaignServiceBean` um die fett dargestellten Anweisungen aus Listing 8–12 erweitern. Man sieht sehr schön, dass die drei ursprünglichen Anweisungen zum Hinzufügen einer Aktion durch eine Klammer von Anweisungen eingeschlossen werden, die sich um die Transaktionssteuerung kümmern. Relevant sind dabei insbesondere die Methoden `begin` und `commit` der `UserTransaction`, die eine Trans-

aktion starten bzw. in der Datenbank festschreiben (engl. *commit*) und dadurch beenden.

Der restliche Quelltext kümmert sich ausschließlich um den Fehlerfall, wenn eine Ausnahme geworfen wird. Dann wird versucht, die Transaktion über die Methode `rollback` zurückzurollen. Schlägt auch dies fehl, so wird dies in der Fehlerausgabe geloggt.

```
public void addCampaign(Campaign campaign) {
   try {
      userTransaction.begin();
      Organizer organizer = getLoggedinOrganizer();
      campaign.setOrganizer(organizer);
      entityManager.persist(campaign);
      userTransaction.commit();
   } catch (Exception e) {
      try {
         userTransaction.rollback();
         System.err.println("addCampaign - Transaktion wurde
         zurückgerollt. Aktion: " + campaign.getName());
      } catch (Exception e2) {
         System.err.println("addCampaign - Fehler beim
         Zurückrollen von Transaktion. Aktion: " + campaign.getName());
      }
   }
}
```

Listing 8–12 *Implementierung der Methode addCampaign mit eigener Transaktionssteuerung*

Nun haben wir auch die zweite Möglichkeit der Transaktionssteuerung durch die Bean kennengelernt. Wie Sie sehen, ist diese Art komplexer, als wenn man sie dem Container überlässt. Sie sollte daher nur verwendet werden, wenn die vom Container unterstützten Strategien nicht eingesetzt werden können.

8.3.3 Transaktionssteuerung über Interzeptoren

Im vorherigen Abschnitt hatten wir gelernt, wie man die Transaktionssteuerung nicht dem Container überlässt, sondern diese stattdessen selbst programmiert. Hierzu haben wir die Methode `addCampaign` der `CampaignServiceBean` als Beispiel um die fettgedruckten Anweisungen aus Listing 8–12 erweitert.

Bei der Transaktionssteuerung handelt es sich jedoch üblicherweise um einen Aspekt, den man einer Vielzahl von Methoden hinzufügen möchte, nicht nur einer einzelnen Methode wie `addCampaign`. Dies bedeutet, dass wir die fettgedruckten Anweisungen zu jeder Methode, die transaktionsgesteuert ablaufen soll, hinzufügen müssten. Dies wäre jedoch ein äußerst fehleranfälliges Unternehmen, das bei einer Änderung der Transaktionssteuerung zu Änderungen in jeder transaktionalen Methode führt. Diese Wiederholungen können – noch schlimmer – zu Inkonsistenzen führen, wenn man die Anweisungen in allen Methoden nicht

gleichartig ändert. Hier gilt wie immer die wichtige Programmierregel DRY (Don't repeat yourself) – auf Deutsch: Wiederhole dich nicht![9]

Unklar ist jedoch, wie man eine Wiederholung vermeiden kann, schließlich findet die Transaktionssteuerung vor und nach der eigentlichen fachlichen Logik (den nicht fettgedruckten Anweisungen der Methode `addCampaign`) statt, und Java SE liefert hierfür keine einfache Lösung.

Java EE 7 bietet daher das Konzept der Interzeptoren. Es handelt sich dabei um eine Klasse, die eine mit `@AroundInvoke` (aus dem Paket `javax.interceptor`) annotierte Methode enthält.

Ein solcher Interzeptor kann über eine Annotation einer bereits bestehenden Methode (aus einer anderen Klasse) zugewiesen werden. Wird eine derart annotierte Methode aufgerufen, wird zunächst die Methode des Interzeptors ausgeführt. Innerhalb des Interzeptors kann an beliebiger Stelle die Ausführung an die ursprünglich aufgerufene Methode delegiert werden. Hierzu muss die Methode `proceed` der Klasse `InvocationContext` (Paket `javax.interceptor`) aufgerufen werden.

Durch dieses Konzept ist es möglich, durch eine Annotation eine Methode um Funktionalitäten zu erweitern, die vor und nach der Methode ausgeführt werden: Genau das, was wir für unsere Transaktionssteuerung benötigen!

Listing 8–13 zeigt die Implementierung eines Interzeptors, der sich um die Transaktionssteuerung kümmert. Speichern Sie die Datei bitte im Verzeichnis `src\main\java\de\dpunkt\myaktion\util` ab.

```
package de.dpunkt.myaktion.util;

import javax.annotation.Resource;
import javax.interceptor.AroundInvoke;
import javax.interceptor.InvocationContext;
import javax.transaction.UserTransaction;

public class TransactionInterceptor {

   @Resource
   private UserTransaction transaction;

   @AroundInvoke
   public Object doTransaction(InvocationContext ctx) throws Exception {
      try {
         transaction.begin();
         Object ret = ctx.proceed();
         transaction.commit();
         return ret;
      } catch (Exception e) {
         try {
            transaction.rollback();
            System.err.println("addCampaign –
                                    Transaktion wurde zurückgerollt.");
```

9. http://de.wikipedia.org/wiki/Don%E2%80%99t_repeat_yourself

```
            } catch (Exception e2) {
                System.err.println("addCampaign –
                                    Fehler beim Zurückrollen von Transaktion.");
            }
            throw e;
        }
    }
}
```

Listing 8–13 *Klasse TransactionInterceptor*

Vergleichen Sie anschließend die fettgedruckten Zeilen von Listing 8–12 und 8–13 – Sie werden feststellen, dass diese identisch sind. Dies ist kein Zufall: Der `TransactionInterceptor` enthält wie gewünscht die komplette Logik der Transaktionssteuerung. Anstelle der fachlichen Logik der Methode `addCampaign` ruft der Interzeptor im Vergleich jedoch die Methode `proceed` des `InvocationContext` auf. Der Container delegiert dadurch wiederum an die eigentliche Methode, die mit dem `TransactionInterceptor` annotiert wurde. Das Sequenzdiagramm aus Abbildung 8–4 verdeutlicht diesen Ablauf.

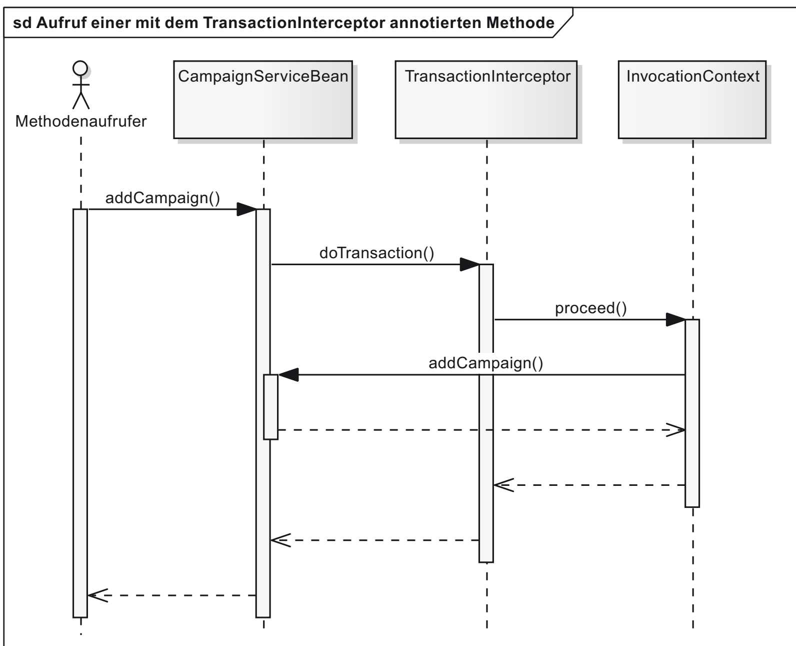

Abb. 8–4 *Sequenzdiagramm des Aufrufs einer mit dem TransactionInterceptor annotierten Methode*

Entfernen Sie nun die fettgedruckten Anweisungen aus Listing 8–12 aus der Methode `addCampaign` und löschen Sie ebenfalls das dadurch nicht mehr benötigte Objekt `UserTransaction` aus der Klasse `CampaignServiceBean`. Ein Neustart würde wieder zu einer `javax.persistence.TransactionRequiredException` führen. Um dies

zu vermeiden, weisen wir der gesamten Klasse `CampaignServiceBean` den `TransactionInterceptor` durch folgende Annotation zu:

```
@Interceptors(TransactionInterceptor.class)
```

Dadurch erfolgt der Aufruf aller Methoden der Klasse über den `TransactionInterceptor`. Bitte überprüfen Sie die Funktionalität der Anwendung nun durch erneutes Deployen und Starten.

Durch die Einführung der Klasse `TransactionInterceptor` konnten wir einen technischen Aspekt (die Transaktionssteuerung) von der eigentlichen fachlichen Logik trennen und dadurch die Codequalität erheblich verbessern.

Erwähnenswert ist auch noch, dass man Interzeptoren auch mit CDI-Beans verwenden kann. Außerdem unterstützen CDI-Beans auch das Injizieren einer `UserTransaction`. Dies bedeutet, dass der `TransactionInterceptor` auch mit CDI-Beans funktioniert.

Da die Transaktionssteuerung unserer Anwendung jedoch vom Container übernommen werden kann (siehe Abschnitt 8.3.1), diente dieser Abschnitt ausschließlich dazu, Ihnen das Konzept von Interzeptoren zu erklären. Entfernen Sie daher die folgenden Annotationen aus der Klasse `CampaignServiceBean`:

```
@TransactionManagement(TransactionManagementType.BEAN)
@Interceptors(TransactionInterceptor.class)
```

Durch diese Änderung steuert der Container wieder ausschließlich unsere Transaktionen.

In Java EE 7 ist die Annotation `@Transactional` aus dem Paket `javax.transaction.cdi` neu hinzugekommen. Über diese kann analog zu unserem `TransactionInterceptor` ebenfalls eine Methode einer CDI-Bean annotiert werden, für die eine Transaktionssteuerung gewünscht ist. Im Gegensatz zu unserem einfach gehaltenen `TransactionInterceptor` unterstützt dieses Feature jedoch alle Transaktionsstrategien aus Abschnitt 8.3.1. Standardmäßig wird dabei wieder die Strategie *REQUIRED* verwendet. Um eine andere einzusetzen, muss die gewünschte Transaktionsstrategie der Annotation `@Transactional` als Parameter übergeben werden. Um beispielsweise die Strategie *MANDATORY* zu verwenden, müssen Sie die Methode folgendermaßen annotieren:

```
@Transactional(TxType.MANDATORY)
```

Interessant ist weiterhin, dass dieses Feature ebenfalls auf CDI-Beans angewendet werden kann. Dies legt die Vermutung nahe, dass schrittweise Aspekte der EJBs in Annotationen ausgelagert werden, um diese CDI-Beans ebenfalls zur Verfügung zu stellen. Im Ergebnis wären EJBs als Komponentenmodell obsolet und könnten komplett durch CDI ersetzt werden.

Momentan ist dies mit Standardmitteln allerdings noch nicht möglich: CDI-Beans unterstützen nicht die Annotationen zur Absicherung einer Methode aus Abschnitt 8.2.5. Zwar könnte man diese ebenfalls über selbst zu entwickelnde

Interzeptoren abbilden, aber dann würden Sie einerseits nichts über EJBs lernen und andererseits müssten Sie auf einige weiterhin bestehende Vorteile von EJBs verzichten (siehe Abschnitt 8.5).

8.4 Zeitgesteuerte Abläufe realisieren

Für den Anwendungsfall *Geld spenden* (siehe Abschnitt 3.4.9) haben wir einen Hintergrundjob vorgesehen, der alle 5 Minuten noch nicht bearbeitete Spenden überweisen soll. Zu dieser Anforderung passt hervorragend die Möglichkeit, durch den *EJB Timer Service* Methoden zeitgesteuert aufrufen zu lassen.

Im folgenden Abschnitt behandeln wir die Methode zur Überweisung von Spenden; Abschnitt 8.4.2 befasst sich dann mit dem zeitlich gesteuerten Aufruf dieser Methode.

8.4.1 Spenden überweisen

In unserer Beispielanwendung können wir leider nicht wirkliche Überweisungen durchführen, da wir keine Schnittstelle zu einem Banksystem besitzen. Trotzdem wollen wir zumindest die Durchführung von Überweisungen simulieren. Dazu erweitern wir zunächst die Schnittstelle `DonationService` um die Methode `transferDonations`. Fügen Sie hierzu die folgende Zeile der Datei `DonationService.java` hinzu:

```
void transferDonations();
```

Dies hat zur Folge, dass wir in der `DonationServiceBean` diese Methode nun auch implementieren müssen. Bevor wir dies jedoch tun, benötigen wir noch eine *NamedQuery*, über die wir diejenigen `Donation`-Objekte abfragen können, die sich in einem bestimmten Status befinden. Dadurch können wir in der Methode `transferDonations` jene `Donation`-Objekte selektieren, die sich im Status `IN_PROCESS` befinden und daher noch überwiesen werden müssen.

Ändern Sie nun den Anfang der Klasse `Donation` folgendermaßen ab, um die *NamedQuery* hinzuzufügen:

```
@NamedQueries({
   @NamedQuery(name = Donation.findByStatus, query =
      "SELECT d FROM Donation d WHERE d.status = :status")
})
@Entity
public class Donation {
   public static final String findByStatus = "Donation.findByStatus";
```