HANSER

Leseprobe

Craig Walls

Spring im Einsatz

ISBN: 978-3-446-42388-6

Weitere Informationen oder Bestellungen unter

http://www.hanser.de/978-3-446-42388-6

sowie im Buchhandel.

6 Transaktionen verwalten

Dieses Kapitel behandelt die folgenden Themen:

- Integration mit Transaktionsmanagern
- Programmgesteuertes Verwalten von Transaktionen
- Verwenden deklarativer Transaktionen
- Beschreiben von Transaktionen mit Annotationen

Lehnen Sie sich entspannt zurück, und denken Sie an Ihre Kindheit. Wahrscheinlich verbrachten Sie viele sorglose Stunden auf dem Spielplatz und auf der Schaukel, kletterten das Klettergerüst hoch und genossen das Kribbeln im Bauch bei der Fahrt mit dem Karussell und auf der Wippe.

Das Problem mit der Wippe besteht darin, dass es nicht möglich ist, alleine Spaß zu haben. Ganz klar: Um zu wippen, benötigen Sie eine zweite Person. Nun haben Sie und Ihr Freund oder Ihre Freundin beschlossen, zu wippen. Diese Übereinkunft ist ein Alles-oder-nichts-Projekt: Entweder, Sie wippen beide – oder keiner von Ihnen beiden. Wenn einer seinen Platz am anderen Ende der Wippe nicht einnimmt, findet kein Wippen statt, sondern es hockt nur ein trauriges Kind da, am Ende eines schrägen Hebels.

In der Software bezeichnet man Alles-oder-nichts-Operationen als *Transaktionen*. Mit Transaktionen können Sie mehrere Operationen zu einer Einheit zusammenfassen, die entweder vollständig oder gar nicht stattfindet. Wenn alles einwandfrei funktioniert, war die Transaktion erfolgreich. Schlägt aber irgendetwas fehl, dann wird alles rückgängig gemacht, so als ob nie irgendetwas geschehen wäre.

Das wohl gängigste Beispiel einer Transaktion in der Praxis ist die Geldüberweisung. Nehmen wir an, Sie wollen 100 € von Ihrem Spar- auf Ihr Girokonto überweisen. Dieser Vorgang umfasst zwei Operationen: 100 € werden von Ihrem Sparkonto abgezogen, und 100 € Ihrem Girokonto gutgeschrieben. Die Überweisung muss vollständig erfolgen, sonst darf sie gar nicht erfolgen. Funktioniert das Abziehen des Betrags von Ihrem Sparkonto, während die Gutschrift auf das Girokonto fehlschlägt, dann haben Sie 100 € weniger – schlecht für Sie, gut für die Bank. Schlägt umgekehrt das Abziehen fehl, während die Gutschrift erfolgt, dann haben Sie 100 € mehr – gut für Sie, schlecht für die Bank. Zufrieden werden beide Beteiligte sein, wenn die gesamte Überweisung rückgängig gemacht wird und wenn eine der Operationen fehlschlägt.

Im vorigen Kapitel haben wir den Spring-Support für den Datenzugriff kennengelernt und mehrere Möglichkeiten beschrieben, Daten in die Datenbank zu schreiben und aus ihr zu lesen. Wenn Sie in die Datenbank schreiben, müssen Sie sicherstellen, dass die Integration der Daten erhalten bleibt. Zu diesem Zweck müssen die Änderungen innerhalb einer Transaktion erfolgen. Spring bietet umfangreichen Support für die programmgesteuerte wie auch für

deklarative Transaktionsverwaltung. In diesem Kapitel erfahren wir, wie man Transaktionen im Anwendungscode so durchführt, dass sie, wenn alles klappt, abgeschlossen werden. Und wenn irgendetwas schiefgeht ... nun ja, das muss doch niemand zu wissen. (Genauer gesagt, beinahe niemand. Sie sollten es aus Gründen der Überwachung nicht versäumen, das Problem in eine Logdatei schreiben zu lassen.)

6.1 Grundlagen zu Transaktionen

Um Transaktionen besser erklären zu können, wollen wir exemplarisch den Kauf einer Kinokarte beschreiben. Der Kauf einer solchen Karte umfasst folgende Vorgänge:

- Die Anzahl der verfügbaren Plätze wird geprüft, um festzustellen, ob noch genügend Plätze für Ihre Bestellung vorhanden sind.
- Die Anzahl der vorhandenen Plätze wird pro gekaufter Karte um 1 verringert.
- Sie bezahlen die Karte.
- Die Karte wird Ihnen übergeben.

Klappt alles, dann wünscht Ihnen Ihr Lichtspielhaus – nun um ein paar Euro reicher – einen angenehmen Abend in Ihrem Film. Was aber, wenn irgendein Schritt misslingt? Angenommen, Sie wollen per Kreditkarte zahlen, haben Ihren Verfügungsrahmen aber bereits überschritten? In diesem Fall bekommen Sie ganz sicher keine Karte – und das Kino kein Geld. Wenn dann die Anzahl der freien Plätze nicht auf den Wert vor der Bestellung zurückgesetzt wird, könnte irgendwann fälschlicherweise der Eindruck entstehen, die Vorstellung sei ausverkauft (was dem Kino finanzielle Einbußen bringt). Ein anderer Fall: Angenommen, alles klappt, aber die Kartenausgabe ist kaputt. In diesem Fall sind Sie ein paar Euro los und müssen sich trotzdem mit den Wiederholungen am heimischen Fernseher begnügen.

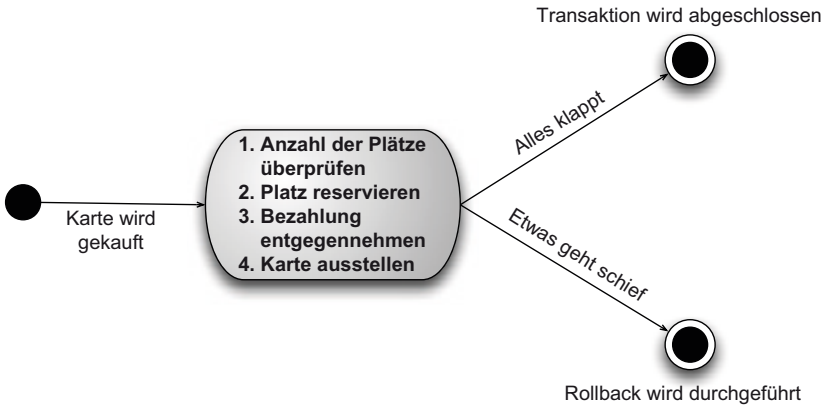

ABBILDUNG 6.1 Beim Kartenkauf müssen alle Schritte durchgeführt werden. Laufen alle Operationen erfolgreich ab, war die gesamte Transaktion erfolgreich. Andernfalls erfolgt ein Rollback – als wäre nie etwas passiert.

Um sicherzustellen, dass weder Sie noch das Kino Geld verlieren, müssen diese Aktionen also in einer Transaktion gekapselt werden. Als Transaktion werden diese Schritte in der Summe als gemeinsamer Vorgang behandelt, wodurch sichergestellt ist, dass sie entweder *alle* durchgeführt oder aber rückgängig gemacht werden, so als ob sie niemals stattgefunden hätten. Abbildung 6.1 zeigt, wie diese Transaktion erfolgt.

Transaktionen spielen in der Software eine wichtige Rolle, denn mit ihnen wird gewährleistet, dass Daten und Ressourcen keinesfalls einen inkonsistenten Zustand einnehmen. Ohne Transaktionen bestünde die Gefahr, dass Daten beschädigt werden oder den Businessregeln der Anwendung nicht mehr entsprechen.

Bevor wir uns jedoch zu sehr in die Transaktionsunterstützung von Spring vertiefen, müssen wir zunächst die Hauptbestandteile einer Transaktion kennenlernen. Werfen wir also einen kurzen Blick auf die vier Faktoren, die die Transaktionen und ihre Funktionsweise bestimmen.

6.1.1 Transaktionen – mit vier Wörtern erklärt

In der großen Tradition der Softwareentwicklung wurde zur Beschreibung von Transaktionen das sogenannte. *ACID*-Prinzip formuliert. ACID steht für die vier Elemente von Transaktionen: Atomarität, Konsistenz, Isoliertheit und Dauerhaftigkeit[1].

- *Atomarität* –Transaktionen bestehen aus einer oder mehreren Aktivitäten, die zu einer einzelnen Arbeitseinheit zusammengefasst sind. Die Atomarität gewährleistet, dass entweder alle oder keine Operation in den Transaktionen stattfinden. Wenn alle Vorgänge einwandfrei ausgeführt werden, ist die Transaktion erfolgreich. Schlägt ein Vorgang fehl, ist auch die gesamte Transaktion misslungen und wird rückgängig gemacht.
- *Konsistenz* – Nach Abschluss einer Transaktion – ob als Erfolg oder Fehlschlag – befindet sich das System in einem Zustand, der zum abgebildeten Business konsistent ist. Die Daten dürfen nicht von der Realität abweichen.
- *Isoliertheit* – Transaktionen sollen die Verarbeitung der Daten durch mehrere Benutzer ermöglichen, ohne dass die Handlungen eines Benutzers mit denen anderer Benutzer verwickelt werden. Aus diesem Grund müssen Transaktionen voneinander isoliert sein, damit gleichzeitige Lese- und Schreibvorgänge für dieselben Daten ausgeschlossen sind. (Beachten Sie, dass die Isoliertheit gewöhnlich das Sperren von Datensätzen und/oder Tabellen in einer Datenbank bedingt.)
- *Dauerhaftigkeit* – Nach Abschluss der Transaktion müssen deren Ergebnisse festgeschrieben werden, damit sie ggf. einen Systemabsturz überstehen. Hierzu werden die Ergebnisse normalerweise in einer Datenbank oder einem anderen Dauerspeicher gesichert.

Im Beispiel mit den Kinokarten könnte eine Transaktion die Atomarität sicherstellen, indem alle Schritte rückgängig gemacht werden, wenn ein Schritt fehlschlägt. Die Atomarität unterstützt die Konsistenz, indem sie sicherstellt, dass die Daten des Systems niemals in einem inkonsistenten, teilbearbeiteten Zustand verbleiben. Auch die Isoliertheit unterstützt die Konsistenz, denn sie verhindert, dass sich andere gleichzeitig erfolgende Transaktionen Plätze nehmen, die zu erwerben Sie gerade im Begriff sind.

[1] Das Akronym ergibt sich aus den englischen Bezeichnungen: *Atomicity, Consistency, Isolation, Durability*. Im Deutschen ist gelegentlich auch von AKID die Rede.

Die Auswirkungen schließlich sind dauerhaft, denn sie wurden an einen persistenten Speicher übergeben. Bei einem Systemausfall oder einem anderen schwerwiegenden Ereignis sollten Sie sich keine Sorgen mehr darüber machen müssen, dass die Ergebnisse der Transaktion verloren gegangen sein könnten.

Eine umfassende Erläuterung von Transaktionen finden Sie auch im Buch *Patterns für Enterprise Application-Architekturen* von Martin Fowler (MITP, 2003). Insbesondere das dortige Kapitel 5 behandelt die Konzepte von Transaktionen und Nebenläufigkeit.

Nun kennen Sie die konzeptionelle Zusammensetzung einer Transaktion. Wir wollen jetzt einmal sehen, wie die Transaktionsfähigkeit einer Spring-Anwendung verfügbar gemacht wird.

6.1.2 Spring-Support für die Transaktionsverwaltung

Wie EJB unterstützt Spring die programmgesteuerte wie auch die deklarative Transaktionsverwaltung. Doch die Spring-Funktionalitäten gehen diesbezüglich noch über das von EJB Gebotene hinaus.

Der Spring-Support für die programmgesteuerte Transaktionsverwaltung unterscheidet sich erheblich vom EJB-Gegenstück. Anders als EJB, das mit einer JTA-Implementierung (Java Transaction API) gekoppelt ist, verwendet Spring einen Callback-Mechanismus, der die eigentliche Transaktionsimplementierung aus dem Transaktionscode heraus abstrahiert. Springs Support für die Transaktionsverwaltung benötigt nicht einmal eine JTA-Implementierung. Wenn Ihre Anwendung nur eine einzige persistente Ressource verwendet, kann Spring den vom Persistenzmechanismus bereitgestellten Transaktionssupport verwenden. Unterstützt werden JDBC, Hibernate und die Java Persistence API (JPA). Erstrecken sich die Transaktionsanforderungen Ihrer Anwendung jedoch über mehrere Ressourcen, dann unterstützt Spring auch verteilte Transaktionen (XA-Transaktionen) unter Verwendung der JTA-Implementierung eines Drittanbieters. Den Spring-seitigen Support für programmgesteuerte Transaktionen behandeln wir in Abschnitt 6.3.

Während die programmgesteuerte Transaktionsverwaltung Ihnen die Flexibilität bietet, Transaktionsgrenzen in Ihrem Code präzise definieren zu können, helfen Ihnen deklarative Transaktionen bei der Entkopplung einer Operation von ihren Transaktionsregeln. Der Support für deklarative Transaktionen in Spring erinnert an die *containerverwalteten Transaktionen (CMTs)* in EJB: Beide gestatten Ihnen die deklarative Definition von Transaktionsgrenzen. Springs deklarative Transaktionen gehen jedoch über CMTs hinaus, denn sie ermöglichen das Deklarieren zusätzlicher Attribute wie der Isolationsebene oder Timeouts. In Abschnitt 6.4 beginnen wir mit der Verwendung des Spring-Supports für deklarative Transaktionen.

Die Wahl zwischen der programmgesteuerten und der deklarativen Transaktionsverwaltung ist im Wesentlichen eine Entscheidung zwischen abgestufter Steuerbarkeit und Bequemlichkeit. Wenn Sie Transaktionen in Ihren Code einprogrammieren, können Sie Transaktionsgrenzen präzise steuern und sie genau dort beginnen und enden lassen, wo Sie es wünschen. Gewöhnlich werden Sie die von programmgesteuerten Transaktionen gebotene abgestufte Steuerbarkeit nicht benötigen und Ihre Transaktionen daher vorwiegend in der Kontextdefinitionsdatei deklarieren.

Unabhängig davon, ob Sie Ihre Transaktionen in Ihren Beans programmieren oder sie als Aspekte deklarieren, werden Sie jedoch einen Spring-Transaktionsmanager verwenden, der als Schnittstelle zur plattformspezifischen Transaktionsimplementierung dient. Wie Trans-

aktionsmanager in Spring Sie vom direkten Umgang mit plattformspezifischen Transaktionsimplementierungen freistellen, zeigt der nächste Abschnitt.

6.2 Auswahl eines Transaktionsmanagers

Spring verwaltet Transaktionen nicht direkt, sondern bietet eine Auswahl von Transaktionsmanagern, die die Zuständigkeit für die Transaktionsverwaltung an eine plattformspezifische Transaktionsimplementierung delegieren, die entweder von JTA oder vom Persistenzmechanismus bereitgestellt wird. Die Transaktionsmanager aus Spring sind in Tabelle 6.1 aufgelistet.

TABELLE 6.1 Spring bietet Transaktionsmanager für jede Gelegenheit.

Transaktionsmanager (org.springframework.*)	Einsatzgebiet
`jca.cci.connection.CciLocalTransactionManager`	Wird bei Verwendung des Spring-Supports für JCA (J2EE Connector Architecture) und CCI (Common Client Interface) verwendet.
`jdbc.datasource.DataSourceTransactionManager`	Wird gemeinsam mit dem JDBC-Abstraktionssupport von Spring verwendet. Ist außerdem nützlich bei der Verwendung von iBATIS zu Persistenzzwecken.
`jms.connection.JmsTransactionManager`	Wird gemeinsam mit JMS 1.1+ verwendet.
`jms.connection.JmsTransactionManager102`	Wird gemeinsam mit JMS 1.0.2 verwendet.
`orm.hibernate3.HibernateTransactionManager`	Wird beim Einsatz von Hibernate 3 zu Persistenzzwecken verwendet.
`orm.jdo.JdoTransactionManager`	Wird beim Einsatz von JDO zu Persistenzzwecken verwendet.
`orm.jpa.JpaTransactionManager`	Wird beim Einsatz von JPA (Java Persistence API) zu Persistenzzwecken verwendet.
`transaction.jta.JtaTransactionManager`	Wird verwendet, wenn Sie verteilte Transaktionen benötigen oder kein anderer Transaktionsmanager für die Anforderungen geeignet ist.
`transaction.jta.OC4JJtaTransactionManager`	Wird gemeinsam mit dem Oracle OC4 J JEE-Container verwendet.
`transaction.jta.WebLogicJtaTransactionManager`	Wird verwendet, wenn Sie verteilte Transaktionen benötigen und Ihre Anwendung in WebLogic ausgeführt wird.
`transaction.jta.WebSphereUowTransactionManager`	In WebSphere müssen die Transaktionen von einem `UOWManager` verwaltet werden.

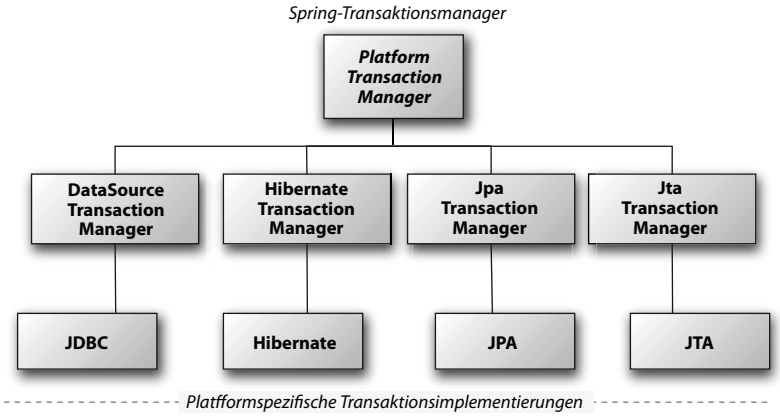

ABBILDUNG 6.2 Die Spring-Transaktionsmanager delegieren die Zuständigkeit für die Transaktionsverwaltung an plattformspezifische Transaktionsimplementierungen.

Alle diese Transaktionsmanager agieren als Fassade für eine plattformspezifische Transaktionsimplementierung. (Abbildung 6.2 veranschaulicht die Beziehung zwischen Transaktionsmanagern und den zugrunde liegenden Plattformimplementierungen für einige Transaktionsmanager.) Auf diese Weise können Sie mit einer Transaktion in Spring arbeiten, ohne sich groß Gedanken machen zu müssen, wie die eigentliche Transaktionsimplementierung aussieht.

Um einen Transaktionsmanager verwenden zu können, müssen Sie ihn in Ihrem Anwendungskontext deklarieren. In diesem Abschnitt erfahren Sie, wie Sie einige der häufiger verwendeten Spring-Transaktionsmanager konfigurieren. Wir beginnen mit `DataSourceTransactionManager`, dem Transaktionsmanager zur Unterstützung für JDBC und iBATIS.

6.2.1 JDBC-Transaktionen

Wenn Sie schlichtes JDBC für die Persistenz Ihrer Anwendung einsetzen, behandelt `DataSourceTransactionManager` die Transaktionsgrenzen für Sie. Sie verwenden `DataSourceTransactionManager` durch Verschaltung in der Anwendungskontextdefinition Ihrer Anwendung. Hierbei kommt der folgende XML-Code zum Einsatz:

```
<bean id="transactionManager" class="org.springframework.jdbc.
    ➥ datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource"/>
</bean>
```

Beachten Sie, dass der Eigenschaft `dataSource` eine Referenz auf eine Bean namens `dataSource` zugeordnet ist. Wahrscheinlich ist die Bean `dataSource` eine Bean des Typs `javax.sql.DataSource`, der an anderer Stelle in Ihrer Kontextdefinitionsdatei definiert ist.

Hinter den Kulissen verwaltet der `DataSourceTransactionManager` Transaktionen über Aufrufe des `java.sql.Connection`-Objekts, das aus der Datenquelle `DataSource` abgerufen wurde. So wird eine erfolgreiche Transaktion durch Aufruf der Methode `commit()` für die Verbindung abgeschlossen. Gleichermaßen wird eine fehlgeschlagene Transaktion durch einen Aufruf der Methode `rollback()` rückgängig gemacht.

6.2.2 Hibernate-Transaktionen

Wenn die Persistenz Ihrer Anwendung von Hibernate behandelt wird, sollten Sie den `HibernateTransactionManager` einsetzen. Bei Hibernate 3 müssen Sie die folgende `<bean>`-Deklaration in die Spring-Kontextdefinition einfügen:

```
<bean id="transactionManager" class="org.springframework.
      ➥ orm.hibernate3.HibernateTransactionManager">
  <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

Die Eigenschaft `sessionFactory` sollte mit einer Hibernate-`SessionFactory` verschaltet werden, die hier naheliegenderweise `sessionFactory` genannt wurde. Weitere Informationen zur Einrichtung einer Hibernate-Session-Factory finden Sie in Kapitel 5.

Was muss ich bei Hibernate 2 tun?

Wenn Sie zu Persistenzzwecken das ältere Hibernate 2 nutzen, können Sie in Spring 3.0 und nicht einmal Spring 2.5 den `HibernateTransactionManager` nicht verwenden. Diese Versionen von Spring unterstützen Hibernate 2 nicht. Wenn Sie unbedingt mit einer älteren Version von Hibernate arbeiten wollen, müssen Sie auf Spring 2.0 zurückgreifen.

Wenn Sie aber mit Ihrer älteren Version von Hibernate auf eine ältere Version von Spring zurückrollen, sollte Ihnen klar sein, dass Sie auf eine Menge der Spring-Features verzichten, über die wir in diesem Buch sprechen. Anstatt eines Rollbacks auf eine ältere Version von Spring empfehle ich deswegen ein Upgrade auf Hibernate 3.

`HibernateTransactionManager` delegiert die Zuständigkeit für die Transaktionsverwaltung an ein `org.hibernate.Transaction`-Objekt, das es aus der Hibernate-Session abruft. Wenn eine Transaktion erfolgreich abgeschlossen wurde, ruft `HibernateTransactionManager` die Methode `commit()` für das `Transaction`-Objekt auf; schlägt die Transaktion hingegen fehl, wird die Methode `rollback()` für das Objekt aufgerufen.

6.2.3 JPA-Transaktionen

Hibernate ist jahrelang der De-facto-Standard für die Persistenz in Java gewesen, mittlerweile hat JPA (Java Persistence API) aber die Bühne betreten und etabliert sich gegenwärtig als echter Java-Persistenzstandard. Wenn Sie auf den JPA-Zug aufspringen wollen, sollten Sie zur Koordination von Transaktionen Springs `JpaTransactionManager` einsetzen. Die Konfiguration von `JpaTransactionManager` in Spring könnte etwa so aussehen:

```
<bean id="transactionManager"
      class="org.springframework.orm.jpa.JpaTransactionManager">
  <property name="entityManagerFactory" ref="entityManagerFactory" />
</bean>
```

`JpaTransactionManager` muss lediglich mit einer JPA-Entity-Manager-Factory verschaltet werden (dies kann eine beliebige Implementierung von `javax.persistence.EntityManagerFactory` sein). `JpaTransactionManager` kollaboriert zur Durchführung von Transaktionen mit dem von der Factory generierten JPA-`EntityManager`.

Neben der Anwendung von Transaktionen bei JPA-Operationen unterstützt `JpaTransactionManager` auch Transaktionen bei einfachen JHDBC-Operationen an derselben Datenquelle, die auch von `EntityManagerFactory` verwendet wird. Damit dies funktioniert, muss `JpaTransactionManager` auch mit einer Implementierung von `JpaDialect` verschaltet werden. Nehmen wir beispielsweise an, Sie haben `EclipseLinkJpaDialect` folgendermaßen konfiguriert:

```
<bean id="jpaDialect"
    class="org.springframework.orm.jpa.vendor.EclipseLinkJpaDialect" />
```

In diesem Fall müssen Sie die `jpaDialect`-Bean wie folgt im `JpaTransactionManager` verschalten:

```
<bean id="transactionManager"
      class="org.springframework.orm.jpa.JpaTransactionManager">
  <property name="entityManagerFactory" ref="entityManagerFactory" />
  <property name="jpaDialect" ref="jpaDialect" />
</bean>
```

Es ist wichtig, festzuhalten, dass die `JpaDialect`-Implementierung gemischten JPA-/JDBC-Zugriff unterstützen muss, damit dies gelingt. Alle anbieterspezifischen `JpaDialect`-Implementierungen in Spring (`HibernateJpaDialect`, `OpenJpaDialect` und `TopLinkJpaDialect`) bieten diese Unterstützung für die gemischte Verwendung von JPA und JDBC, `DefaultJpaDialect` aber nicht.

6.2.4 JTA-Transaktionen

Erfüllt keiner der vorgenannten Transaktionsmanager Ihre Anforderungen, oder erstrecken sich Ihre Transaktionen über mehrere Transaktionsquellen (z. B. zwei verschiedene Datenbanken), dann müssen Sie `JtaTransactionManager` verwenden:

```
<bean id="transactionManager" class="org.springframework.
      ➥ transaction.jta.JtaTransactionManager">
  <property name="transactionManagerName"
      value="java:/TransactionManager" />
</bean>
```

`JtaTransactionManager` delegiert die Zuständigkeit für die Transaktionsverwaltung an eine JTA-Implementierung. JTA spezifiziert eine Standard-API zur Koordination von Transaktionen zwischen einer Anwendung und einer oder mehreren Datenquellen. Die Eigenschaft `transactionManagerName` gibt dabei einen JTA-Transaktionsmanager an, der via JNDI nachgeschlagen wird.

`JtaTransactionManager` arbeitet mit `javax.transaction.UserTransaction`- und `javax.transaction.TransactionManager`-Objekten und delegiert die Verantwortung für die Transaktionsverwaltung an diese Objekte. Eine erfolgreiche Transaktion wird mit dem Aufruf der Methode `UserTransaction.commit()` abgeschlossen, eine fehlgeschlagene mit der `rollback()`-Methode von `UserTransaction` rückgängig gemacht.

Mittlerweile sollte also nun klar sein, welche der Transaktionsmanager von Spring für die Spitter-Applikation am besten passen – soweit wir uns für einen Persistenzmechanismus entschieden haben. Nun wollen wir diesen Transaktionsmanager zum Laufen bringen. Wir beginnen mit der Verwendung eines Transaktionsmanagers zur manuellen Programmierung von Transaktionen.

6.3 Transaktionen in Spring programmieren

Es gibt zwei Arten von Menschen: Kontrollfreaks und andere. Kontrollfreaks nehmen nichts für bare Münze. Wenn Sie Entwickler *und* Kontrollfreak sind, bevorzugen Sie wahrscheinlich die Befehlszeile und schreiben lieber eigene Anfrage- und Änderungsmethoden, als diese Arbeit einer IDE zu überlassen.

Kontrollfreaks wollen auch immer genau wissen, was gerade in ihrem Code passiert. Wenn es um Transaktionen geht, wollen sie exakt festlegen können, wo eine Transaktion startet, Daten festschreibt und endet. Deklarative Transaktionen sind nicht präzise genug für solche Leute.

Das ist aber auch nichts Schlimmes: Kontrollfreaks sind zumindest teilweise durchaus im Recht. Wie Sie im weiteren Verlauf dieses Kapitels sehen werden, sind Sie bei der Deklaration von Transaktionen auf die Methodenebene beschränkt. Wollen Sie die Grenzen Ihrer Transaktionen mit feinerer Abstufung steuern, dann *müssen* Sie den Weg der programmgesteuerten Transaktionen gehen.

Betrachten Sie die Methode `saveSpittle()` aus `SpitterServiceImpl` (Listing 6.1) als Beispiel für eine Transaktionsmethode.

LISTING 6.1 `saveSpittle()` speichert einen `Spittle`.

```
public void saveSpittle(Spittle spittle) {
  spitterDao.saveSpittle(spittle);
}
```

Obwohl diese Methode recht einfach wirkt, steckt doch mehr dahinter, als auf den ersten Blick sichtbar. Wenn der gespeichert wird, hat der zugrunde liegende Persistenzmechanismus wohl eine Menge zu tun. Auch wenn es letzten Endes nur darum geht, eine Zeile in eine Datenbanktabelle einzufügen, muss man darauf achten, dass alles, was passiert, sich im Rahmen einer Transaktion abspielt. Läuft dies erfolgreich ab, sollte die Arbeit übermittelt werden. Wenn nicht, sollte man alles zurückrollen.

Ein Ansatz zum Hinzufügen von Transaktionen besteht darin, die Transaktionsgrenzen durch Programmierung direkt in der Methode `saveSpittle()` zu ergänzen. Hierbei käme die Spring-Vorlage `TransactionTemplate` zum Einsatz. Wie andere Vorlagenklassen in Spring (z. B. das in Kapitel 5 beschriebene `JdbcTemplate`) verwendet `TransactionTemplate` einen Callback-Mechanismus. Hier ist eine aktualisierte `saveSpittle()`-Methode, um zu zeigen, wie mit einem `TransactionTemplate` ein Transaktionskontext eingefügt wird.

LISTING 6.2 Programmgesteuertes Hinzufügen von Transaktionen zu `saveSpittle()`

```
public void saveSpittle(final Spittle spittle) {
  txTemplate.execute(new TransactionCallback<Void>() {
    public Void doInTransaction(TransactionStatus txStatus) {
      try {
      spitterDao.saveSpittle(spittle);
      } catch (RuntimeException e) {
        txStatus.setRollbackOnly();
        throw e;
      }
      return null;
    }
  });
}
```

Um das `TransactionTemplate` verwenden zu können, implementieren Sie zunächst die Schnittstelle `TransactionCallback`. Weil `TransactionCallback` über nur eine Methode zur Implementierung verfügt, ist es häufig am einfachsten, sie wie in Listing 6.2 als anonyme innere Klasse zu implementieren. Der transaktionsgestützte Code wird innerhalb der Methode `doInTransaction()` abgelegt.

Bei Aufruf der Methode `execute()` für die `TransactionTemplate`-Instanz wird der in der `TransactionCallback`-Instanz enthaltene Code ausgeführt. Wenn Ihr Code auf ein Problem stößt, wird die Transaktion durch den Aufruf von `setRollbackOnly()` für das `TransactionStatus`-Objekt rückgängig gemacht. Andernfalls –wenn also die Methode `doInTransaction()` erfolgreich zurückkehrt – wird die Transaktion abgeschlossen.

Woher stammt nun die `TransactionTemplate`-Instanz? Gute Frage – sie sollte wie folgt in `SpitterServiceImpl` injiziert werden:

```
<bean id="spitterService"
      class="com.habuma.spitter.service.SpitterServiceImpl">
 ...
  <property name="transactionTemplate ">
    <bean class="org.springframework.transaction.support.
            TransactionTemplate">
      <property name="transactionManager"
          ref="transactionManager" />
    </bean>
  </property>
</bean>
```

Beachten Sie, dass dem `TransactionTemplate` ein `transactionManager` injiziert wird. Hinter den Kulissen verwendet `TransactionTemplate` eine Implementierung von `PlatformTransactionManager` zur Behandlung der plattformspezifischen Details der Transaktionsverwaltung. Hier haben wir eine Referenz auf eine Bean namens `transactionManager` verschaltet, bei der es sich um irgendeine der in Tabelle 6.1 auf Seite 159 aufgeführten Transaktionen handeln kann.

Programmgesteuerte Transaktionen sind die beste Wahl, wenn Sie vollständige Kontrolle über die Transaktionsgrenzen wünschen. Sie können jedoch, wie Sie dem Code in Listing 6.1 auf Seite 163 entnehmen können, ein wenig aufdringlich sein. Dort mussten Sie die Implementierung von `saveSpittle()` mithilfe Spring-spezifischer Klassen ändern, um den programmgesteuerten Transaktionssupport von Spring verwenden zu können.

Gewöhnlich benötigen Sie keine derart präzise Kontrolle über die Transaktionsgrenzen. Aus diesem Grund werden Sie Ihre Transaktionen in der Regel außerhalb Ihres Anwendungscodes deklarieren – z. B. in der Spring-Konfigurationsdatei. Der Rest dieses Kapitels wird die deklarative Transaktionsverwaltung in Spring behandeln.

6.4 Transaktionen deklarieren

Noch vor nicht allzu langer Zeit war die deklarative Transaktionsverwaltung eine Fähigkeit, die nur in EJB-Containern verfügbar war. Mittlerweile bietet Spring jedoch Unterstützung für deklarative Transaktionen mit POJOs. Dies ist ein ganz wesentliches Feature von Spring, da Ihnen zur Deklaration atomarer Operationen nun eine Alternative zur Verfügung steht.

Der Spring-Support für die deklarative Transaktionsverwaltung wird über das AOP-Framework von Spring implementiert. Dieser Ansatz ist naheliegend, denn Transaktionen sind ein Dienst auf Systemebene oberhalb der primären Funktionalität einer Anwendung. Sie können sich eine Spring-Transaktion als Aspekt vorstellen, der eine Methode mit Transaktionsgrenzen „kapselt“.

Spring bietet drei Möglichkeiten der Deklaration von Transaktionsgrenzen in der Spring-Konfiguration und hat deklarative Transaktionen schon immer durch Proxy-Generierung für Beans mithilfe von Spring AOP und `TransactionProxyFactoryBean` unterstützt. Seit Spring 2.0 gehen die beiden jedoch in Form des Konfigurationsnamensraums `tx` und der Annotation `@Transactional`. Sonderwege, um Transaktionen zu deklarieren.

Obwohl es in modernen Versionen von Spring auch die Legacy-Bean `TransactionProxyFactoryBean` gibt, ist sie effektiv obsolet, und wir werden uns damit auch nicht mehr beschäftigen. Stattdessen konzentrieren wir uns weiter unten auf den `tx`-Namensraum und annotationsorientierte deklarative Transaktionen. Zunächst untersuchen wir aber die Transaktionen definierenden Attribute.

6.4.1 Transaktionsattribute definieren

In Spring werden deklarative Transaktionen mit *Transaktionsattributen* definiert. Ein Transaktionsattribut stellt eine Beschreibung dar, wie Transaktionsrichtlinien auf eine Methode angewendet werden sollen. Es gibt fünf Facetten eines Transaktionsattributs (siehe Abbildung 6.3).

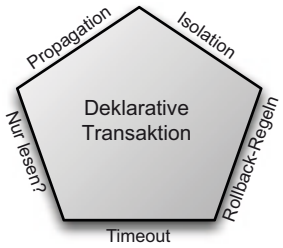

ABBILDUNG 6.3 Deklarative Transaktionen definieren sich durch ihr Propagationsverhalten, die Isolationsebene, Angaben zum Nur-Lese-Verhalten, Timeouts und Rollback-Regeln.

Zwar bietet Spring mehrere Mechanismen zur Deklaration von Transaktionen, sie alle basieren jedoch auf diesen fünf Parametern, die bestimmen, wie Transaktionsrichtlinien administriert werden. Aus diesem Grund ist es wichtig, diese Parameter zu kennen, um die Transaktionsrichtlinien in Spring deklarieren zu können.

Unabhängig von dem von Ihnen verwendeten deklarativen Transaktionsmechanismus haben Sie stets die Möglichkeit, diese Attribute zu definieren. Wir wollen sie im Einzelnen beschreiben, um zu verstehen, wie sie sich jeweils auf die Transaktion auswirken.

Propagationsverhalten

Die erste Facette einer Transaktion ist das *Propagationsverhalten*. Dieses definiert die Grenzen der Transaktion in Bezug auf den Client und die aufgerufene Methode. Spring definiert sieben separate Verhaltensweisen für die in Tabelle 6.2 beschriebene Propagation.

> **Propagationskonstanten**
>
> Die in der Tabelle aufgelisteten Verhaltensweisen sind als Konstanten in der Schnittstelle `org.springframework.transaction.TransactionDefinition` definiert.

Die in Tabelle 6.2 aufgeführten Propagationsverhaltensweisen kommen Ihnen unter Umständen bekannt vor – was daran liegt, dass sie die in EJB-CMTs vorhandenen Propagationsregeln reflektieren. So entspricht beispielsweise Springs `PROPAGATION_REQUIRES_NEW` dem CMT-Verhalten `RequiresNew`. Spring ergänzt ein zusätzliches, in CMT nicht vorhandenes Propagationsverhalten namens `PROPAGATION_NESTED`, um verschachtelte Transaktionen zu unterstützen.

Propagationsregeln bestimmen die Antwort auf die Frage, ob eine neue Transaktion gestartet oder unterbrochen oder ob eine Methode überhaupt in einem Transaktionskontext ausgeführt werden sollte.

Wenn beispielsweise eine Methode als transaktionsgestützt mit dem Verhalten `PROPAGATION_REQUIRES_NEW` deklariert wird, bedeutet dies, dass die Transaktionsgrenzen den Grenzen

der Methode selbst entsprechen: Es wird eine neue Transaktion gestartet, wenn die Methode beginnt, und die Transaktion endet, sobald die Methode zurückkehrt oder eine Exception auslöst. Weist die Methode hingegen das Verhalten `PROPAGATION_REQUIRED` auf, so hängen die Transaktionsgrenzen davon ab, ob eine Transaktion bereits offen ist.

TABELLE 6.2 Propagationsregeln definieren, wann eine Transaktion erstellt wird oder eine offene Transaktion verwendet werden kann. Spring bietet diverse Propagationsregeln zur Auswahl an.

Propagationsverhalten	Bedeutung
`PROPAGATION_MANDATORY`	Gibt an, dass die Methode innerhalb einer Transaktion ausgeführt werden muss. Ist keine offene Transaktion vorhanden, wird eine Exception ausgelöst.
`PROPAGATION_NESTED`	Gibt an, dass die Methode innerhalb einer verschachtelten Transaktion ausgeführt werden soll, sofern eine offene Transaktion voliegt. Die verschachtelte Transaktion kann separat von der umschließenden Transaktion abgeschlossen oder rückgängig gemacht werden. Wenn keine umschließende Transaktion vorhanden ist, verhält sich die Regel wie `PROPAGATION_REQUIRED`. Die anbieterseitige Unterstützung für dieses Propagationsverhalten ist maximal als punktuell zu bezeichnen. Schlagen Sie in der Dokumentation zu Ihrem Ressourcenmanager nach, um zu ermitteln, ob verschachtelte Transaktionen unterstützt werden.
`PROPAGATION_NEVER`	Gibt an, dass die aktuelle Methode nicht innerhalb eines Transaktionskontexts ausgeführt werden darf. Ist eine offene Transaktion vorhanden, wird eine Exception ausgelöst.
`PROPAGATION_NOT_SUPPORTED`	Gibt an, dass die Methode nicht innerhalb einer Transaktion ausgeführt werden darf. Wenn eine offene Transaktion vorhanden ist, wird diese für die Dauer der Methode unterbrochen. Wenn der `JTATransactionManager` verwendet wird, ist ein Zugriff auf `TransactionManager` erforderlich.
`PROPAGATION_REQUIRED`	Gibt an, dass die aktuelle Methode innerhalb einer Transaktion ausgeführt werden muss. Wenn eine offene Transaktion vorhanden ist, wird die Methode innerhalb dieser Transaktion ausgeführt. Andernfalls wird eine neue Transaktion gestartet.
`PROPAGATION_REQUIRES_NEW`	Gibt an, dass die aktuelle Methode innerhalb ihrer eigenen Transaktion ausgeführt werden muss. Eine neue Transaktion wird gestartet und – sofern eine offene Transaktion vorhanden ist – für die Dauer der Methode unterbrochen. Wenn der `JTATransactionManager` verwendet wird, ist ein Zugriff auf `TransactionManager` erforderlich.
`PROPAGATION_SUPPORTS`	Gibt an, dass die aktuelle Methode keinen Transaktionskontext erfordert, aber innerhalb einer ggf. bereits offenen Transaktion ausgeführt werden kann.

Isolationsebenen

Die zweite Dimension einer deklarierten Transaktion ist die *Isolationsebene*. Eine Isolationsebene definiert, wie stark eine Transaktion von den Aktivitäten anderer nebenläufiger Transaktionen betroffen werden kann. Eine andere Möglichkeit, die Isolationsebene einer Transaktion zu formulieren, besteht darin, sich zu fragen, wie „egoistisch" eine Transaktion mit den Transaktionsdaten verfährt.

In einer typischen Anwendung laufen Transaktionen gleichzeitig ab und arbeiten dabei oft mit denselben Daten, um ihre Aufgabe zu erledigen. Die Nebenläufigkeit kann – wiewohl erforderlich – zu den folgenden Problemen führen:

- *Dirty Reads* erfolgen, wenn eine Transaktion Daten ausliest, die von einer anderen Transaktion gespeichert, aber noch nicht festgeschrieben wurden. Sollten die Änderungen später rückgängig gemacht werden, dann wären die von der ersten Transaktion abgerufenen Daten ungültig.
- *Nonrepeatable Reads* finden statt, wenn eine Transaktion dieselbe Abfrage mehrfach durchführt und die Daten sich jedes Mal unterscheiden. Dies liegt gewöhnlich daran, dass eine andere nebenläufige Transaktion die Daten zwischen den Abfragen ändern.
- *Phantom Reads* ähneln den Nonrepeatable Reads. Diese treten auf, wenn eine Transaktion T1 mehrere Datensätze ausliest und eine nebenläufige Transaktion T2 danach Datensätze einfügt. Bei nachfolgenden Abfragen findet Transaktion T1 neue Datensätze, die zuvor nicht vorhanden waren.

In einer idealen Situation wären Transaktionen vollständig voneinander isoliert, wodurch diese Probleme vermieden würden. Allerdings kann sich die vollständige Isolation auf die Leistungsfähigkeit auswirken, weil hierbei häufig Datensätze (und manchmal auch ganze Tabellen) im Datenspeicher gesperrt werden. Ein allzu aggressives Sperren kann die Nebenläufigkeit beeinträchtigen – Transaktionen müssen aufeinander warten, um ihre Aufgaben erledigen zu können.

Die Feststellung, dass eine vollständige Isolation Leistungseinbußen verursachen kann, und die Tatsache, dass nicht alle Anwendungen eine solche umfassende Isolation benötigen, machen eine gewisse Flexibilität in Bezug auf die Transaktionsisolation gelegentlich wünschenswert. Aus diesem Grund gibt es mehrere Isolationsebenen, die in Tabelle 6.3 aufgeführt sind.

> **Konstanten der Isolationsebenen**
>
> Die in der Tabelle aufgelisteten Verhaltensweisen sind als Konstanten in der Schnittstelle `org.springframework.transaction.TransactionDefinition` definiert.

`ISOLATION_READ_UNCOMMITTED` ist die effizienteste Isolationsebene, isoliert die Transaktion jedoch am wenigsten stark und lässt so Raum für Dirty Reads, Nonrepeatable Reads und Phantom Reads. Das andere Extrem `ISOLATION_SERIALIZABLE` verhindert alle Arten von Isolationsproblemen, ist aber am wenigsten effizient.

Beachten Sie, dass nicht alle Datenquellen alle in Tabelle 6.3 aufgeführten Isolationsebenen unterstützen. Schlagen Sie in der Dokumentation zu Ihrem Ressourcenmanager nach, um zu ermitteln, welche Isolationsebenen vorhanden sind.

TABELLE 6.3 Die Isolationsebenen bestimmen, bis zu welchem Grad sich nebenläufig ausgeführte Transaktionen aufeinander auswirken können.

Isolationsebene	Bedeutung
`ISOLATION_DEFAULT`	Verwendet die vorgegebene Isolationsebene des zugrunde liegenden Datenspeichers.
`ISOLATION_READ_UNCOMMITTED`	Gestattet Ihnen das Lesen von Änderungen, die noch nicht abgeschlossen wurden. Die Folge können Dirty Reads, Phantom Reads und Nonrepeatable Reads sein.
`ISOLATION_READ_COMMITTED`	Gestattet das Lesen aus nebenläufigen Transaktionen, die bereits abgeschlossen wurden. Dies verhindert Dirty Reads, während Phantom Reads und Nonrepeatable Reads nach wie vor auftreten können.
`ISOLATION_REPEATABLE_READ`	Mehrfaches Auslesen desselben Feldes führt zu identischen Ergebnissen, sofern nicht durch die Transaktion selbst Änderungen vorgenommen wurden. Dies verhindert Dirty Reads und Nonrepeatable Reads, während Phantom Reads nach wie vor auftreten können.
`ISOLATION_SERIALIZABLE`	Diese vollständig ACID-kompatible Isolationsebene gewährleistet, dass Dirty Reads, Nonrepeatable Reads und Phantom Reads nicht auftreten können. Es handelt sich hierbei um die langsamste aller Isolationsebenen, denn diese wird in der Regel durch vollständiges Sperren der an der Transaktion beteiligten Tabellen realisiert.

Nur lesende Transaktionen

Das dritte Merkmal einer deklarierten Transaktion ist der Schreibschutz. Wenn eine Transaktion lediglich Leseoperationen am zugrunde liegenden Datenspeicher ausführt, ist der Datenspeicher unter Umständen in der Lage, bestimmte Optimierungen durchzuführen, die auf dem nichtschreibenden Wesen der Transaktion fußen. Indem Sie eine Transaktion als nur lesend deklarieren, bieten Sie dem Datenspeicher die Möglichkeit, diese Optimierungen nach Bedarf anzuwenden.

Da Nur-Lese-Optimierungen am zugrunde liegenden Datenspeicher zu Beginn einer Transaktion ausgeführt werden, ist es nur dann sinnvoll, eine Transaktion als nur lesend zu deklarieren, wenn Methoden mit Propagationsverhalten betroffen sind, die eine neue Transaktion einleiten könnten (also `PROPAGATION_REQUIRED`, `PROPAGATION_REQUIRES_NEW` und `PROPAGATION_NESTED`).

Hinzu kommt, dass, wenn Sie Hibernate als Persistenzmechanismus einsetzen, das Deklarieren einer Transaktion als nur lesend dazu führt, dass der Leerungsmodus von Hibernate auf `FLUSH_NEVER` festgelegt wird. Hierdurch wird Hibernate angewiesen, eine nicht erforderliche Synchronisierung von Objekten mit der Datenbank zu unterlassen, wodurch alle Änderungen ans Ende der Transaktion verschoben werden.

Timeout

Wenn eine Anwendung eine gute Leistung erzielen soll, dürfen die zugehörigen Transaktionen nicht allzu lange dauern. Aus diesem Grund ist der nächste Parameter einer deklarierten Transaktion der *Timeout*.

Angenommen, die Ausführung Ihrer Transaktion dauert unerwartet lang. Weil es bei Transaktionen zu Sperrungen des zugrunde liegenden Datenspeichers kommen kann, können lang andauernde Transaktionen die Datenbankressourcen unnötig binden. Statt bis zum Ende zu warten, können Sie eine Transaktion so deklarieren, dass sie nach einer bestimmten Zeit automatisch einen Rollback durchführt.

Da die Zeitnahme für den Timeout zu Beginn einer Transaktion gestartet wird, ist es nur dann sinnvoll, einen Timeout für eine Transaktion zu deklarieren, wenn Methoden mit Propagationsverhalten betroffen sind, die eine neue Transaktion einleiten könnten (also `PROPAGATION_REQUIRED`, `PROPAGATION_REQUIRES_NEW` und `PROPAGATION_NESTED`).

Rollback-Regeln

Der letzte Aspekt des Transaktionsfünfecks ist ein Regelsatz, der definiert, welche Exceptions einen Rollback auslösen und welche nicht. Standardmäßig werden Transaktionen nur bei Laufzeit-Exceptions rückgängig gemacht, nicht jedoch bei geprüften Exceptions. (Dies entspricht dem Rollback-Verhalten bei EJBs.)

Sie können allerdings deklarieren, dass eine Transaktion bei Auftreten bestimmter geprüfter Exceptions sowie von Laufzeit-Exceptions rückgängig gemacht wird. Ähnlich können Sie festlegen, dass Transaktionen bei Auftreten bestimmter geprüfter Exceptions nicht rückgängig gemacht werden, auch wenn es sich dabei um Laufzeit-Exceptions handelt.

Nachdem Sie nun einen Überblick darüber erhalten haben, wie Transaktionsattribute das Verhalten einer Transaktion bestimmen, wollen wir als Nächstes besprechen, wie man diese Attribute bei der Deklaration von Transaktionen in Spring verwendet.

6.4.2 Transaktionen in XML deklarieren

Bei früheren Spring-Versionen bedingte die deklarative Transaktion die Verschaltung einer speziellen Bean namens `TransactionProxyFactoryBean`. Das Problem bei `TransactionProxyFactoryBean` war, dass dies zu extrem umfangreichen Spring-Konfigurationsdateien führte. Zum Glück ist diese Phase vorbei, und Spring bietet nun den Konfigurationsnamensraum `tx`, der die deklarativen Transaktionen in Spring deutlich vereinfacht.

Um diesen Namensraum `tx` zu nutzen, muss er in die XML-Konfigurationsdatei von Spring eingefügt werden:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/
           spring-beans-3.0.xsd
      http://www.springframework.org/schema/aop
      http://www.springframework.org/schema/aop/spring-aop-3.0.xsd
      http://www.springframework.org/schema/tx
      http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">
```

Beachten Sie, dass der Namensraum `aop` ebenfalls enthalten sein sollte. Dies ist wichtig, weil die neuen Konfigurationselemente für deklarative Transaktionen auf einige der neuen AOP-Konfigurationselemente in Spring (vgl. Kapitel 4) angewiesen sind.

Der Namensraum `tx` bietet eine Handvoll neuer XML-Konfigurationselemente, deren wohl wichtigstes das Element `<tx:advice>` ist.Der folgende XML-Ausschnitt zeigt, wie `<tx:advice>` zur Deklaration von Transaktionsrichtlinien ähnlich denjenigen verwendet werden kann, die wir für den Spitter-Dienst in Listing 6.2 verwendet haben:

```
<tx:advice id="txAdvice">
  <tx:attributes>
    <tx:method name="add*" propagation="REQUIRED" />
    <tx:method name="*" propagation="SUPPORTS"
        read-only="true"/>
  </tx:attributes>
</tx:advice>
```

Bei `<tx:advice>` werden die Transaktionsattribute in einem `<tx:attributes>`-Element definiert, das ein oder mehrere <tx:method>-Elemente enthält. Das `<tx:method>`-Element definiert die Transaktionsattribute für eine oder mehrere gegebene Methoden entsprechend der Definition im Attribut `name` (unter Verwendung von Platzhaltern).

`<tx:method>` verfügt über eine Reihe von Attributen, die die Definition von Transaktionsrichtlinien unterstützen (siehe Tabelle 6.4).

TABELLE 6.4 Die Eigenschaften des Transaktionspentagons (vgl. Abbildung 6.3) werden in den Attributen des `<tx:method>`-Elements festgelegt.

Attribut	Zweck
`isolation`	Gibt die Isolationsebene für die Transaktion an.
`propagation`	Definiert die Propagationsregel für die Transaktion.
`read-only`	Legt fest, dass eine Transaktion nur liest und nicht schreibt.

Rollback-Regeln
`rollback-for`
`no-rollback-for`
	`rollback-for` legt geprüfte Exceptions fest, bei deren Auftreten die Transaktion nicht abgeschlossen, sondern über einen Rollback rückgängig gemacht wird.
	`no-rollback-for` legt Exceptions fest, bei deren Auftreten die Transaktion fortgesetzt werden soll (d. h. es findet kein Rollback statt).

`timeout`	Definiert einen Timeout für eine länger andauernde Transaktion.

Wie im Transaktion-Advice `txAdvice` definiert, lassen sich die konfigurierten Transaktionsmethoden in zwei Kategorien unterteilen: solche, deren Namen mit `add` beginnen, und alle anderen. Die Methode `saveSpittle()` gehört zur ersten Kategorie und ist so deklariert, dass sie eine Transaktion erfordert. Die übrigen Methoden werden mit `propagation ="supports"` deklariert, d. h. sie laufen in einer Transaktion, sofern eine solche vorhanden ist, sind allerdings nicht darauf angewiesen.

Wenn Sie eine Transaktion mit `<tx:advice>` deklarieren, benötigen Sie ebenso wie oben bei der Verwendung von `TransactionProxyFactoryBean` einen Transaktionsmanager. Eher konventions- als konfigurationsorientiert, setzt `<tx:advice>` voraus, dass der Trans-

aktionsmanager als Bean mit der ID `transactionManager` deklariert wird. Wenn Sie Ihrem Transaktionsmanager einen anderen Wert für `id` geben (z. B. `txManager`), müssen Sie diese `id` im Attribut `transactionmanager` angeben:

```
<tx:advice id="txAdvice"
      transaction-manager="txManager">
...
</tx:advice>
```

Für sich genommen definiert `<tx:advice>` nur einen AOP-Advice, der Methoden mit Transaktionsgrenzen verknüpft. Dies ist jedoch nur ein Transaktions-Advice und kein vollständiger transaktionsgestützter Aspekt. Nirgendwo im `<tx:advice>`-Element haben wir angegeben, welche Beans mit dem Advice verknüpft werden sollen – wir benötigen hierfür einen Pointcut. Um die Definition des Transaktionsaspekts zu vervollständigen, müssen wir einen Advisor definieren. Hier kommt der Namensraum `aop` ins Spiel. Der folgende XML-Code definiert einen Advisor, der mithilfe von `txAdvice` allen Beans, die die Schnittstelle `SpitterService` implementieren, einen Advice zur Verfügung stellt:

```
<aop:config>
  <aop:advisor
      pointcut="execution(* *..SpitterService.*(..))"
      advice-ref="txAdvice"/>
</aop:config>
```

Das Attribut `pointcut` verwendet einen AspectJ-Pointcut-Ausdruck, um anzugeben, dass der Advisor allen Methoden der Schnittstelle `SpitterService` einen Advice verfügbar machen soll. Welche Methoden tatsächlich innerhalb einer Transaktion ausgeführt werden und wie die Transaktionsattribute dieser Methoden aussehen, definiert der Transaktions-Advice, der mit dem Attribut `advice-ref` referenziert wird (in unserem Fall der Advice `txAdvice`).

Zwar bewirkt das Konfigurationselement `<tx:advice>` schon eine ganze Menge, um Spring-Entwicklern die deklarativen Transaktionen schmackhafter zu machen, doch gibt es ein weiteres – neues – Feature in Spring 2.0, das diese Aufgabe all jenen, die in einer Java 5-Umgebung arbeiten, weiter vereinfacht. Betrachten wir also Spring-Transaktionen in ihrer annotationsgetriebenen Ausprägung.

6.4.3 Annotationsgetriebene Transaktionen definieren

Das Konfigurationselement `<tx:advice>` vereinfacht den für deklarative Transaktionen in Spring erforderlichen XML-Code erheblich. Und wenn eine noch stärkere Vereinfachung möglich wäre? Oder wenn Sie Ihrem Spring-Kontext nur eine einzige XML-Codezeile hinzufügen müssten, um Transaktionen deklarieren zu können?

Zusätzlich zum `<tx:advice>`-Element bietet der Namensraum `tx` nämlich noch das Konfigurationselement `<tx:annotation-driven>`. Die Verwendung von `<tx:annotation-driven>` erfordert oft lediglich die folgende XML-Codezeile:

```
<tx:annotation-driven />
```

Tja – und das war's schon! Wenn Sie mehr erwartet haben, muss ich Sie enttäuschen. Ich hätte den Sachverhalt vielleicht ein wenig interessanter schildern können, etwa durch Angabe einer bestimmten Transaktionsmanager-Bean mit dem Attribut `transactionmanager` (dessen Vorgabewert `transactionManager` lautet):

```
<tx:annotation-driven transaction-manager="txManager" />
```

Mehr gibt es aber nicht zu besprechen. Diese eine Codezeile leistet Erstaunliches, denn sie gestattet die Definition von Transaktionsregeln an der Stelle, wo es am sinnvollsten ist: in den Methoden, die transaktionsgestützt ausgeführt werden sollen.

Annotationen gehören zu den umfangreichsten und meistdiskutierten Features von Java 5. Sie gestatten Ihnen die Definition von Metadaten direkt im Code statt in externen Konfigurationsdateien. Ich halte sie für den perfekten Partner bei der Deklaration von Transaktionen.

Das Konfigurationselement `<tx:annotation-driven>` weist Spring an, alle Beans im Anwendungskontext zu überprüfen und nach Beans zu suchen, die auf der Klassen- oder der Methodenebene mit `@Transactional` annotiert sind. Alle Beans mit dieser Annotation `@Transactional` erhalten über `<tx:annotation-driven>` automatisch einen Transaktions-Advice. Die Transaktionsattribute des Advices werden von Parametern der Annotation `@Transactional` definiert.

Das folgende Listing beispielsweise zeigt `SpitterServiceImpl` in einer Form an, die so abgeändert wurde, dass die `@Transactional`-Annotationen enthalten sind.

LISTING 6.3 Annotieren des Spitter-Dienstes zur Transaktionsunterstützung

```
@Transactional(propagation=Propagation.SUPPORTS, readOnly=true)
public class SpitterServiceImpl implements SpitterService {
...
  @Transactional(propagation=Propagation.REQUIRED, readOnly=false)
  public void addSpitter(Spitter spitter) {
...
  }
...
}
```

Auf der Klassenebene wurde `SpitterServiceImpl` mit einer `@Transactional`-Annotation versehen, die besagt, dass alle Methoden Transaktionen unterstützen und nur lesen. Auf der Methodenebene wurde die Methode `saveSpittle()` annotiert und gibt nun an, dass sie einen Transaktionskontext benötigt.

6.5 Zusammenfassung

Transaktionen sind ein wesentlicher Bestandteil der Entwicklung von Unternehmensanwendungen und haben eine größere Robustheit auf Seiten der Software zur Folge. Sie gewährleisten ein Alles-oder-nichts-Verhalten und verhindern Dateninkonsistenzen beim Auftreten unvorhergesehener Ereignisse. Zudem unterstützen sie die Nebenläufigkeit, da sie verhindern, dass gleichzeitig laufende Anwendungs-Threads einander stören, wenn sie dieselben Daten bearbeiten.

Spring unterstützt sowohl programmgesteuerte als auch deklarative Transaktionsverwaltung. In beiden Fällen erspart Ihnen Spring den direkten Umgang mit einer bestimmten Transaktionsverwaltungsimplementierung, indem es die entsprechende Plattform hinter einer allgemeinen API verbirgt.

Spring nutzt sein eigenes AOP-Framework zur Unterstützung der deklarativen Transaktionsverwaltung. Der Support deklarativer Transaktionen durch Spring steht in Konkurrenz zu EJB-CMTs und ermöglicht Ihnen in Zusammenhang mit POJOs die Deklaration nicht nur des Propagationsverhaltens, sondern auch anderer Parameter: Isolationsebenen, Nur-Lese-Optimierungen und Rollback-Regeln für bestimmte Exceptions.

In diesem Kapitel haben Sie auch gesehen, wie man deklarative Transaktionen mithilfe von Annotationen im Programmiermodell von Java 5 unterbringt. Dank der Java 5-Annotationen müssen Sie, um einer Methode die Transaktionsfähigkeit beizubringen, diese nur noch mit der passenden Transaktionsannotation versehen.

Wie wir gesehen haben, verleiht Spring POJOs die Leistungsfähigkeit deklarativer Transaktionen. Dies ist eine spannende Entwicklung, denn immerhin waren deklarative Transaktionen zuvor den EJBs vorbehalten. Deklarative Transaktionen sind jedoch nur der Anfang dessen, was Spring den POJOs zu bieten hat. Im nächsten Kapitel erfahren wir, wie Spring die deklarative Sicherheit auf POJOs erweitert.