Transaktionen basierend auf Commands und Mementos

Markus Völter, voelter@acm.org, www.voelter.de

In vielen Anwendungen wird Transaktionssemantik benötigt. Um diese zu implementieren, gibt es verschiedene Hilfsmittel, darunter Transaktionsmonitore bzw. die Interfaces und Klassen im Package *javax.jta*. Jedoch ist dies für viele Anwendungen zuviel des Guten. Beispielsweise hat auch das Rückgängigmachen von Gruppen von Kommandos einige Aspekte mit Transaktionen gemeinsam. Im Rahmen dieses Artikels soll aufgezeigt werden, wie sich derartige Transaktionen implementieren lassen. Dieser Artikel baut auf dem Artikel über Mementos aus dem letzten Heft auf.

Ein Beispiel

Stellen wir uns als Beispiel ein UML-Modellierungstool vor (oder irgend ein anderes Zeichenprogramm, aber UML-Modeller sind gerade in!). In solchen Anwendungen kann man normalerweise eine Gruppe von Objekte markieren, und auf allen Objekten die gleiche Operation durchführen, beispielsweise verschieben der Klassen in ein anderes Java-Package. Dabei kann es passieren, dass die geplante Änderung bei einer der Klassen aus irgend einem Grunde nicht möglich ist. Am Ende des Vorgangs wurde die Operation dann bei einigen Objekten erfolgreich durchgeführt, bei anderen nicht. Es gibt Anwendungen, wo derartiges Verhalten nicht erwünscht ist, sondern Transaktionssemantik erforderlich ist, das heisst: entweder alle Aktionen sind erfolgreich oder keine.

Transaktionen

Zunächst eine kleine Einführung in das, was als *Transaktionssemantik* bezeichnet wird. Eine Transaktion ist definiert als eine Änderung an einem oder mehreren Datenspeicher, die folgenden Eigenschaften aufweist:

- Atomar: Eine Transaktion wird als eine unteilbare Aktion angesehen. Sie läuft entweder gar nicht ab (sie wird abgebrochen, ein Rollback findet statt), oder sie wird komplett ausgeführt (die Transaktion wird committed). Teilausführungen sind verboten.
- Konsistent: Eine Transaktion überführt ein Objekt von einem konsistenten Zustand in einen anderen konsistenten Zustand. Nur während der Ausführung einer Transaktion darf der Zustand kurzzeitig inkonsistent sein.
- Isoliert: Eine Transaktion muß unabhängig von anderen eventuell im System laufenden Transaktionen sein. Interaktionen mit anderen Transaktionen sind nicht erlaubt. Transaktionen müssen also zumindest scheinbar seriell ablaufen.

- Dauerhaft: Wenn eine Transaktion erfolgreich beendet wurde, müssen die am Datenspeicher durchgeführten Veränderungen dauerhaft sein. Dies gilt auch für den Fall eines Systemausfalls.

Diese vier Eigenschaften werden mit ACID (**A**tomic, **C**onsistent, **I**solated, **D**urable) bezeichnet. Transaktionen sind heute in vielen Systemen zwingende Voraussetzung, vor allem in grossen Datenverarbeitungssystemen die daher auch als OLTP Systeme (OnLine Transaction Processing) bezeichnet werden. Beispiele sind Flug- oder Hotelbuchungssysteme oder Banksysteme. Diese Anwendungen basieren in der Regel auf einem Transaktionsmonitor, der eine Laufzeitumgebung für transaktionelle Komponenten zur Verfügung stellt.

Im hier beschriebenen Falls sollen die Transaktionen auf eine Anwendung, also auch einen Prozess beschränkt bleiben.

Implementierungsmöglichkeiten

Die Forderung nach **Atomizität** der Transaktion ist im Allgemeinen die wichtigste. Insbesondere dann, wenn eine Transaktion aus verschiedenen Teilschritten besteht die verschiedene Ressourcen (d.h. von Transaktionen veränderte Objekte) betreffen, ist die Implementierung dieser Forderung nicht trivial. Bei verteilten Systemen ist genau dieses die Hauptaufgabe des Transaktionsmonitors. Um diese Forderung zu realisieren, wird üblicherweise ein zweiphasiger Commitvorgang (Two Phase Commit, 2PC) durchgeführt. Dieser kann folgendermassen beschrieben werden:

In der ersten Phase werden alle beteiligten Ressourcen beauftragt, die Transaktion vorzubereiten. D.h. der Inhalt der auszuführenden Transaktion wird i.d.R. geprüft (ggfs. durch vorläufige Ausführung). Tritt in dieser Phase ein Problem auf, d.h. kann eine Ressource die Transaktion nicht fehlerfrei durchführen, wird die Transaktion bei allen Ressourcen abgebrochen und die evtl. bereits gemachten Änderungen werden rückgängig gemacht. Dies nennt man Rollback.

Wenn keine der Ressourcen ein Problem meldet, so wird Phase zwei, die sog. Commitphase eingeleitet. Alle Ressourcen werden nacheinander beauftragt, ihre Änderungen dauerhaft zu speichern. Damit dieser Prozess richtig funktioniert ist allerdings von jeder Ressource sicherzustellen, dass während der Commit-Phase kein Fehler mehr auftreten kann. Abbildung 1 zeigt diesen Prozess als Sequenzdiagram.

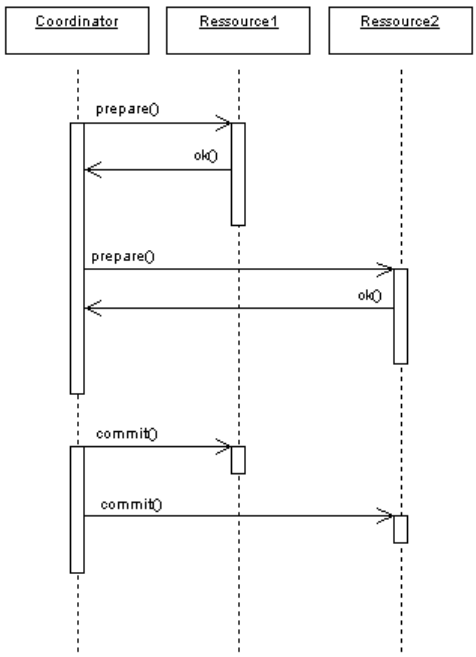

Abbildung 1: Two-Phase Commit Prozess

Eine Möglichkeit, dieses Verhalten der Ressourcen zu implementieren besteht darin, dass die Ressourcen die Änderung wirklich in Phase eins durchführen, und im Falle eines Rollbacks diese Änderungen rückgängig machen. An dieser Stelle kommen Mementos ins Spiel.

Die Forderung nach **Konsistenz** laesst sich dadurch realisieren, dass eine Ressource nachdem sie durch ein Kommando modifiziert wurde auf Konsistenz überprüft wird. Ist dies nicht der Fall, wird die Transaktion abgebrochen.

Die Forderung nach **Isoliertheit** einer Transaktion (die natürlich nur in parallelen Systemen nötig ist) lässt sich in guter Näherung dadurch realisieren, dass die Transaktionen (wie oben bereits angedeutet), wirklich serialisiert werden. Dazu später mehr.

Die Forderung nach **Dauerhaftigkeit** ist im Rahmen des Beispiels hier nicht eindeutig zu beantworten. Im Falle eines Systemausfalls ist bei Änderungen die rein im Speicher ablaufen natürlich nichts mehr zu retten. Wenn man mit dieser Einschränkung leben kann, sind alle Änderungen im Rahmen einer Transaktion selbstverständlich dauerhaft.

Eine Lösung basierend auf Commands und Mementos

Im Artikel zum Thema Mementos im letzten Heft wurde ja bereits erwähnt, wie gut sich Kommandoobjekte und Mementos ergänzen: Bevor ein Kommando ausgeführt wird,

fordert es ein Memento des zu ändernden Objektes an und speichert dieses. Wenn das Kommando rückgängig gemacht werden muss, so wird dem modifizierten Objekt einfach das zwischengespeicherte Mememto zurückgegeben. Basierend auf diesem Mechanismus sollen nun Transaktionen implementiert werden.

Als Voraussetzung gilt, dass ein Kommando nur ein einziges Zielobjekt (Ressource) modifiziert. Die abstrakte Kommandoklasse sieht dann folgendermassen aus (die Ergänzungen zur Kommandoklasse im letzten Artikel sind fett gedruckt).

```
public abstract class Command {
    protected Memento memento;
    protected TransactionResource target;

    public Command(TransactionResource theTarget) {
        target = theTarget;
    }

    public void execute() throws Abort {
        memento = target.createMemento();
        doExecute();
        target.checkInvariants();
    }

    public void undo() {
      memento.activate();
    }

    public abstract void doExecute() throws Abort;
}
```

Es haben sich eigentlich nur zwei Dinge geändert:

- während der Ausführung eines Kommandos (in *doExecute()*) kann das Kommando oder das Zielobjekt die gesamte Transaktion abbrechen, indem es eine *Abort*-Exception wirft. Alle bis dahin im Rahmen der Transaktion durchgeführten Kommandos werden dann rückgängig gemacht.
- Das Zielobjekt eines Kommandos ist nun nicht mehr *MementoCapable*, sondern vom Typ *TransactionResource*. Auch dies ist eine Schnittstelle, sie erbt von *MementoCapable* und fügt die Operation *checkInvariants()* hinzu.

Eine im Rahmen einer Transaktion veränderte Ressource darf nur in einem konsistenten Zustand hinterlassen werden. Um dies sicherzustellen, bietet das Interface *TransactionResource* die Operation *checkInvariants()*.

```
public interface TransactionResource extends MementoCapable {
    public abstract void checkInvariants() throws Abort;
}
```

Eine Ressource muss diese Operation so implementieren, dass eine *Abort*-Exception geworfen wird, wenn das sich das Objekt in einem inkonsistenten Zustand befindet. Auch dann wird die gesamte Transaktion abgebrochen, und wieder ein konsistenter Zustand – der alte nämlich – wiederhergestellt. Die Operation *execute()* verdeutlicht dies:

```
public void execute() throws Abort {
    memento = target.createMemento();
    doExecute();
    target.checkInvariants();
}
```

Zunächst wird ein Memento des modifizierten Zielobjektes angefordert. Dann wird die eigentliche Änderung am Zielobjekt ausgeführt (*doExecute()*). Bei Problemen kann diese Operation die Transaktion abbrechen (durch Werfen von *Abort*). Nach Ausführen der Änderung wird die Operation *checkInvariants()* aufgerufen, um die Konsistenz des Zielobjekts zu prüfen. Auch diese Operation kann die Transaktion abbrechen. Diese Lösung hat den Vorteil, dass die Überprüfung der Konsistenz automatisch erfolgt und nicht in jedem Kommando neu implementiert werden muss – die Verantwortung liegt also beim Ersteller der Zielklasse und nicht beim Ersteller der konkreten Kommandos.

Bevor wir den Kern des Systems – die Klasse *Transaction* – beschreiben, soll die Verwendung des Transaktionsmechanismus verdeutlicht werden. Dazu legen wir zunächst zwei Instanzen der Klasse *TestObject* an, die das Interface *TransactionResource* implementiert.

```
TestObject resource1 = new TestObject( "resource1", 1  );
TestObject resource2 = new TestObject( "resource2", 2 );
```

Dann wird ein Transaktionsobjekt angelegt, und zwei Kommandos hinzugefügt, die jeweils eines der beiden *TestObject*s modifizieren:

```
Transaction t = new Transaction();
t.addAction( new ModifyCommand( resource1 ) );
t.addAction( new ModifyCommand( resource2 ) );
```

Dann wird die Transaktion ausgeführt. Wenn eine *Abort*-Exception gefangen wird, so wird automatisch ein Rollback durchgeführt:

```
try {
    t.execute();
} catch ( Abort a ) {
    // Transaktion fehlgeschlagen!
    // Rollback wurde bereits automatisch
    // durchgeführt.
}
```

Die Transaktionsklasse

Jetzt fehlt nur noch die Transaktion selbst. Eine Transaktion muss sich selbstverständlich die auszuführenden Aktionen merken und für den Fall eines Aborts muss zum Zwecke

des Rollbacks die Nummer der letzten durchgeführten Aktion bekannt sein. Der Konstruktor initialisiert die entsprechenden Attribute:

```
public class Transaction {
    protected List actions;
    protected int lastExecutedIndex;

    public Transaction() {
        super( null );
        actions = new ArrayList();
        lastExecutedIndex = -1;
    }
```

Das Hinzufügen von Aktionen bedarf wohl keiner weiteren Erklärung:

```
    public void addAction(Command c) {
        actions.add( c );
    }
```

Das Ausführen der Transaktion in *execute()* ist schon interessanter. Es werden die vorhandenen Kommandos der Reihe nach abgearbeitet. Wenn das Ausführen eines Kommandos (*execute()*) einen *Abort* hervorruft, wird die Methode beendet und in *lastExecutedIndex* steht die Nummer des zuletzt versuchten (also des gescheiterten) Kommandos:

```
    public void execute() throws Abort {
        try {
            for ( lastExecutedIndex=0;
                  lastExecutedIndex<actions.size();
                  lastExecutedIndex++ ) {
                Command c = (Command)actions.
                     get( lastExecutedIndex );
                c.execute();
            }
        } catch ( Abort abort ) {
            rollback();
            throw( abort );
       }
    }
```

Das Rollback der Transaktion ruft auf allen bisher ausgeführten Kommandos *undo()* auf:

```
    protected void rollback () {
        for ( int i=0; i<=lastExecutedIndex; i++ ) {
            Command c = (Command)actions.get(i);
            c.undo();
        }
    }
}
```

Dies ist schon fast alles. Was noch fehlt ist die Serialisierung von parallel ablaufenden Transaktionen. Zur Vereinfachung nehmen wir an, dass im gesamten System zu einer Zeit

immer nur eine Transaktion ablaufen soll. Man kann dann zur Serialisierung von in verschiedenen Threads laufenden Transaktionen einen globale Semaphore verwenden. Diese hat hier den Namen *TransactionSerializer* und sieht folgendermassen aus:

```
public class TransactionSerializer {

    protected static TransactionSerializer instance;
    protected boolean locked = false;

    public static TransactionSerializer instance() {
        if ( instance == null ) {
            instance = new TransactionSerializer();
        }
        return instance;
    }

    public synchronized void lock() {
        while ( locked ) {
            try {
                wait();
            } catch ( Exception ex ) {}
        }
    }

    public synchronized void unlock() {
        locked = false;
        notifyAll();
    }

}
```

Die Klasse ist ein *Singleton* (siehe [1]), es kann also systemweit nur eine einzige Instanz geben. Wenn ein Thread auf dieser Instanz die Operation *lock()* aufruft, so wird das Flag *locked* auf *true* gesetzt – falls es nicht schon *true* ist. In diesem Fall wird der Thread des Aufrufers blockiert. Ruft ein (anderer) Thread *unlock()* auf, wird das Flag wieder auf *false* gesetzt und alle wartenden Threads werden benachrichtigt – sie können also ein weiteres Mal versuchen, die Instanz zu locken.

Damit die Transaktionen diese Semaphore benutzen, muss die Operation *execute()* etwas angepasst werden: Vor dem Start der Transaktion, muss der *TransactionSerializer* gelockt werden, nach erfolgreichem Abschluss oder nach einem Rollback muss die Sperre wieder aufgehoben werden:

```
public void execute() throws Abort {
    try {
        TransactionSerializer.instance().lock();
        for ( lastExecutedIndex=0;
                lastExecutedIndex<actions.size();
                lastExecutedIndex++ ) {
```

```
                Command c = (Command)actions.
                        get( lastExecutedIndex );
                c.execute();
            }
        } catch ( Abort abort ) {
            rollback();
            throw( abort );
        } finally {
            TransactionSerializer.instance().unlock();
        }
    }
```

Übrigens hat dieses Vorgehen noch einen zweiten Effekt: Dadurch, dass zwei Transaktionen immer nacheinander ausgeführt werden, und der Rollback im Rahmen desselben Locks durchgeführt wird, kann keine andere Transaktion ausgeführt werden, bevor dass Rollback vollständig durchgeführt wurde – dies dient der Forderung nach Konsistenz.

Verbesserungsmöglichkeiten

Wie immer ist diese Lösung nur ein Startpunkt, der viel Potential für Verbesserungen bietet.

Zum einen ist es natürlich nicht nötig, dass für jede Aktion ein Memento erzeugt wird, wenn diese Aktionen dasselbe Zielobjekt modifizieren. Um dies zu vermeiden, könnte man die Anforderung und Aufbewahrung des Mementos in die Verantwortung der Transaktionsklasse stellen. Diese würde dann vor dem Ausführen der Aktionen von allen Zielobjekten ein Memento anfordern und dies speichern (dazu müssten die Kommandos in der Lage sein, ihr Zielobjekt zurückzuliefern.) Beim Rollback würde ein Zielobjekt dann nur einmal mit einem zu verwendenden Memento versorgt.

Der zweite Punkt betrifft Parallelität und Serialisierung: Im oben beschriebenen Fall kann im System immer nur eine Transaktion zu einer Zeit ausgeführt werden – auch wenn diese Transaktionen verschiedene Zielobjekte modifizieren. Dies ist sicherlich unnötig restriktiv. Man sollte also realistischerweise eine Semaphore pro Zielobjekt haben, denn dann werden Transaktionen nur serialisiert, wenn sie die gleichen Zielobjekte modifizieren. An dieser Stelle sind noch zwei weitere Dinge erwähnenswert: Zum einen muss eine modifizierte Ressource weiterhin für die gesamte Transaktion gelockt werden, und zum anderen können hierbei Deadlocks auftreten – Vorsicht ist also geboten.

Der Code für diesen Artikel findet sich wie immer auf der Begleit-CD oder auf www.voelter.de.

Referenzen

[1] Gamma, Helm, Johnson, Vlissides, *Entwurfsmuster*, Addison-Wesley 1995