Ein Scheduler basierend auf Commands und Strategien

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

Eine gängige Anforderung bei der Anwendungsentwicklung ist, bestimmte Aktionen regelmäßig und wiederholt auszuführen. Dabei können verschiedene Aktionen in verschiedenen Abständen wiederholt ablaufen. Üblicherweise wird zur Koordination dieser Aktionen ein Scheduler (englisch für Planer) eingesetzt, dem die auszuführenden Aktionen übergeben werden, und der sich dann darum kümmert, sie zu den richtigen Zeiten auszuführen. Der vorliegende Artikel beschreibt einen einfachen Scheduler, der auf dem Command- und dem Strategie-Muster beruht.

Grundlegenden Anforderungen

Ein Scheduler hat mindestens zwei Stellen, an denen Flexibilität notwendig ist: Zunächst wäre das natürlich die auszuführende Aktion selbst: Ein Scheduler, dem man die auszuführenden Aktionen nicht mitteilen kann, wäre sinnlos. Aktionen lassen sich sehr schön mittels des ja schon in den vorherigen Artikeln beschriebenen Kommandomusters kapseln. Kommandos sind Unterklassen einer abstrakten Oberklasse, die alle durch den Aufruf einer Operation *execute()* ausgeführt werden können ohne das konkrete Kommando zu kennen.

Desweiteren sollte bei einem Scheduler parametrierbar sein, ob und wie die auszuführenden Aktionen wiederholt werden sollen. Verschiedene Möglichkeiten sind vorstellbar (Darauf wird weiter unten im Detail eingegangen). Derartige Flexibilität lässt sich leicht mit Hilfe des Strategiemusters implementieren.

Das grundlegende Design

Abbildung 1 zeigt das grundlegende Design des Schedulers.

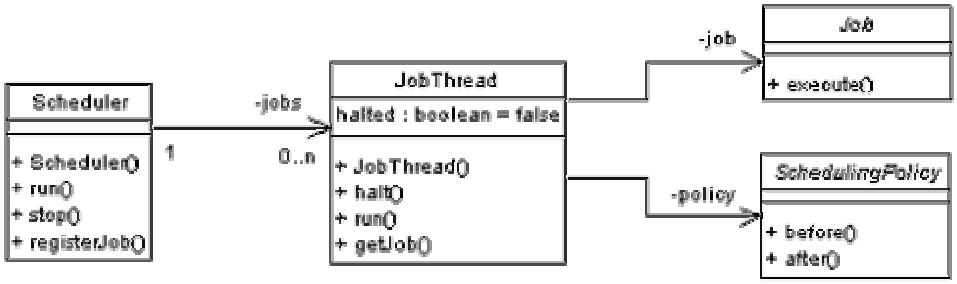

Abbildung 1: Die Struktur des Schedulers. Job und SchedulingPolicy sind die primären Eingriffspunkte der Anwendungsprogrammierer.

Aus Sicht des späteren Anwenders ist der *Job* das wichtigste Element der Klassenstruktur. Die Klasse *Job* ist entsprechend dem Kommandomuster implementiert, das bedeutet, dass anwendungsspezifische Jobs realisiert werden, indem von der abstrakten Klasse *Job* abgeleitet, und die Operation *execute()* anwendungsspezifisch überschrieben wird. Das Ausführen eines Jobs wird, - wie immer bei Kommandoobjekten, - durch Aufrufen von *execute()* eingeleitet.

Der zweite wichtige Punkt ist die Art und Weise, wie und wann der *Job* ausgeführt wird. Dazu dient die Klasse *SchedulingPolicy*. Wie diese arbeitet, wird weiter unten erläutert.

Für das konkrete Ausführen eines *Job*s im Rahmen des Schedulers ist die Klasse *JobThread* verantwortlich. Wenn ein *Job* beim *Scheduler* registriert wird, dann wird dabei von der Anwendung eine *SchedulingPolicy* mitgeliefert. Der *Scheduler* erzeugt daraufhin einen *JobThread*, der eine Referenz auf den auszuführenden *Job* und auf die *SchedulingPolicy* besitzt. Der *JobThread* delegiert alle Operationen die das Scheduling betreffen an die *SchedulingPolicy*, so wie dies das Strategiemuster vorschlägt.

SchedulingPolicies

Die Klasse *SchedulingPolicy* besitzt zwei Operationen, *before()* und *after()*. Diese beiden Operationen werden im Rahmen der *run()*-Operation des *JobThreads* aufgerufen und implementieren das Verhalten des Schedulers bezüglich eines Jobs. Die *run()*-Operation ist prinzipiell folgendermassen implementiert:

```
class JobThread extends Thread {
  // ...
  public void run()
  {
    do {
      if (policy.before( this ) == SchedulingPolicy.OK ) {
       job.execute();
       policy.after( this, true );
      } else // ...
```

```
      } while ( !halted );
    }
  }
```

Zunächst wird die Operation *before()* der *SchedulingPolicy* aufgerufen. Der Rückgabewert dieser Operation bestimmt, ob der *Job* überhaupt ausgeführt wird; möglicherweise gibt es Gründe, den *Job* nicht auszuführen, Details siehe unten. Gegebenenfalls wird dann der *Job* an sich ausgeführt (*job.execute()*). Danach wird die *after()*-Operation der *SchedulingPolicy* aufgerufen. Einer ihrer Parameter ist ein boolescher Wert der angibt, ob der *Job* ausgeführt wurde oder nicht.

Um die erläuterten Konzepte zu verdeutlichen, hier nun einige Beispiele für *SchedulingPolicies*. Die einfachste Policy ist, einen *Job* nur einmal auszuführen, und zwar direkt beim Start des Schedulers. Die Klasse *OnceSP* (SP steht für *SchedulingPolicy*) sieht also wie folgt aus:

```
public class OnceSP extends SchedulingPolicy {
  public int before(JobThread jt) {
      // OK zurückliefern, Job wird also ausgeführt
    return SchedulingPolicy.OK;
  }
  public void after(JobThread jt, boolean executed) {
      // nach erster Ausführung Job anhalten
    jt.halt();
  }
}
```

Kern ist die Operation *after()*, die den entsprechenden *JobThread* beendet. D.h. der Job wird einmal ausgeführt – *before()* liefert *OK* – und dann nie wieder.

Etwas interessanter – und im Rahmen eines Schedulers auch deutlich sinnvoller – ist die Klasse *DelaySP*. Diese wartet nach jeder Ausführung des Jobs eine definierte, einstellbar Zeit bevor sie den Job das nächste Mal startet:

```
public class DelaySP extends SchedulingPolicy {
   private long delayTime;

   public DelaySP(long delay) {
     delayTime = delay;
   }

   public int before(JobThread jt) {
     return SchedulingPolicy.OK;
   }

   public void after(JobThread jt, boolean executed) {
     try {
       jt.sleep( delayTime );
     } catch ( Exception ex ) {
       // ...
```

```
      }
    }
  }
```

Das Prinzip ist einfach: Die Operation *after()* lässt den entsprechenden *JobThread* so lange schlafen, wie es der Parameter *delayTime* vorgibt. Natürlich hat dies den Nachteil, dass die Zeitspanne, die das Ausführen des Jobs selbst benötigt, unberücksichtigt bleibt. Wird also ein Delay von 1000ms angestrebt und benötigt der Job selbst 20ms, so verzögert sich jede weitere Ausführung um eben diese 20ms. Dies kann durchaus gewollt sein, nämlich dann, wenn zwischen dem Ende der einen Ausführung und dem Anfang der nächsten wirklich genau diese 1000ms eingehalten werden müssen. Oft ist die Anforderung jedoch einen andere: Der Job soll regelmässig in bestimmten Abständen gestartet werden. Dies erledigt die Klasse *IntervalSP*:

```
public class IntervalSP extends SchedulingPolicy {
  private long interval;
  private long startTime;
  private long nextTime;

  public IntervalSP(long inter) {
   interval = inter;
    startTime = System.currentTimeMillis();
    nextTime = startTime;
  }

  public int before(JobThread jt) {
    nextTime += interval;
    return SchedulingPolicy.OK;
  }

  public void after(JobThread jt, boolean executed) {
    long dt = nextTime - System.currentTimeMillis();
    try {
      jt.sleep( dt );
    } catch ( Exception ex ) {
      // ...
    }
  }
}
```

Im Konstruktor merkt sich die Klasse, wann sie das erste mal ausgeführt wurde, die sogenannte *startTime*. Der jeweils folgende Ausführungszeitpunkt *nextTime* wird dadurch bestimmt, dass vor der Ausführung des Jobs, also in *before()*, zur aktuellen *nextTime* das gewünschte Intervall addiert wird. Nach der Ausführung des *Job*s in *after()* wird dann die noch nötige Pause berechnet, indem die Differenz aus der nächsten Ausführungszeit (also *nextTime*) und der aktuellen Zeit bestimmt wird. Dadurch wird die Ausführungszeit des Jobs (und evtl. einmal auftretenden Ungenauigkeiten) automatisch kompensiert.

Abbildung 2 verdeutlicht den Unterschied der beiden *SchedulingPolicies*. Während im Falle von *DelaySP* die Länge der Pausen konstant ist und sich der Ausführungszeitpunkt (aufgrund der Ausführungsdauer) gegenüber der Zeitachse verschiebt, sorgt *IntervalSP* dafür, dass der *Job* immer zu den geplanten Zeiten gestartet wird, und sich der Ausführungszeitpunkt unabhängig von der Ausführungsdauer gegenüber der Zeitachse nicht verschiebt (solange die Dauer kleiner ist also das Intervall).

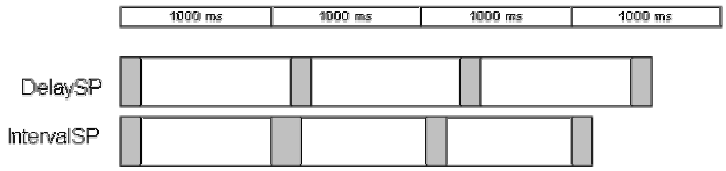

Abbildung 2: Vergleich von DelaySP und IntervalSP. DelaySP hält die Pausen ein, IntervalSP in dafür pünktlich.

Anwendung des Schedulers

Um ein Gefühl für die Anwendung des Schedulers zu bekommen, soll diese hier zunächst veranschaulicht werden. Die Anwendung ist recht einfach. Zunächst legt man sich eine Instanz der Klasse *Scheduler* an.

```
Scheduler scheduler = new Scheduler();
```

Bei diesem Scheduler kann man nun *Job*s zusammen mit ihrer *SchedulingPolicy* registrieren:

```
scheduler.registerJob( new PrintlnJob("Hello World" ), new
        IntervalSP(100) );
scheduler.registerJob( new PrintlnJob("Hello World" ), new
        DelaySP(200) );
```

Um alle diese Jobs zu starten, muss die *run()* Operation auf dem *Scheduler* aufgerufen werden.

```
s.run();
```

Beendet wird das ganze mittels

```
s.halt();
```

Quality of Service

Die Anwendung eines derartigen Schedulers in Java hat natürlich ihre Einschränkungen. Erstens können keine beliebig kleinen Intervalle angegeben werden, da der Threading-Mechanismus nicht beliebig genau ist. Desweiteren ist Java nicht echtzeitfähig, d.h. eine bestimmter Programmblock kann unterschiedlich lange dauern, wenn er mehrmals ausgeführt wird. Dies liegt in erster Linie an Java's Speicherbereinigungsmechanismus (Garbage Collection). Es ist nicht vorherbestimmbar, wann und ob der Garbage Collector seine Arbeit startet und damit andere Threads bremst. Auch durch die verschiedenen Just-

In-Time Compilierungsmechanismen kann es bei kleinen Intervallen bei verschiedenen JVMs zu verschiedenen Ausführungsdauern kommen.

Die oben genannten Einschränkungen können in bestimmten Anwendungen ein ernsthaftes Problem sein, in anderen sind sie möglicherweise tolerierbar. Jedenfalls sollte ein Scheduler die Möglichkeit haben, auf ein verspätetes Event gebührend zu reagieren. Echtzeitsysteme reagieren üblicherweise mit einer der folgenden Alternativen:

- Eine Alarmmeldung wird protokolliert, die besagt, dass das System zu langsam wird, das Events verspätet sind. Die für den Zeitpunkt geplante Aktion wird allerdings trotzdem (verspätet) ausgeführt.
- Die geplante Aktion wird nicht ausgeführt, da es besser ist, eine Aktion gar nicht auszuführen als sie verspätet auszuführen.
- Das gesamte System (also der Scheduler) wird angehalten.

Alle diese Mechanismen lassen sich mittels der oben eingeführten Klassen problemlos implementieren. Basis ist die Feststellung, ob eine Aktion zu spät ausgeführt wird, oder nicht.

```
public class IntervalLimitSP extends IntervalSP {
  private long maxDeviation;
  private int reactionOnDeviation;

  public IntervalLimitSP(long inter, int maxDev, int react )   {
    super( inter );
    maxDeviation = maxDev;
    reactionOnDeviation = react;
  }

  public int before(JobThread jt) {
    long currentTime = System.currentTimeMillis();
    if ( currentTime > (nextTime + maxDeviation) ) {
      return reactionOnDeviation;
    }
    if ( currentTime < (nextTime - maxDeviation) ) {
      return reactionOnDeviation;
    }
    else return super.before( jt );
  }
}
```

Die obige Klasse bekommt zusätzlich zum eigentlichen Intervall noch zwei weitere Parameter im Konstruktor übergeben:

- *maxDev* gibt die maximal erlaubte Abweichung von der Sollausführungszeit an.

- *reaction* ist eine Konstante, die angibt, wie das System reagieren soll, wenn die Abweichung überschritten wird.

Die folgenden Werte für die Konstante *reaction* sind definiert:

- *SchedulingPolicy.OK* führt die Aktion ganz normal aus.
- *SchedulingPolicy.DO_NOT_EXEC* führt die Aktion dieses eine Mal nicht aus.
- *SchedulingPolicy.STOP_JOB* beendet den Job, das heisst, wenn die Aktion einmal verspätet ist, wird sie nie wieder ausgeführt.
- *SchedulingPolicy.STOP_SCHEDULER* beendet den Scheduler, es wird also keiner der Jobs mehr ausgeführt.

Um dieses Verhalten zu implementieren, muss die *run()* Operation der *JobThread*s dieses Flag entsprechend abfragen. Das folgende Stück Code zeigt die komplette Implementierung dieser Operation:

```
class JobThread extends Thread {
  // ..
  public void run() {
    do {
      int retVal = policy.before( this );
      if ( retVal == SchedulingPolicy.OK ) {
          // Job ausführen wenn alles OK ist, und dann
          // after mit true aufrufen
        job.execute();
        policy.after( this, true );
      } else if ( retVal == SchedulingPolicy.STOP_JOB ) {
          // aktuellen T. anhalten, after wird nicht aufgerufen
        halt();
      } else if ( retVal == SchedulingPolicy.STOP_SCHEDULER ) {
          // ganzen Scheduler anhalten
        scheduler.stop();
      } else {
          // sonst (also bei DO_NOT_EXEC) Job nicht ausführen,
          // aber after() mit false aufrufen
        policy.after( this, false );
      }
    } while ( !halted );
  }
  // ..
}
```

Zusammenfassung und Ausbaumöglichkeiten

Wenn man mit den oben beschriebenen Einschränkungen leben kann, stellt dieser einfache Scheduler schon ein ganz nützliches Tool dar. Erweiterungen sind natürlich wie immer möglich.

Ein Problem kann sein, dass pro Aktion ein eigener Thread erzeugt wird. Der Thread schläft zwar während des größten Teils seiner Existenz, jedoch benötigen Threads immer Ressourcen, und bei einer grossen Anzahl an Aktionen kann dies zu Problemen führen. Das lässt sich z.B. dadurch umgehen, dass man einem Thread mehrere Jobs zuweist, die (näherungsweise) zur gleichen Zeit ablaufen sollen. Dazu kann man das Composite Muster auf das Command anwenden. Man erzeugt damit praktisch "Makros", die dann dem Scheduler zur Ausführung übergeben werden.

Eine weitere Erweiterungsmöglichkeit kann das Thema Synchronisation sein. Möglicherweise dürfen bestimmte Jobs nicht gleichzeitig ablaufen. Dies kann natürlich durch Einstellung ihrer Timing-Parameter erzwungen werden, in manchen Situationen ist aber explizite Synchronisation unumgänglich. Dies kann mit den üblichen Synchronisationsmechanismen wie z.B. Semaphoren oder Monitore erreicht werden. Java bietet dafür mittels *wait()* und *notify()* bereits eingebaute Mechanismen.

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