**A01**

Nebenläufige Roboterfabrik

Kalauner Paul, Ritter Mathias

**Zusammenfassung**

Simulation einer Roboterfabrik mit mehreren Threads in Java

Inhaltsverzeichnis

Aufgabenstellung/Requirementsanalyse 3

Zeitabschätzung & Zeitaufzeichnung 5

Planung 5

Implementierung 5

Testen 6

Gesamt 6

Klassendiagramm 7

Implementierung 8

Simulation 9

Sekretariat 9

Lagermitarbeiter 9

Lager 9

Lieferant 9

Montagemitarbeiter 10

Threadee 10

Teil 10

Teiltyp 10

Watchdog 10

WatchableWorker 10

Systemtests 11

Test der Laufzeit 11

Test der Argument-Validierung 11

Lessons Learned 12

Verwenden von mehreren Threads 12

Synchonisieren von Threads 12

Apache Commons CLI 13

Quellen 14

Apache Log4J 14

Apache Commons CLI 14

Mockito 14

JUnit 14

# Aufgabenstellung/Requirementsanalyse

Es soll eine Spielzeugroboter-Fabrik simuliert werden. Die einzelnen Bestandteile des Spielzeugroboters (kurz Threadee) werden in einem Lager gesammelt. Dieses Lager wird als Verzeichnis und die einzelnen Elementtypen werden als Files im Betriebssystem abgebildet. Der Lagermitarbeiter verwaltet regelmäßig den Ein- und Ausgang des Lagers um Anfragen von Montagemitarbeiter und Kunden zu beantworten. Die Anlieferung der Teile erfolgt durch Ändern von Files im Verzeichnis, eine Lagerung fertiger Roboter ebenso.  
  
Ein Spielzeugroboter besteht aus zwei Augen, einem Rumpf, einem Kettenantrieb und zwei Armen.    
Die Lieferanten schreiben ihre Teile ins Lager-File mit zufällig (PRNG?) erstellten Zahlenfeldern. Die Art der gelieferten Teile soll nach einer bestimmten Zeit gewechselt werden.  
  
Die Montagemitarbeiter müssen nun für einen "Threadee" alle entsprechenden Teile anfordern und diese zusammenbauen. Der Vorgang des Zusammenbauens wird durch das Sortieren der einzelnen Ganzzahlenfelder simuliert. Der fertige "Threadee" wird nun mit der Mitarbeiter-ID des Monteurs versehen.  
  
Es ist zu bedenken, dass ein Roboter immer alle Teile benötigt um hergestellt werden zu können. Sollte ein Monteur nicht alle Teile bekommen, muss er die angeforderten Teile wieder zurückgeben um andere Monteure nicht zu blockieren. Fertige "Threadee"s werden zur Auslieferung in das Lager zurück gestellt.  
  
Alle Aktivitäten der Mitarbeiter muss in einem Logfile protokolliert werden. Verwenden Sie dazu Log4J [1].  
  
Die IDs der Mitarbeiter werden in der Fabrik durch das Sekretariat verwaltet. Es dürfen nur eindeutige IDs vergeben werden. Das Sekretariat vergibt auch die eindeutigen Kennungen für die erstellten "Threadee"s.  
  
Beachten Sie beim Einlesen die Möglichkeit der Fehler von Files. Diese Fehler müssen im Log protokolliert werden und entsprechend mit Exceptions abgefangen werden.

**Tipps und Tricks**

Verwenden Sie (optional) für die einzelnen Arbeiter das ExecutorService mit ThreadPools. Achten Sie, dass die Monteure nicht "verhungern". Angeforderte Ressourcen müssen auch sauber wieder freigegeben werden.

**Beispiel für Teile-Files**  
-- "auge.csv"  
Auge,11,24,3,4,25,6,8,8,9,10,11,12,13,14,15,16,17,18,195,5  
Auge,91,62,3,4,54,6,7,8,9,10,11,12,13,14,15,16,17,18,119,32  
Auge,91,62,3,4,54,6,7,8,9,10,11,12,13,14,15,16,17,18,119,520  
  
-- "rumpf.csv"  
Rumpf,91,62,3,4,54,6,7,8,9,10,11,12,13,14,15,16,17,18,119,21

**Beispiel für Threadee-File**  
-- "auslieferung.csv"  
Threadee-ID123,Mitarbeiter-ID231,Auge,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,Auge,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,Rumpf,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,Kettenantrieb,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,Arm,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,Arm,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20  
Threadee-ID124,Mitarbeiter-ID231,Auge,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,Auge,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,Rumpf,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,Kettenantrieb,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,Arm,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,Arm,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20  
  
  
**Ausführung**  
  
Zu bedenken sind die im Beispiel angeführten Argumente. Diese können mit eigenem Code oder mit einer CLI-Library implementiert werden (z.B. [2]).  
Alle Argumente sind verpflichtend und die Anzahl muss positiv sein. Die obere Grenze soll sinnvoll festgelegt werden. Vergessen Sie auch nicht auf die Ausgabe der Synopsis bei einer fehlerhaften Eingabe! Sollten Sie zusätzliche Argumente benötigen sind diese erst nach einer Rücksprache implementierter.  
  
java tgm.sew.hit.roboterfabrik.Simulation --lager /verzeichnis/zum/lager --logs /verzeichnis/zum/loggen --lieferanten 12 --monteure 25 --laufzeit 10000

# Zeitabschätzung & Zeitaufzeichnung

Schätzung: 4h/Person, insgesamt 8h

## Planung

|  |  |  |
| --- | --- | --- |
| **Aufgabe** | **Name** | **Zeitaufwand** |
| Requirementsanalyse | Kalauner Paul, Ritter Mathias | 45 Minuten |
| UML-Diagramm | Kalauner Paul, Ritter Mathias | 100 Minuten |
| UML-Redesign | Kalauner Paul, Ritter Mathias | 60 Minuten |
| Fertigstellen der Dokumentation | Kalauner Paul, Ritter Mathias | 65 Minuten |

## Implementierung

|  |  |  |
| --- | --- | --- |
| **Aufgabe** | **Name** | **Zeitaufwand** |
| Git und Maven eingerichtet | Kalauner Paul | 15 Minuten |
| Klassen und Methoden erstellt | Ritter Mathias | 15 Minuten |
| Überprüfung der Konsolenargumente | Kalauner Paul | 20 Minuten |
| Lager, Lagermitarbeiter, Lieferant, Sekretariat | Kalauner Paul | 70 Minuten |
| Überprüfung der Teileanzahl | Ritter Mathias | 15 Minuten |
| Einlagern der Teile | Ritter Mathias | 20 Minuten |
| Bereitstellen der Teile | Ritter Mathias | 30 Minuten |
| Auslesen der Files | Kalauner Paul | 30 Minuten |
| Zusammenbauen eines Threadees | Ritter Mathias | 40 Minuten |
| Letzte Zeile aus File löschen Bug Fix, kleines Redesign | Kalauner Paul | 70 Minuten |
| Schreiben in die Files implementiert | Ritter Mathias | 35 Minuten |
| Bugfixing | Ritter Mathias | 20 Minuten |

## Testen

|  |  |  |
| --- | --- | --- |
| **Aufgabe** | **Name** | **Zeitaufwand** |
| Klassen Lager und Lagermitarbeiter | Ritter Mathias | 65 Minuten |
| Klassen Lieferant, Montagemitarbeiter, Teil, Sekretariat, Threadee | Ritter Mathias | 150 Minuten |
| Simulation | Kalauner Paul | 35 Minuten |
| Systemtests | Kalauner Paul | 20 Minuten |

## Gesamt

Ritter Mathias: 8,75 Stunden

Kalauner Paul: 6,6 Stunden

Gesamtaufwand: 15 Stunden 20 Minuten

# 

# Klassendiagramm

Das Klassendiagramm befindet sich auch als png-File im Dokumenteordner des Jar-Files.

# Macintosh HD:Users:Mathias:4AHITT:SEW:A01:Class Diagram2.pngImplementierung

Zuerst wurde Maven-Projekt mit folgenden Dependencies erstellt:

<dependencies>

<dependency>

<groupId>org.apache.logging.log4j</groupId>

<artifactId>log4j-api</artifactId>

<version>2.0.2</version>

</dependency>

<dependency>

<groupId>org.apache.logging.log4j</groupId>

<artifactId>log4j-core</artifactId>

<version>2.0.2</version>

</dependency>

<dependency>

<groupId>commons-cli</groupId>

<artifactId>commons-cli</artifactId>

<version>1.2</version>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.11</version>

</dependency>

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-all</artifactId>

<version>1.9.5</version>

</dependency>

</dependencies>

Log4j: Wird zum Loggen benötigt

Commons-CLI: Wird zur Überprüfung der Konsolenargumente benötigt

JUnit: Wird zur Durchführung der Unit-Tests benötigt

Mockito: Wird ebenfalls zur Durchführung der Unit-Tests benötigt

Auf Github wurde ein Private-Repository erstellt und das Projekt wurde in Eclipse entsprechend konfiguriert.

Danach wurden alle Klassen und Methoden wie im UML-Klassendiagramm dargestellt implementiert:

## Simulation

* Konfigurieren des Loggers
  + Legt den Speicher-Ort für die Log-Files fest
  + Erstellt die Logfiles
* Implementierung der main-Methode
  + Nimmt die Konsolenargumente entgegen
  + Gibt die Konsolenargumente zur Überprüfung weiter
* Überprüfung der Konsolenargumente
  + Max. 1000 Monteure/Lieferanten
  + Max. Laufzeit von 3600000ms (1 Stunde)

## Sekretariat

* Generieren von IDs für
  + Mitarbeiter
  + Threadees
* Empfangen von Lieferungen
  + Erhält Lieferungen von einem Lieferanten
  + Weitergabe an den Lagermitarbeiter

## Lagermitarbeiter

* Einlagerung empfangener Teile
  + Schreibt die Teile in das entsprechende File
  + Erhöht den Lagerstand im Lager
* Bereitstellung von Teilen für einen Threadee
  + Überprüft die Lagerstände
  + Löscht die Teile aus den entsprechenden Files
  + Übergibt die Teile einem Mitarbeiter
* Einlagerung fertiger Threadees
  + Schreibt die Threadees in ein File

## Lager

* Beinhält Informationen über die Anzahl der vorhandenen Teile, damit nicht immer aus den Files gelesen werden muss

## Lieferant

* Liefert Teile
  + Generiert Teile
  + Übergibt die Teile dem Sekretariat

## Montagemitarbeiter

* Zusammenbauen eines Threadees
  + Fordert Teile vom Lagermitarbeiter an
  + Nimmt Teile entgegen
  + Baut den Threadee zusammen
  + Übergibt den Threadee dem Lagermitarbeiter

## Threadee

* Repräsentiert einen Roboter bestehend aus
  + 2 Augen
  + 2 Armen
  + Rumpf
  + Kettenantrieb

## Teil

* Repräsentiert ein Teil bestehend aus
  + Dem Teiltyp, welcher von einem Enum vorgegeben wird
  + Einer Kette von Zahlen

## Teiltyp

* Repräsentiert einer der folgenden Typen
  + Auge
  + Arm
  + Rumpf
  + Kettenantrieb

## Watchdog

* Stoppen von Threads
  + Lieferanten
  + Monteuere

## WatchableWorker

* Schreibt die vom Watchdog aufgerufene stop-Methode vor

# Systemtests

## Test der Laufzeit

Bei diesem Test wird sichergestellt, ob die angegebene Laufzeit auch wirklich eingehalten wird.

**Angegebene Argumente:** --lager /Pfad/ --logs /Pfad/ --lieferanten 12 –monteure 25 --laufzeit 10000

**Erwartete Laufzeit des Programms:** 10 Sekunden

**Tatsächliche Laufzeit:** 12 Sekunden

Der Test ist aus folgenden Gründen trotzdem erfolgreich:

* Initialisierung der Files wird nicht in die Laufzeit einberechnet
* Threads werden nach Ablauf der Laufzeit noch fertig bearbeitet

## Test der Argument-Validierung

Bei diesem Test wird überprüft, ob bei Eingabe falscher Argumente die Simulation nicht gestartet und eine Fehlermeldung angezeigt wird.

**Angegebene Argumente:** --lager /Pfad/ --logs /Pfad/ --lieferanten 12 –monteure 25

(Hier wurde das Laufzeit Argument nicht angegeben)

**Erwartetes Verhalten:** Anzeige einer Fehlermeldung und Beendigung der Simulation

**Tatsächliches Verhalten:** Es wurde eine Fehlermeldung angezeigt und das Programm wurde beendet.

# Lessons Learned

## Verwenden von mehreren Threads

Mehrere Threads können parallel Programmcode abarbeiten, um das gesamte Programm zu beschleunigen (Vorausgesetz, der Prozessor verfügt über mehrere Kerne).

Threads können auf 2 verschiedene Arten implementiert werden:

Entweder eine Klasse implementiert (implements) die Klasse Thread oder erbt (extends) davon.

Die Verwendung von implements ist sinnvoller, da sonst die Vererbungshirarchie blockiert wird.

Ein Thread wird durch aufrufen von run() gestartet. Er sollte nicht durch ein Interrupt beendet werden, da Interrupts eine Ausnahme darstellen und auch anderswertig ausgelöst werden können. Stattdessen sollte ein Watchdog, der alle Threads überwacht, sie z.B. durch setzen einer boolean-Variable stoppen.

## Synchonisieren von Threads

Das synchronized Schlüsselwort stellt sicher, dass immer nur 1 Thread gleichzeitig auf eine Methode oder einen bestimmten Block zugreift.

Der Bereicht, indem sich immer nur ein Thread befinden darf, heißt kritischer Abschnitt.

Man sollte solche Bereiche als kritische Abschnitte definieren, in denen auf eine bestimme Ressource zugegriffen wird. Sonst kann es zu Problemen kommen, wenn z.B. ein Thread etwas aus einem File löscht, während ein anderer liest.

## Apache Commons CLI

Mit den Apache Commons CLI können die angegebenen Konsolenargumente einfach validiert werden.

Bsp.:

Alle Argumente müssen zu Options hinzugefügt werden:

Options options = **new** Options();

options.addOption(OptionBuilder.*hasArg*(**true**).*isRequired*().*create*("lager"));

//...

Anschließend können die Argumente geparsed werden:

CommandLineParser parser = new BasicParser();

CommandLine cmd = parser.parse(options, args);

Die parse Methode wirft eine Exception, falls die angegebenen Argumente ungültig sind.

Außerdem kann ganz einfach eine –-help Funktion implementiert werden:

HelpFormatter hf = new HelpFormatter();

hf.printHelp("Hilfe: ", options);

# Quellen

## Apache Log4J

http://logging.apache.org/log4j/2.0/manual/configuration.html

## Apache Commons CLI

http://commons.apache.org/sandbox/commons-cli2/manual/index.html

## Mockito

https://code.google.com/p/mockito/

## JUnit

http://junit.org/