2014

Roboterfabrik – A01

NEBENLÄUFIGE PROGRAMMIERUNG

STEFAN ERCEG, MARTIN KRITZL, SEBASTIAN STEINKELLNER

Inhalt

1. Aufgabenstellung 2
2. Requirementanalyse 4
3. Designüberlegung mittels UML-Klassendiagramm 5
3.1. Abbildung 5
3.2. Überlegungen zur Struktur 6
4. detaillierte Arbeitsaufteilung mit Aufwandabschätzung 7
4.1. Aufwandabschätzung 7
4.2. Arbeitsaufteilung für die Code-Erstellung 7
5. anschließende Endzeitaufteilung 8
5.1. Erceg 8
5.2. Kritzl 8
5.3. Steinkellner 9
5.4. Gesamtsumme 9
6. Arbeitsdurchführung 10
6.1. Probleme 10
6.1.1 Genauere Erläuterungen 10
7. Testbericht 11
8. Quellen 11

1. Aufgabenstellung

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

2. Requirementanalyse

Lager:

- darin werden die einzelnen Bestandteile des Roboters (= Threadee) gesammelt
- wird als Verzeichnis abgebildet

Lagermitarbeiter:

- verwaltet Ein- und Ausgang des Lagers

Spielzeugroboter:

- besteht aus zwei Augen, einem Rumpf, einem Kettenantrieb und zwei Armen
- die jeweiligen Bestandteile des Roboters werden als Files abgebildet

Lieferant:

- gibt dem Lagermitarbeiter Bestandteile die er liefert
- Art der gelieferten Teile soll nach einer bestimmten Zeit gewechselt werden

Montagemitarbeiter:

- fordert alle Bestandteile des Roboters
- falls nicht alle Teile geliefert werden, schickt der jeweilige Montagemitarbeiter sie wieder zurück, um andere Mitarbeiter nicht zu blockieren
- baut den Threadee zusammen
- stellt den fertigen Threadee wieder in das Lager zurück

Sekretariat:

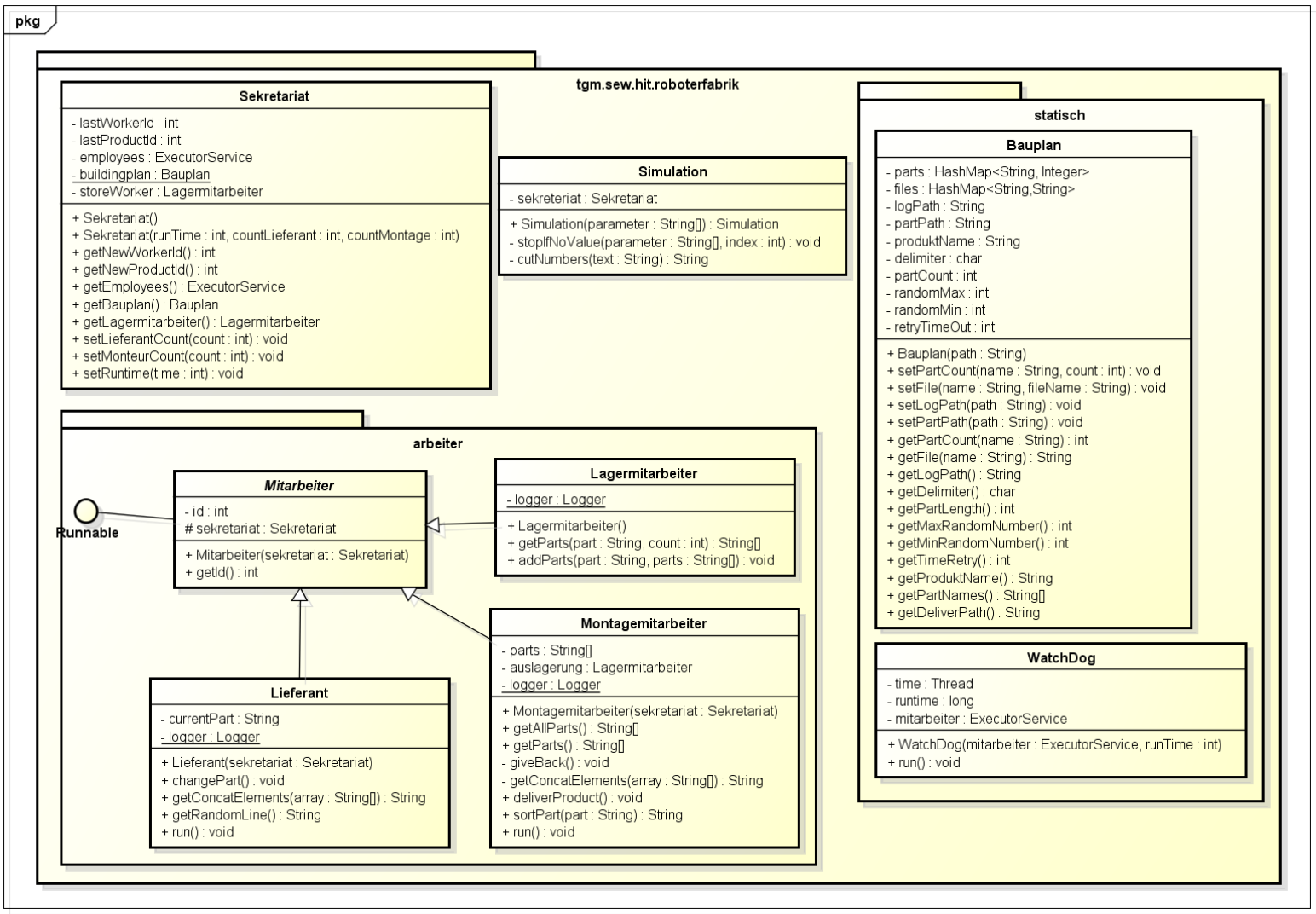
- verwaltet die IDs der Mitarbeiter, damit nur eindeutige vorkommen

Loggen:

- alle Aktivitäten der Mitarbeiter werden protokolliert
- zu jeder Aktivität wird auch gespeichert, wer diese ausgeführt hat, dies wird durch die IDs realisiert

3. Designüberlegung mittels UML-Klassendiagramm

3.1. Abbildung



3.2. Überlegungen zur Struktur

Das UML-Diagramm wurde mittels des Programms „Astah Professional“ erstellt.

Unser Projekt besteht aus **3 Packages**:

1.) das Hauptpackage „tgm.sew.hit.roboterfabrik“

In diesem Package befinden sich die Klassen „Sekretariat“ und „Simulation“.

- Das Sekretariat ist für die Vergabe von IDs zuständig. Außerdem vergibt es auch eindeutige Kennungen für die erstellten Threadees.
- Die Simulation ist unsere „Main“-Klasse für das Programm. Mittels dieser Klasse startet das Programm und man kann verschiedene Argumente, z.B. das Logverzeichnis, angeben.

2.) das Subpackage „tgm.sew.hit.roboterfabrik.arbeiter“

In diesem Package befindet sich die Klasse Mitarbeiter mit ihren Subklassen Lieferant, Lagermitarbeiter und Montagemitarbeiter. Alle Personen, welche aktiv am Erstellen des Threadees etwas Bestimmtes beitragen, wurden in dieses Package gegeben. Zusätzlich implementiert die Klasse „Mitarbeiter“ das Interface „Runnable“.

- In der Klasse „Mitarbeiter“ holen wir uns die neue ID, welche vom Sekretariat mittels der Methode „getNewWorkerId()“ vergeben wurde.
- Der Lagermitarbeiter ist dafür zuständig, dass die Montagemitarbeiter bei Anfragen die gewünschte Anzahl an Bestandteilen bekommen. Dies wird mit der Methode „getParts(part,count)“ gelöst. Kommen durch die Lieferanten neue Bestandteile ins Lager, werden diese mit Hilfe der Methode „addParts(part,parts[])“ in die bestimmte Datei geschrieben.
- Der Lieferant fügt dem Lager ständig zufällige Parts hinzu. Die Methode „changePart()“ sorgt dafür, dass der Part, der geliefert werden soll, zufällig geändert wird.
- Der Montagemitarbeiter ist für die Sortierung der Parts und somit für die Produktion des Threadees verantwortlich. Die Methode „getAllParts()“ überprüft, ob alle Teile angefordert werden konnten.

3.) das Subpackage „tgm.sew.hit.roboterfabrik.statisch“

In diesem Package befinden sich die Klassen „Bauplan“ und „WatchDog“. Wie der Name des Packages bereits sagt, werden hier die Informationen, welche die ganze Zeit über statisch sind, d.h. nicht verändert werden, implementiert.

- In der Klasse „Bauplan“ werden Methoden wie z.B. „setPartCount(name,count)“, welche die Anzahl des genannten Teiles festlegt, die für einen Threadee gebraucht werden, implementiert. Generell sind bei Bauplan getter- und setter-Methoden zu finden.
- Die Klasse „WatchDog“ sorgt dafür, dass Threads nach einer bestimmten Zeit beendet werden sollen. Unter den Threads versteht man in diesem Programm die bestimmten Mitarbeiter.

4. detaillierte Arbeitsaufteilung mit Aufwandabschätzung

4.1. Aufwandabschätzung

Teilaufgabe	Aufwandabschätzung (in Minuten)
UML-Diagramm	250
Code-Erstellung inkl. Javadoc-Kommentaren	500
Debuggen	200
Testen	250
Protokoll	200
Gesamt	***1400 (ca. 23 ½ Stunden)***

4.2. Arbeitsaufteilung für die Code-Erstellung

Klasse	Steinkellner	Kritzl	Erceg
Simulation	x		
Sekretariat	x		
Mitarbeiter			x
Lagermitarbeiter			x
Montagemitarbeiter		x	
Lieferant		x	
Bauplan	x		
Watchdog		x	

5. anschließende Endzeitaufteilung

5.1. Erceg

Arbeit	Datum	Zeit in Minuten
Requirementanalyse	11.09.2014	20
UML	11.09.2014	60
UML	16.09.2014	20
UML	17.09.2014	15
UML	18.09.2014	20
Egit	23.09.2014	30
Log4J	25.09.2014	90
Mitarbeiter,Lagermitarbeiter	27.09.2014	110
RAF	28.09.2014	20
Testen	29.09.2014	30
Protokoll	29.09.2014	50
Änderungen am UML	29.09.2014	20
UML-Änderungen	29.09.2014	60
Debuggen	30.09.2014	180
Protokoll	30.09.2014	120
Testen	01.10.2014	120
Gesamt	***01.10.2014***	***965 (ca. 16 Stunden)***

5.2. Kritzl

Arbeit	Datum	Zeit in Minuten
Requirementanalyse	11.09.2014	3
UML	11.09.2014	60
UML	16.09.2014	20
UML	17.09.2014	70
UML	18.09.2014	20
UML	21.09.2014	45
Egit	21.09.2014	70
ExecutorService	23.09.2014	20
UML	23.09.2014	15
Arbeitseinteilung	24.09.2014	30
WatchDog, Lieferant, Montagemitarbeiter	24.09.2014	150
Lieferant, Montagemitarbeiter	25.09.2014	120
Kommentieren, Änderungsvorschläge	25.09.2014	45
log-statements, sortPart	26.09.2014	60
edit log-statements, improve Code	28.09.2014	90
change getAllParts(), tests	28.09.2014	120
change giveBack(ArrayList<String[], int)	29.09.2014	40

Debuggen	29.09.2014	60
Debuggen	30.09.2014	120
Testen, getAllParts	30.09.2014	120
Gesamt	***01.10.2014***	***1278 (ca. 21 ½ Stunden)***

5.3. Steinkellner

Arbeit	**Datum**	**Zeit in Minuten**
UML	11.09.2014	60
UML	17.09.2014	70
UML	23.09.2014	15
Arbeitseinteilung	24.09.2014	30
Bauplan,Sekretariat,Programmparameterprüfung	25.09.2014	80
Dateizugriff	27.09.2014	60
Krisensitzung (telefonisch)	28.09.2014	16
Anpassungen	28.09.2014	40
Anpassungen	29.09.2014	56
Debugging	01.10.2014	150
Gesamt	***01.10.2014***	***577 (ca. 9 ½ Stunden)***

5.4. Gesamtsumme

2820 Minuten = 47 Stunden

Im Vergleich zur Aufwandabschätzung (23 ½ Stunden) haben wir gesamt um das Doppelte länger an Zeit gebraucht.

6. Arbeitsdurchführung

6.1. Probleme

Probleme, welche während des Erstellens des Quellcodes für das Programm entstanden sind, waren folgende:

- Planung des Threadpools
- Planung des Bauplans (statisch)
- Zugriff auf den Lagermitarbeiter (synchronized)
- Effektives Schreiben/Löschen aus dem Random Access File
- Testen der Methoden, die andere Objekte benötigen (getParts())
- Loggen
- Fehlerhafte Pfadangaben (vergessene ‚/' bei zusammengesetzten Pfaden)

6.1.1 Genauere Erläuterungen

Effektives Schreiben/Löschen aus dem Random Access File

Falls der Montagemitarbeiter bestimmte Bestandteile anfordert, sollen diese vom Lagermitarbeiter aus der Datei gelesen werden und die Zeilen aus der Datei je nach Anzahl des Bestandteils gelöscht werden. Fügt der Lieferant neue Bestandteile ins Lager hinzu, sollen die Zeilen je nach Anzahl des Bestandteils hinzugefügt werden.
Unser Problem war, dass bei jedem Hinzufügen eines bestimmten Parts in der Datei kein Zeilenumbruch gemacht wurde und dem Montagemitarbeiter beim Übergeben von Parts nicht die Zeilen aus dem File, sondern null zurückgeliefert wurde.

Loggen

Beim Ausführen des Programms tauchten bezüglich der Log4J-Implementierung folgende Warnings auf:

```
log4j:WARN No appenders could be found for logger (dao.hsqlmanager).
log4j:WARN Please initialize the log4j system properly.
```

Dies lag daran, dass die Log4J-Konfigurationsdatei vorher nicht hinzugefügt wurde. Bei Log4J 1 wird im Gegensatz zu Log4J 2 nicht XML, sondern ein Propertiesfile verwendet.

7. Testbericht

Klasse Bauplan:

variablen mit Getter und Setter Methoden: testen ohne Probleme erledigt (Setter aufgerufen, Getter aufgerufen, Input & Output verglichen)

variablen mit Getter ohne Setter: Default Werte abgefragt ohne Probleme

Klasse Sekretariat:

Produkt-ID: 3 mal neue Produkt-ID angefordert, sind fortlaufend => ✓

Arbeiter-ID: anfangs 3 mal, dann 6 mal aufgerufen, wobei die ID ein sehr seltsames Verhalten zeigt.
eigentlich identischer Code, wie bei Produkt-ID, nur mit unterschiedlicher Variable.
in beiden Fällen wird die Variable nur einmal auf 0 gesetzt, dann von der synchronized Methode mit ++ um 1 erhöht und zurückgegeben, ansonsten nirgendwo aufgerufen.
trotzdem beginnt diese Variable bei 0, springt beim ersten ++ auf 2, beim nächsten Mal zurück auf 1 und dann auf andere Werte.
z.B: bei 25 Monteuren und 15 Lieferanten hatten eine Arbeiter-ID von 54, obwohl maximal 40 (25+15) möglich wäre

Klasse Monatgemitarbeiter:

sortPart: die zufällige Reihenfolge der Zahlen des Parts werden in die richtige Reihenfolge gebracht => ✓

getParts: Probleme mit dem Festlegen der Returns von Mockito Objekten. Es werden nicht die erwünschten Werte wiedergegeben.

Klasse Lieferant:

getRandomLine: Probleme mit dem Festlegen des Regex zum Vergleichen der zufällig generierten Zahlen.

Klasse Mitarbeiter:

getId: Lieferte als Anfangs-ID 2 zurück.

Klasse Lagermitarbeiter:

getParts: Schrieb die gewünschten Zeilen nicht ins File. Lieferte null zurück, wenn vorher Teile hinzugefügt und dann ausgelesen wurden.

8. Quellen

Unser Git-Repository: https://github.com/ssteinkellner/Roboterfabrik

Java API 8: http://docs.oracle.com/javase/8/docs/api/

Log4J: http://www.torsten-horn.de/techdocs/java-log4j.htm