C:\Users\FOCK\Desktop\tgm_logo.gif

**SEW**

**Nebenläufige Roboterfabrik**

**Fock, Hackenberger, Tiryaki**

Inhalt

[1. Aufgabenstellung / Requirementsanalyse 2](#_Toc399941734)

[2. Zeitabschätzung / Zeitaufzeichnung 3](#_Toc399941735)

[3. Diagramme (UML / EER... ) 4](#_Toc399941736)

[4. Things we've done 6](#_Toc399941737)

[5. Lessons learned 6](#_Toc399941738)

[6. Quellen 7](#_Toc399941739)

# 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.

**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

# Zeitabschätzung / Zeitaufzeichnung

Planung

|  |  |  |
| --- | --- | --- |
| **Aufgabe** | **Abgeschätzt** | **Aktuell** |
| UML-Diagramm | 1.5h | 2h |
| UML-Diagramm  Verbessert |  | 1h |

Organisation

|  |  |  |
| --- | --- | --- |
| **Aufgabe** | **Abgeschätzt** | **Aktuell** |
| Git | 0.2h | 0.2h |

Implementierung

|  |  |  |
| --- | --- | --- |
| **Aufgabe** | **Abgeschätzt** | **Aktuell** |
| Klassen & Methoden | 8h | 9h |
| CLI | 0.6h | 0.6h |
| Log4J | 1 h | 1h |

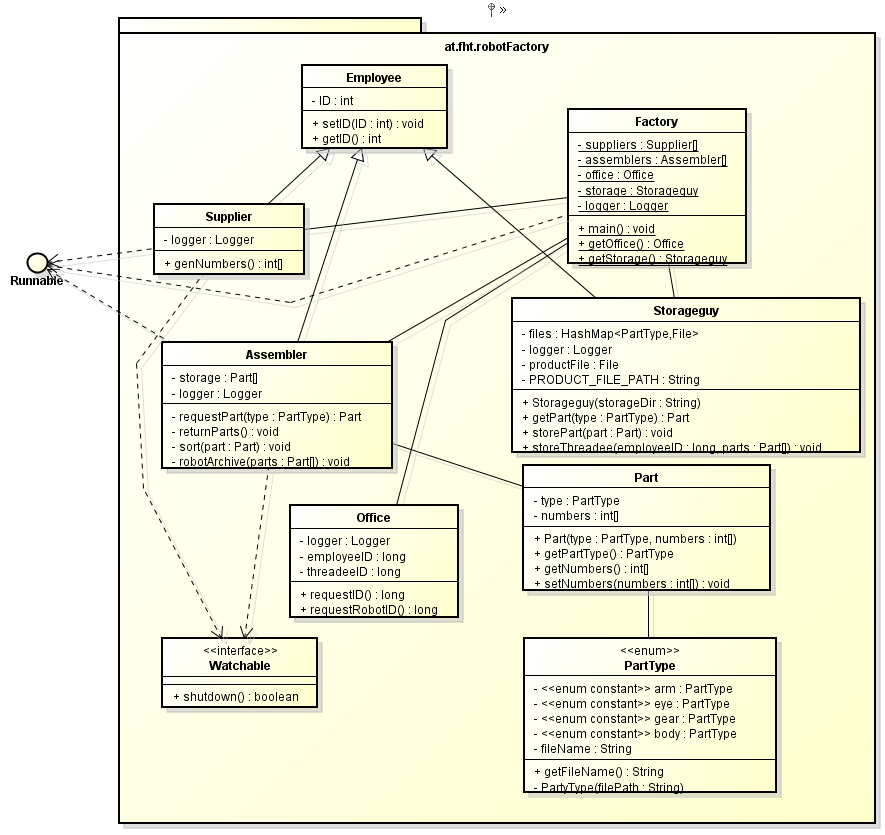
Dokumentation

|  |  |  |
| --- | --- | --- |
| **Aufgabe** | **Abgeschätzt** | **Aktuell** |
| Dokumentation des Projektes | 2h | 3h |

Gesamt

|  |  |  |
| --- | --- | --- |
| **Abgeschätzt : Aktuell** | **13.3h** | **16.8** |

# Klassendiagramm



# Things we've done

Log4j

Ermöglicht, zu loggen.

Commons CLI

Ermöglicht, Konsolenargumente zu definieren.

JUnit

Ermöglicht, einfach Java-Code zu testen.

Klassen + Methoden

**Office**

* Registriert die IDs der Mitarbeiter
* Registriert die IDs der Threadees

**Employee**

* Vererbt seinen Kindern (Assembler, Storageguy, Supplier)
  + ID bekommen
  + ID verändern

**Assembler (extends Employee implements Runnable, Watchable)**

* Fordert einen Part mit einem expliziten PartType + generierten Nummern, vom Storageguy, an.
* Gibt alle Parts zurück die er angefragt hatte, falls eines bei seiner „Bestellung“ fehlt.
* Sortiert die Nummern, die bei einem Part dabei sind, als ein symbolisches Zeichen für das Zusammensetzen eines Threadees.
* Lagert den zusammengesetzten Threadee via Storageguy in ein .csv File ein.
* Laufzeit
  + Fordert alle Parts für einen Threadee an
  + Sortiert jeden einzelnen Part
  + Schickt den „assemblierten“ Threadee an den Storageguy damit dieser ihn einlagert

**Storageguy (extends Employee)**

* Holt sich einen Part, mit einem expliziten PartType aus einem dem entsprechenden .csv flie.
* Fügt einen Part, mit einem expliziten PartType, einem dem entsprechenden .csv File hinzu.
* Fügt einen Threadee einem .csv hinzu, welches nur für die Lagerung der Threadees dient.

**Supplier (extends Employee implements Runnable, Watchable)**

* Generiert Nummern für einen Part
* Laufzeit
  + Liefert dem Storageguy, nach einem Zufallsprinzip, den PartTyp & selbst generierte Nummern.

**Factory**

* Mit Commons CLI werden Konsolargumente validiert
* Startet den Log4j Job
* Weckt (produziert) den Storageguy & das office auf.
* Weckt (produziert) je nach Konsolargumenten seine Arbeiter auf
* Laufzeit
  + Lässt die Factory „schlafen“ solange die erweckten Arbeiter arbeiten & sobald die Factory wieder aufwacht holt sie den Watchdog welcher all die herumeifernden Arbeiter in deren Betten jagt.

**Part**

* Ein Part besteht aus einem PartTyp (ARM, EYE,…) und aus 20 zufällig generierten Zahlen.
  + - Getter- Setter- Methoden

**PartType**

* Enum

# Lessons learned

Threads

Threads können parallel arbeiten 🡺 gleichzeitig einen Programmcode ausführen.

Bezeichnungen für Threads

* Aktivitätsträger
* Leichtgewichtiger Prozess

Es gibt 2 Arten der Implementierung eines Threads

* Implementation der Klasse Thread 🡺 (implements Runnable)
  + Diese Methode ist “besser” da man nicht die Verebungshirarchie blockt
* Erweiterung der Klasse Thread (extends Thread)

**INTERRUPT() 🡺 DONT DO IT;   
WATCHDOG 🡺 YES SIR;**

Ein Thread muss die Klasse run() überschreiben und wird gestartet durch .start() welcher die run() startet;

Man kann Threads synchronisieren indem man mit dem Schlüsselwort synchronized kritische Abschnitte definiert. Diese kritischen Abschnitte werden blockiert sobald ein Thread in ihm agiert und lässt andere währenddessen warten.

* Die Threads können sich nicht in ihre Arbeit reinpfuschen

COMMONS CLI

Apache Commons CLI ermöglicht es Konsolenargumente zu validieren.

Kleines Beispiel.:

Alle Argumente werden zu Options hinzugefügt  
 Danach müssen die Argumente zergliedert werden.

Log4j

Apaches Log4j erlaubt es einfach zu loggen.

# Quellen

* Apache Commons-CLI
  + <http://commons.apache.org/sandbox/commons-cli2/manual/index.html>
  + <http://commons.apache.org/proper/commons-cli/usage.html>
* Apache
  + <http://logging.apache.org/log4j/2.0/manual/configuration.html>
* CSV
  + <http://javacsv.sourceforge.net/>
  + <http://opencsv.sourceforge.net/apidocs/index.html>
* JUnit
  + http://junit.org/