**Documentație**

Tehnici de Programare

Tema 2 : Queue Simulator

A realizat: Cozma Victoria,

Grupa 30225, Automatica si Calculatoare

1. Obiectivele lucrarii

Obiectivul lucrarii este simularea unei cozi (de exemplu, intr-un supermarket) cu ajutorul limbajului orientat pe obiect Java si a threadurilor. Pasii care au fost urmati pentru atingerea obiectivului au fost:

1. Analiza scenariului si identificarea use-case-ului
2. Identificarea cerintelor functionale si maparea acestora pe module
3. Realizarea unei diagrame de clase
4. Definirea claselor si obiectelor
5. Abstractizarea
6. Testarea lucrarii
7. Analiza problemei, modelare, scenarii, cazuri de utilizare

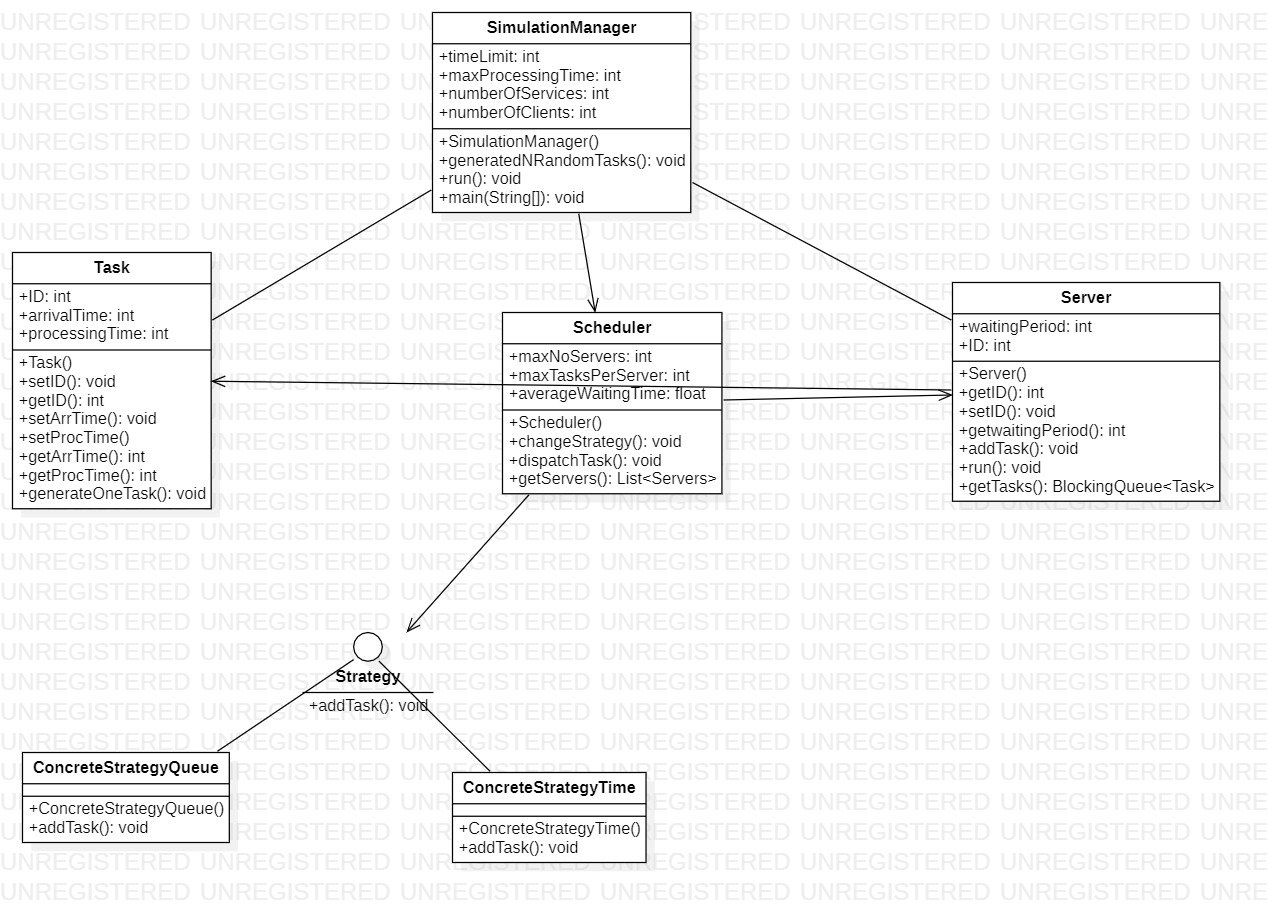
In procesul de realizare a temei am analizat cerintele problemei si am identificat urmatoarele use-case-uri si cerinte fundamentale:

* Implementarea serverilor (care tin locul casierului)
* Implementarea Task-urilor (care reprezinta clientii)
* Simularea asteptarii la coasa a clientilor si calcularea timpului mediu de asteptare

Pentru aceasta, am definit clasele generice Server si Task de maxim 200 linii cu metodele aferente de maxim 30 din linii. De asemenea, am definit clasele SimultionManager, Scheduler si Strategy, care implementeaza Runnable pentru a face posibila simularea a proiectului in timp real cu ajutorul Threadurilor.

1. Proiectare

Prima data, am construit diagrama de clase si relatiile dintre clase. Acesta este un prim pas, conform caruia ne vom ghida in continuare. Diagrama de clase contine 4 clase principale: Server, Task, Scheduler si SimulationManager (care tine locul clasei MainClass). De asemenea, diagrama contine interfata Strategy, care trebuie implementate conform specificatiilor temei. Din aceste clase au fost extrase use-case-urile determinate anterior.



1. Implementare

In continuare vom descrie fiecare clasa, cu atributele aferente si metodele de baza.

1. *Clasa Server* – tine locul casierului (intr-o asociatie cu lumea reala ) si are atributele waitingPeriod, care reprezinta suma timpilor de asteptare a tuturor clientilor (Task-urilor) care se afla in coada si in dependenta de care se va face selectia serverului in cazul utilizarii strategiei ConcreteStrategyTime, ID-ul serverului este folosit cu scop de afisare a numarului serverului (cozii). De asemenea, serverul mai contine si o lista de task-uri ce urmeaza a fi indelpinite, stocate intr-o structura de date numita BlockingQueue <Tasks>. Am preferat aceasta structura de data simplului ArrayList pentru a evita problemele cu firele de lucru. Toate atributele au gradul de vizibilitate private. Am folosit fenomenul de incapsulare pentru a limita accesul obiectelor de alta clasa la atributele clasei Server, Astefel, am implementat settere si gettere pentru fiecare atribut. De asemenea, Serverul poate fi initializat din constructorul Server(int c, int g). Metoda run este cea care asigura posibilitatea simularii in timp real a cozii. In cadrul acestei metode, se ia primul element din coada , si pune threadul serverului pe wait un interval de timp egal cu timpul de procesare al Task-ului. Clasa Server implementeaza interfata Runnable si implementeaza metoda run, pentru a face posibila implementarea Threaduri-lor.

**public** **class** Server **implements** Runnable {

**private** BlockingQueue<Task> tasks = **new** LinkedBlockingQueue();

**private** **int** waitingPeriod;

**private** **int** ID;

**public** Server() {

waitingPeriod = 0;

}

**int** getID() {

**return** ID;

}

**void** setID(**int** id) {

ID = id;

}

**int** getwaitingPeriod() {

**return** waitingPeriod;

}

**public** **void** addTask(Task newTask) {

tasks.add(newTask);

waitingPeriod += newTask.getProcTime();

}

@Override

**public** **void** run() {

**while** (**true**) {

Task first = **new** Task();

**try** {

first = tasks.element();

} **catch** (Exception e) {

**continue**;

}

// System.out.println("s-a extras task-ul" + first.getID());

**try** {

**int** aux = first.getProcTime();

**for** (**int** i = 0; i < aux; i++) {

Thread.*sleep*(1000);

tasks.element().setProcTime(tasks.element().getProcTime()-

}

tasks.take();

} **catch** (InterruptedException e) {

System.***out***.println("exceptie");

}

// folosim pentru strategy

waitingPeriod -= first.getProcTime();

}

}

**public** BlockingQueue<Task> getTasks() {

**return** tasks;

}

}

1. *Clasa Task* – are drept atribute ID, arrivalTime si processingTime. Aceasta clasa poate fi asociata cu clientii, daca e sa facem o analogie cu lume reala. Id-ul clientului este folosit pentru a identifica fiecare client, arrivalTime-ul reprezinta timpul cand clientul ajunge a fi primul element din coada, adica poate fi servit de Server si procTime-ul indica timpul necesar Task-ului pentru a fi indeplinit. De fiecare data cand Task-ul ajunge sa fie servit, thread-ul corespunzator serverului isi opreste executia pentru “processingTime” secunde, timp in care acest atribut al Task-ului devine 0 si Task-ul se considera indeplinit.Au fost implementate metodele principale: gettere, settere si generateOneTask, care genereaza Task-uri random in functie de datele primite din fisier. Timpul arrive si timpul de procesare sunt cuprinse in intervalul stabilit si se genereaza random.

**public** **class** Task {

**private** **int** ID;

**private** **int** arrivalTime;

**private** **int** processingTime;

**public** **void** setID(**int** id) {

ID = id;

}

**public** **int** getID() {

**return** ID;

}

**void** setArrTime(**int** a) {

arrivalTime = a;

}

**void** setProcTime(**int** b) {

processingTime = b;

}

**int** getArrTime() {

**return** arrivalTime;

}

**int** getProcTime() {

**return** processingTime;

}

**public** **void** generateOneTask(**int** id, **int** tMinArr, **int** tMaxArr, **int** tMinSer, **int** tMaxSer) **throws** Exception {

Random rand = **new** Random();

**int** aux1 = rand.nextInt((tMaxArr - tMinArr) + 1) + tMinArr;

**this**.setArrTime(aux1);

**int** aux2 = rand.nextInt((tMaxSer - tMinSer) + 1) + tMinSer;

**this**.setProcTime(aux2);

**this**.setID(id);

}

1. *Clasa SimulationManager* – este responsabila pentru gestionarea tuturor proceselor din program()supermarket. Am incercat sa implementez aceasta clasa cat mai user-firendly, adica sa afisez in consola cat mai clar coada, clientii si sa implementez o simulare real-time. De asemenea, am dat nume cat mai sugestive variabilelor , pentru ca user-ul sa stie instinctiv ce reprezinta fiecare.

Aceasta clasa declara si initializeaza toate elementele principale a programului: Sheduler-ul, SelecyionPolicy-ul si task-urile generate. Aceasta casa implementeaza interfta Runnable si suprascrie metoa run(), in care am initializat thread-ul de baza, care tine evidenta timpului. Aici au loc si afisarile.

**public** **class** SimulationManager **implements** Runnable {

**public** **int** timeLimit = 100; // max processing time

**public** **int** maxProcessingTime = 10;

**public** **int** minProcessingTime = 2;

**public** **int** numberOfServices = 3;

**public** **int** numberofClients = 100;

**public** SelectionPolicy selectionPolicy = SelectionPolicy.***SHORTEST\_QUEUE***;

**private** Scheduler scheduler;

**private** List<Task> generatedTasks = **new** ArrayList();

**public** SimulationManager(**int** N, **int** Q, **int** simTime, **int** ta1, **int** ta2, **int** ts1, **int** ts2) **throws** Exception {

timeLimit = simTime;

// maxProcessingTime = ts2;

// minProcessingTime = ts1;

numberOfServices = Q;

numberofClients = N;

scheduler = **new** Scheduler(numberOfServices, numberofClients, selectionPolicy);

generateNRandomTasks(numberofClients, ta1, ta2, ts1, ts2);

}

@Override

**public** **void** run() {

// creare nou fisier iesire;

FileWriter myWriter = **null**;

**int** currentTime = 0;

**int** closedCounter = 0;

**while** (currentTime <= timeLimit) {

System.***out***.println("Time:" + currentTime);

Iterator itr = generatedTasks.iterator();

**while** (itr.hasNext()) {

Task task = (Task) itr.next();

**if** (task.getArrTime() == currentTime) {

scheduler.dispatchTask(task);

itr.remove();

}

}

System.***out***.print("Wainting clients: ");

**for** (Task task : generatedTasks) {

System.***out***.print("(" + task.getID() + "," + task.getArrTime() + "," + task.getProcTime() + ")");

}

System.***out***.println();

**for** (Server s : scheduler.getServers()) {

System.***out***.print("Coada " + s.getID() + ": ");

**if** (s.getTasks().isEmpty()) {

System.***out***.println("closed");

} **else** {

**for** (Task m : s.getTasks()) {

System.***out***.print("(" + m.getID() + "," + m.getArrTime() + "," + m.getProcTime() + ") ");

}

System.***out***.println();

}

}

System.***out***.println();

currentTime++;

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**for** (Server s : scheduler.getServers()) {

**if** (s.getTasks().isEmpty()) {

closedCounter++;

}

}

**if** ((scheduler.getServers().size() == closedCounter) && (generatedTasks.isEmpty())) {

System.***out***.println("Time:" + currentTime);

**try** {

myWriter = **new** FileWriter("in-test-3.txt", **true**);

myWriter.append("Time\n");

myWriter.close();

} **catch** (IOException e) {

System.***out***.println("An error occurred.");

e.printStackTrace();

}

System.***out***.println("Waiting clients:");

**for** (Server s : scheduler.getServers()) {

System.***out***.println("Coada " + s.getID() + ": closed");

}

**break**;

}

closedCounter = 0;

}

// afisam average waiting time-ul

scheduler.averageWaitingTime = scheduler.averageWaitingTime / numberofClients;

System.***out***.println("Average waiting time: " + scheduler.averageWaitingTime);

System.*exit*(0);

}

1. *Clasa Scheduler*

Aceasta clasa este responsabila pentru impartirea Task-urilor spre servere si este un simulator a “organizatorului” real al unui procesor care randuieste task-urile catre executie.. Tot in interiorul clasei sunt implementate metodele changeStrategy, care selecteaza metoda de selectare a serverului (in dependenta de lungimea cozii sau de timpul de asteptare), dispatchTask - care imparte task-urile catre servere, si getter pentru lista de servere. Toate atributele sunt, la fel, Private, pentru a respecta principiul de incapsulare.

**public** **class** Scheduler {

**private** List<Server> servers = **new** ArrayList();

**private** **int** maxNoServers;

**private** **int** maxTasksPerServer;

**public** **float** averageWaitingTime = 0;

**private** Strategy strategy;

**public** Scheduler(**int** maxNoServers, **int** maxTasksServer, SelectionPolicy policy) {

**this**.maxNoServers = maxNoServers;

**this**.maxTasksPerServer = maxTasksServer;

**for** (**int** i = 0; i < maxNoServers; i++) {

Server aux = **new** Server();

aux.setID(i);

Thread t = **new** Thread(aux);

servers.add(aux);

t.start();

}

**this**.changeStrategy(policy);

}

**public** **void** changeStrategy(SelectionPolicy policy) {

**if** (policy == SelectionPolicy.***SHORTEST\_QUEUE***) {

strategy = **new** ConcreteStrategyQueue();

}

**if** (policy == SelectionPolicy.***SHORTEST\_TIME***) {

strategy = **new** ConcreteStrategyTime();

}

}

**public** **void** dispatchTask(Task t) {

strategy.addTask(servers, t);

averageWaitingTime += t.getProcTime();

}

**public** List<Server> getServers() {

**return** servers;

}

}

1. *Interfata Strategy*

Aceasta interfata are rolul de a concepe o strategie conform careia se vor distrbui clientii in cozi. Aceasta interfata va fi implementate de clasele ConcreteStrategyQueue si ConcreteStrategyTime, car vor trebuie sa implementeze metoda addTask() declarata in interfata;

**public** **interface** Strategy {

**public** **void** addTask(List<Server> servers , Task t);

}

1. *Clasele ConcreteStrategyTime si ConcreteStrategyQueue*

Aceste clase implementeaza interfata Strategy si implementeaza metoda addTask. In functie de clasa, aceasta metoda este implementate diferit. In priml caz, Un client va fi adaugat la serverul cu cel mai scurt timp de asteptare. In al doilea caz, Task-ul va fi preluat de server-ul cu cele mai putine task-uri.

**public** **class** ConcreteStrategyTime **implements** Strategy {

@Override

**public** **void** addTask(List<Server> servers, Task t) {

**int** min = 1000;

**for** (Server s : servers) {

**if** (s.getwaitingPeriod() < min) {

min = s.getwaitingPeriod();

}

}

**for** (Server s : servers) {

**if** (s.getwaitingPeriod() == min) {

s.addTask(t);

;

}

}

}

}

**import** java.util.List;

**public** **class** ConcreteStrategyQueue **implements** Strategy {

@Override

**public** **void** addTask(List<Server> servers, Task t) {

**int** min = 10000;

**for** (Server s : servers) {

**if** (s.getTasks().size() < min) {

min = s.getTasks().size();

}

}

**for** (Server s : servers) {

**if** (s.getTasks().size() == min) {

s.addTask(t);

**break**;

}

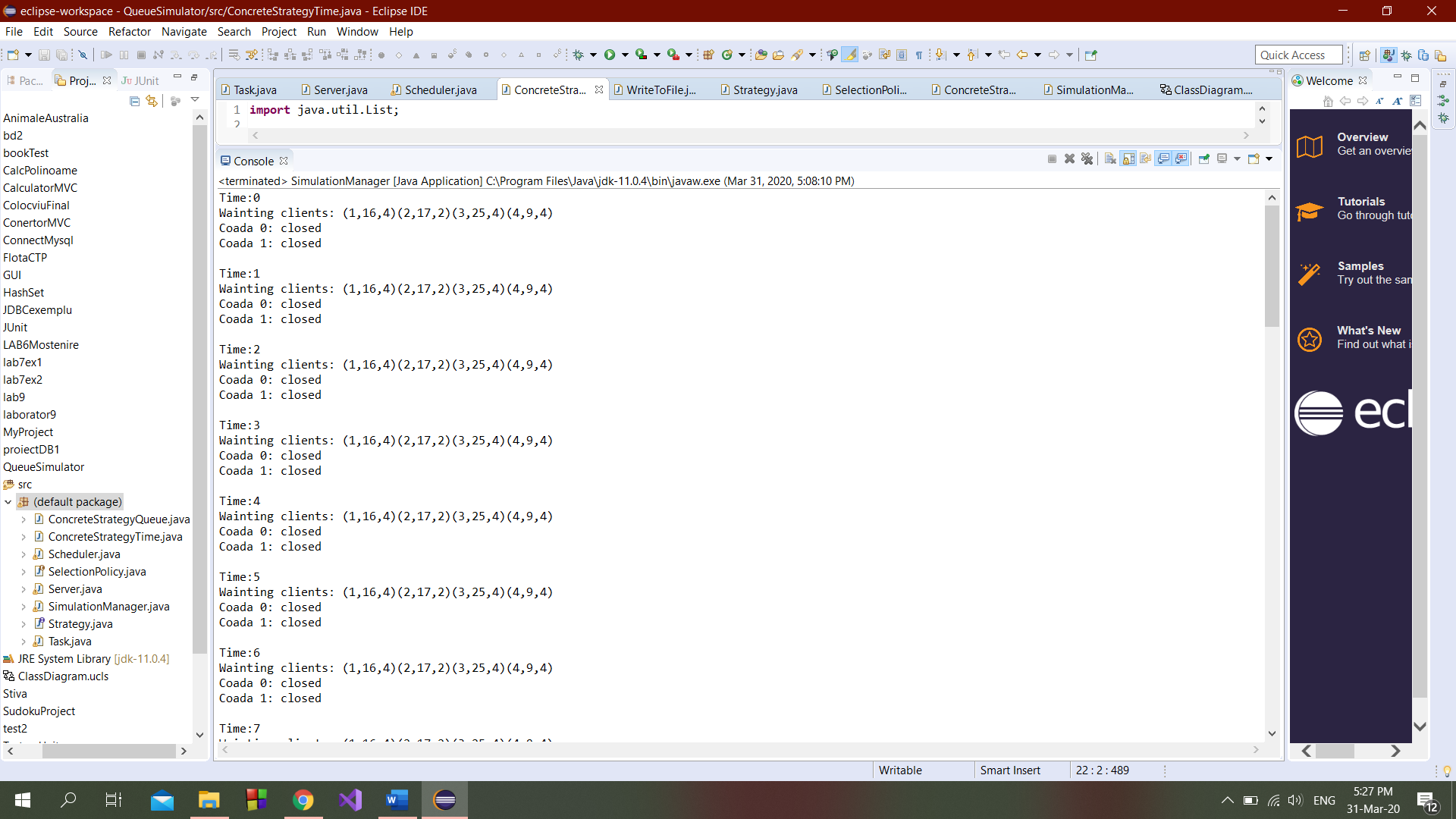
}

}

}

1. Rezultate

Ambele strategii au fost testate manual, prin introducerea a mai multe date de intrare. De asemenea, au fost testate si tratate si cazurile speciale.



1. Concluzii

Toate obiectivele proiectului au fost atinse. Aceasta tema mi-a consolidat cunostintele de programare orientata pe obiect, intrucat implementarea temei a necesitat structurarea claselor si crearea obiectelor : Scheduler, SimulationManager. De asemenea, am invatat sa implementez strategiile, ceea ce este o buna tehnica de programare si o voi implementa pe viitor. Am invatat sa folosesc firele de executie si sa simulez cu ajutorul lor o aplicatie in timp real. Ca posibilitati de dezvoltare ulterioara, ar putea fi implementata o inerfata care sa semene cat de mult cu coada unui supermarket. Ca functionalitate, s-ar putea adauga alte strategii de impartire a clientilor .

1. Bibliografie
2. <http://coned.utcluj.ro/~salomie/PT_Lic/5_Courses/>
3. <https://moodle.cs.utcluj.ro/pluginfile.php/44577/mod_resource/content/3/12-Testarea_Unitara.pdf>
4. <https://moodle.cs.utcluj.ro/pluginfile.php/44568/mod_resource/content/4/POO08.pdf>
5. <http://coned.utcluj.ro/~salomie/PT_Lic/4_Lab/Assignment_2/Java_Concurrency.pdf>