



UNIVERSITATEA DE AUTOMATICA SI CALCULATOARE

## DOCUMENTATIE TEMA 2

### Simulator de cozi

Nume si prenume: Timis Iulia Georgeana

Grupa: 30226

Profesor laborator: Dorin Vasile Moldovan

# CUPRINS

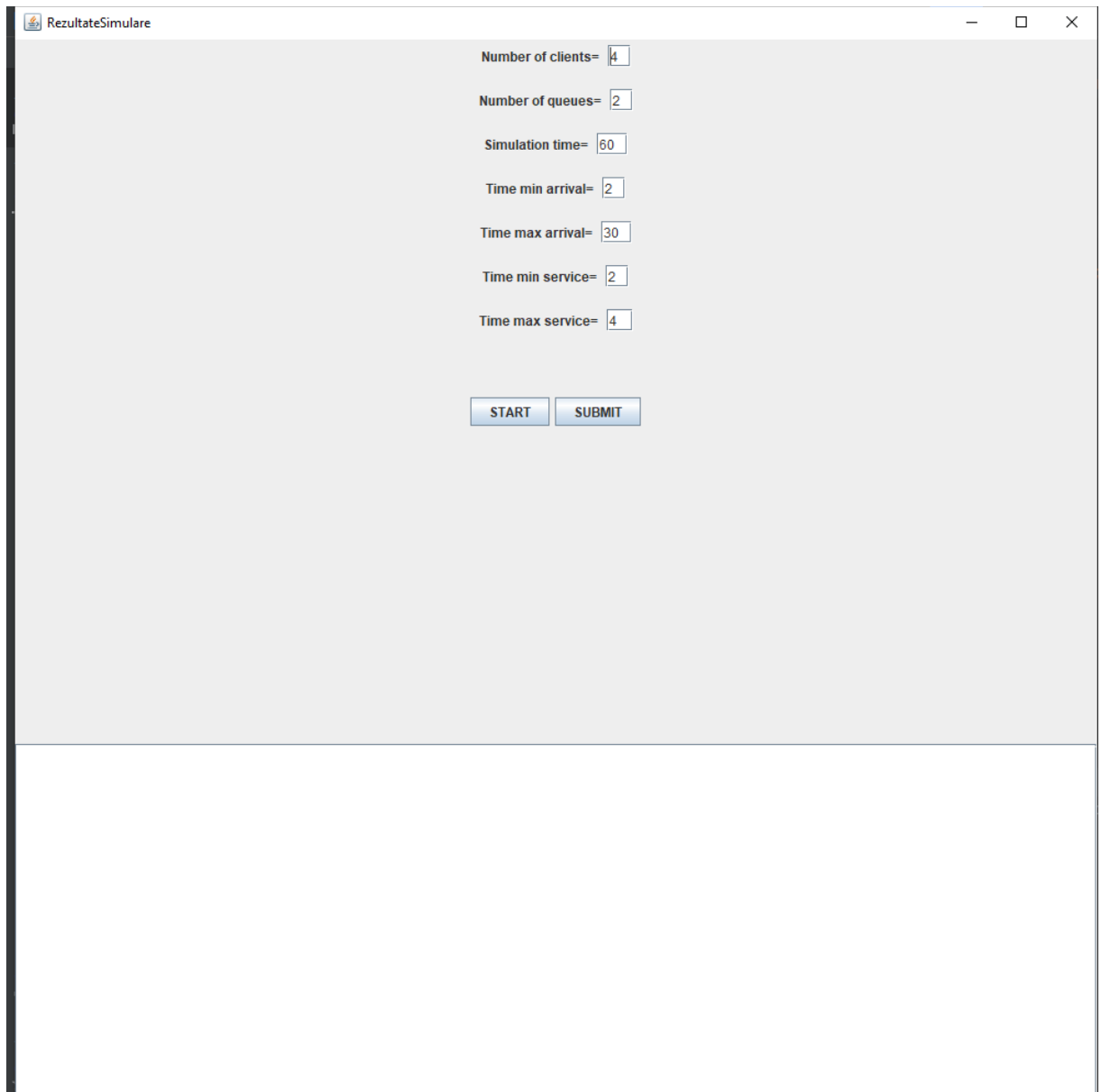
1. Probleme si solutia problemei
2. Obiective
3. Studiul problemei, modelare, scenarii, cazuri de utilizare
4. Proiectare(decizii de proiectare, diagrame UML, structuri de date, proiectare clase, interfata utilizator)
5. Implementare
6. Rezultat
7. Concluzii
8. Bibliografie

## 1. Probleme si solutia problemei

Acest program are drept problema construirea unor cozi pentru diferite activitati, de exemplu: intr-un magazin alimentar, unul cu imbracaminte, la aeroport/gara sau in diferite institutii, cu ajutorul unei interfete.

Pentru a ne usura munca, solutia acestei probleme ar fi folosirea unui simulator de cozi. Cum realizam acest lucru? Dupa deschiderea ferestrei cu interfata, vom introduce date de la tastatura, vom genera cozile, intrarea si iesirea din ele, si momentele de timp: average wait time, average service time si peak hour.

Un exemplu pentru interfata grafica ar fi:



The screenshot shows a graphical user interface for a simulation program. The window title is "RezultateSimulare". It contains several input fields for simulation parameters, each with a label and a text box. The parameters are: "Number of clients=" with value 4, "Number of queues=" with value 2, "Simulation time=" with value 60, "Time min arrival=" with value 2, "Time max arrival=" with value 30, "Time min service=" with value 2, and "Time max service=" with value 4. Below these fields are two buttons: "START" and "SUBMIT". The interface is simple and functional, with a light gray background and standard Windows-style window controls.

Parameter	Value
Number of clients	4
Number of queues	2
Simulation time	60
Time min arrival	2
Time max arrival	30
Time min service	2
Time max service	4

Buttons: START, SUBMIT

## 2. Obiective

Obiectivul temei a fost sa proiectam in Java un simulator de cozi. Acest simulator primea ca date de intrare numarul de cozi disponibile, numarul de clienti care urma sa fie serviti si timpul maxim (secunde intervalului de simulare). Pentru clienti avem specificat timpul maxim si minim la care acestia isi terminau treaba si intervalul de timp (prin minim si maxim) pe care trebuiau sa il astepte la casa pentru a ajunge la procesarea comenzii. Proiectul trebuia sa genereze pornind de la aceste date de intrare o colectie de clienti cu date aleatorii care sa respecte intervalele de timp. Cerinta presupunea de asemenea si inchiderea sau deschiderea automata a cozilor, astfel incat, la inceput toate cozile sa fie inchise, urmand sa fie deschise cand primul client este gata sa fie procesat de acea coada, iar apoi inchise la final, iar cand nu mai sunt clienti care asteapta. Trebuia prevazut un "simulator de cozi" care se eficientizeze sistemul, astfel incat fiecare client sa fie directionat spre coada cu cel mai mic timp de asteptare in acel moment. Simularea trebuie sa aiba loc intr-un fisier de iesire, unde se vor afisa pentru fiecare perioada de timp (de cate o secunda) clientii care fac cumparaturile si clientii care si-au terminat cumparaturile, distribuiti la cozile aferente. La sfarsitul fisierului de iesire, trebuia afisat peakhour-ul, averageServiceTime si averageWaitTime. De asemenea, datele de intrare se citeau dintr-un fisier de intrare. Trebuia ca aplicatia sa fie implementata pe thread-uri (adica fire de executie), astfel incat fiecare coada avea un thread asociat si functiona independent de celelalte. Datele necesare vor fi introduse de utilizator random, iar rezultatul va fi generat, dupa apasarea butonului de „Start”. In cazul interfetei am folosit javafx pentru am considerat cel mai adecvat aspect si usor de utilizat pentru oricine ar vrea sa rezolve cat mai simplu acest aspect.

## 3. Studiul problemei, modelare, scenarii, cazuri de utilizare

Programul simulator de cozi este folosit pentru calcularea timpului si aranjarea persoanelor in cozi si iesirea acestora din coada la momentul finalizarii, pana cand toate cozile devin goale.

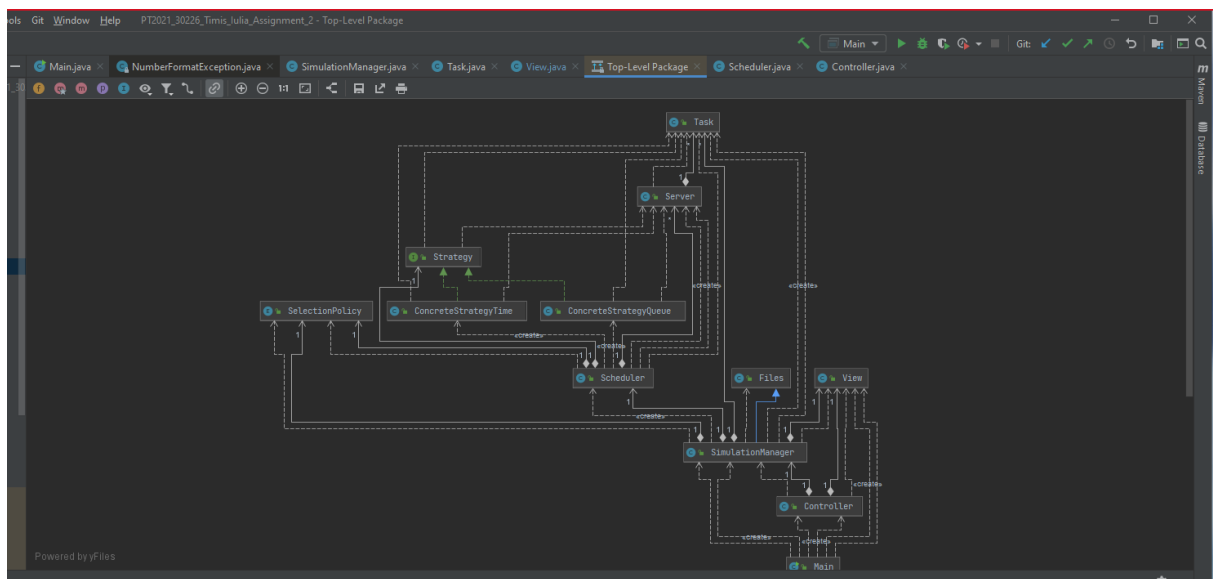
O coada este o structura de date logice (implementarea este făcută utilizând alte structuri de date) și omogene (toate elementele sunt de același tip). Aceasta structura are două operații de bază: adăugarea și extragerea unui element. Diferența fundamentală între cele două structuri este disciplina de acces. Coada functioneaza pe principiul FIFO, imaginati-va o coada la intrarea intr-un club. Primul client sosit este cel cu numarul unu, urmat de 2, 3, s.a.m.d. In ce ordine vor intra persoanele in club? In ordinea in care au venit. In momentul cand sosesc clienti noi, acestia se vor aseza in spatele numarului 5.

Problema a fost facuta in dupa exemplul alaturat prezentarii temei, astfel incat cozile sunt modelate ca servere, care primesc clienti pe care trebuie sa le proceseze. Serverele sunt monitorizate si primesc datele de intrare, care consulta timpul de asteptare la fiecare server in parte si ia o decizie cu privire la coada careia sa ii asigneze urmatorul client pentru a facilita eficienta algoritmului. De asemenea este responsabil si pentru inchiderea si deschiderea serverelor atunci cand este nevoie, datorita faptului ca se citeste din fisier. Pentru a putea simula acat mai usor am ales sa implementez si o interfata grafica care sa il ajute pe utilizator sa poata urmarii mai usor evolutia clientilor care asteapta la rand. Interfata are doua butoane, ea implicit este setata pe primul test prezentat temei, daca dorim sa modificam ceva va trebui mai intai sa dam submit si mai apoi poate incepe simularea cu datele dorite.

## 4. Proiectare (decizii de proiectare, diagrame UML, structuri de date, proiectare clase, interfata utilizator)

Unified Modeling Language (prescurtat UML) este un limbaj standard pentru descrierea de [modele](#) și [specificatii](#) pentru [software](#). Limbajul a fost creat de către consorțiul [Object Management Group](#) (OMG) care a mai produs printre altele și standardul de schimb de mesaje între sisteme [CORBA](#). UML a fost la bază dezvoltat pentru reprezentarea complexității programelor orientate pe obiect, al căror fundament este structurarea programelor pe clase, și instanțele acestora (numite și obiecte). Cu toate acestea, datorită eficienței și clarității în reprezentarea unor elemente abstracte, UML este utilizat dincolo de domeniul IT. Așa se face că există aplicații ale UML-ului pentru management de proiecte, pentru business Process Design etc.

Diagrama UML pentru proiectul meu se afla in exemplul de mai jos:



Ulterior pentru o solutie mai buna, am creat 3 pachete cu nume sugestive: model, view, si controller unde am incapsulat restul de 12 clase, pe langa acestea, mai am clasa main.

In controller se afla subclasele pentru ascultatori, mai exact action listener, doua la numar: startActionListener si submitActionListener, care au rol in simularea butoanelor, adica asteapta apasarea noastra.

In model se afla implementate functionalitatile pentru rularea controller-ului: sunt 9 la numar: ConcreteStrategyQueue, ConcreteStrategyTime, Files, Scheduler, SelectionPolicy, Server, SimulationManager, Strategy, Task.

In view se afla clasa view, este implementata legatura dintre calculator si utilizatorul macestuia.

Pe langa acestea, mai exista clasa Main.

Pentru crearea interfetei m folosit java swing.

## 5. Implementare

Clasele acestui proiect au fost implementate in cel mai intuitiv mot considerat de mine.

Urmeaza descrierea detaliata a claselor:

Clasa Task:

Incepe cu extensia comparable care ma ajuta in ordonarea cozii de clienti. Contine 3 variabile: timpul de ajungere , un timp de sosire care reprezint momentul in care clientul ajunge la finalul cozi, timpul de iesire din coada determinat de momentul in care clientul ajunge la casierie, perioada de procesare care reprezinta timpul de care este nevoie pentru a reusi sa servim clientul cu marfa dorita, si id-ul clientului, fiind numarul dat unui anumit client in momentul in care el ajunge in coada. Pentru realizare acestor functii avem nevoie de un constructor pentru initializarea functiilor. Avem diverse functii cu nume sugestiv folosite pentru returnarea diverselor valori de care avem nevoie. Tot in aceasta clasa se afla suprascrierea metodei comparable care ajuta la o sortare mai usoara pentru utilizator, atat pentru scrierea in fisier.

```
package model;

public class Task implements Comparable<Task> {
    private int arrivalTime=-1;
    private int finishTime=-1;
    private int processingPeriod=-1;
    private int id;

    public Task() {

    }

    public Task(int arrivingTime, int processingTime, int id)
    {
        this.arrivalTime = arrivingTime;
        this.processingPeriod = processingTime;
        this.id=id;
    }
}
```

Clasa server

Incepe cu initializarea celor 4 variabile cu nume sugestiv: id: un numar de ordine a coziilor; avem nevoie de acesta pentru ca fiecare coada va rula pe un thread diferit, un BlockingQueue care este un sir de taskuri implementat astfel ca sa tina rolul unei masuri de siguranta penru utilizarea in mod corect a threaduri. WaitingTime: timpul de asteptare este crescut de fiecare data cand adaugam un task in aceasta coada, total waiting time timpul de procesare a asteptarii in coada a clientului respectiv. Cea mai importanta metoda de aici este run() unde luam persoana de la inceputul cozii si ii modificam timpul de procesare, prin scaderea acestuia, , scazand in acelasi timp si timpul de asteptare. Daca persoana care se afla la casa are timpul de procesare egal cu zero inseamna ca am terminat cu ea si trebuie scoasa din coada. Avem diferite functii penru realizarea cerintelor cerute, de exemplu add task

ne creeaza clienti noi si ne ofera timpul acestora de procesare. Am suprascris metoda toString() tot pentru a putea afisa placut in interfata si in fisier.

```
public class Server extends TimerTask implements Runnable
{
    private final int id;
    private int waitingTime;
    private int totalWaitingTime;
    private BlockingQueue<Task> tasks;
    public Server(int id, int maxqueue){
        this.id=id;
        this.waitingTime=0;
        this.totalWaitingTime=0;
        this.tasks = new ArrayBlockingQueue<>(maxqueue);
    }

    @Override
    public void run()
    {
```

Clasa Scheduler.

Aceasta clasa are rolul de a implementa felul in care clientii sunt asezati in coada, semanand la perfectie cu cozile de clienti din viata reala si cu modul lor de functionare. Strategia acestei clase este inspirata tot din viata reala, clasa a fost implementata in asa fel incat clientul nou sa fie asezat in coada cu cei mai putini oameni sau cu timpul cel mai mic de asteptare.

Acest lucru a fost posibil datorita enumeratiei SelectionPolicy care contine 2 variabile cu nume sugestiv: shortest\_queue si shortest\_time

```
public enum SelectionPolicy {
    shortest_queue, shortest_time
}
```

Tot in aceasta clasa am folosit functia string pentru afisarea placuta a cozilor formate, si inca o functie pentru oprirea thread-urilor.

```
public class Scheduler {
    private List<Server> servers;
    private List<Thread> threads;
    private int nrServers;
    private int nrTasksPerServer;
    private Strategy strategy;
    SelectionPolicy policy;
    private int nn=0;
```

```
    public String toString() {
        String rezultat="";
        for (Server i: servers) {
            rezultat+= "Queue " + i.getId() + ": " + i.toString() + "\n";
            int nnn=i.getQueue_size();
            if (nnn>nn){
                nn=nnn;
            }
        }
    }
```

### Clasa ConcreteStrategyTime

În această clasă se implementează așezarea clienților într-o coadă care nu este goală după strategia explicată mai sus: timp mai scurt de așteptare sau clienți mai puțini. Această clasă se leagă de strategia tot explicată mai sus, focusându-se pe timpul de așteptare. La început am inițializat 2 variabile, una pentru timpul minim și id-ul clientului. Timpul minim este o variabilă foarte mare pentru a nu exista posibilitatea să fie depășită. Verific pentru fiecare client dacă timpul de așteptare este mai mic decât timpul minim, iar dacă e, îi atribui timpului minim valoarea timpului de așteptare și atribul clientului coadă cu proprietatea de mai sus. După ce se va termina procesul, se va trece la clientul următor. Mai avem funcția `add_task` care are ca parametrii toate cozile și noul client pe care îl vom adăuga.



```

import java.util.List;
public class ConcreteStrategyTime implements Strategy {
    public void add_Task (List<Server> servers, Task t)
    {
        int tmin = 1000000;
        int id = 0;
        for(Server i : servers)
            if(i.getWaitingTime() < tmin) {
                tmin = i.getWaitingTime();
                id = i.getId();
            }
        for(Server i : servers) {
            if(i.getId() == id) ;
            i.add_Task(t);
        }
    }
}

```

#### Clasa ConcreteStrategyQueue

In aceasta clasa se implementeaza asezarea clientilor intr-o coada care nu este goala dupa strategia explicata mai sus: timp mai scurt de asteptare sau clienti mai putini. Aceasta clasa se leaga de strategia tot explicata mai sus, focusandu-se pe cea mai scurta coada. La inceput am initializat 2 variabile, una pentru timpul minim si id-ul clientului. Timpul minim este o variabila foarte mare pentru a nu exista posibilitatea sa fie depasita. Verific pentru fiecare client daca timpul de asteptare e mai mic decat timpul minim, iar daca e, ii atribui timpului minim valoarea timpului de asteptare si atribul clientului coada cu proprietatea de mai sus. Dupa ce se va termina procesul, se va trece la clientul urmator. Mai avem functia add\_task care are ca parametrii toate cozile si noul client pe care il vom adauga.

```

import java.util.*;

public class ConcreteStrategyQueue implements Strategy {
    public void add_Task (List<Server> servers, Task t)
    {
        int qmin = 1000000;
        int id = 0;
        for(Server i : servers)
            if(i.getTasks().size() < qmin) {
                qmin = i.getTasks().size();
                id = i.getId();
            }
        for(Server i : servers) {
            if(i.getId() == id) {
                i.add_Task(t);
            }
        }
    }
}

```

### Clasa SimulationManager

Aceasta este clasa de simulare propriu-zisa. Incepe cu declararea tuturor variabilelor necesare, acestea avand nume sugestive: nrClients care e numarul clientilor, nrServers care e numarul de Server-e, tMaxSimulation reprezinta timpul maxim de simulare, tMinArrival este timpul minim de sosire, iar tMaxArrival e timpul maxim de sosire, tMinService e timpul minim de asteptare iar tMaxService e timpul maxim, avem de asemenea timpul mediu de asteptare si timpul mediu de procesare, nr de persoane si ora de preluare a clientului. Se da lista de clienti, selectarea clientilor fisierul cu datele.

Avem o functie pentru crearea fisierului, una care da client random, una pentru timpii de asteptare, timpii de procesare, timpii de sosire.

Metoda run este o functie foarte importanta in aceasta clasa. Functioneaza dupa momentul de timp actual, clientii generate, astfel se scriu clientii in fisier.

```

public class SimulationManager extends Files implements Runnable {
    public int nrClients = 100;
    public int nrServers = 2;
    public int tMaxsimulation ;
    public int tMinArrival = 2;
    public int tMaxArrival= 30;
    public int tMinService = 2;
    public int tMaxService= 10;
    public double avgerageTime=0;
    public double avgerageProcesingTime=0;
    public int pick=0;
    int nrPers=0;
    String result;
}

```

Clasa Controller.

Face parte din pachetul cu acelasi nume. In aceasta clasa avem cate o subclasa pentru fiecare ascultator de buton. Prin aceasta clasa se face legatura intre View si Model.

Pentru functionarea corecta se va apasa prima oara pe butonul submit care , in spatele acestui buton se afla fiecare string introdus in casutele text specific locului lor. La final se va reseta view-ul, dupa se va porni simularea propriu-zisa.

```
public class Controller {  
    private Thread t;  
    private View view;  
    private SimulationManager simulator;  
  
    public Controller(SimulationManager s, View v){  
        this.view=v;  
        this.simulator=s;  
        t = new Thread(s);  
        v.submitListener(new submitActionListener());  
        v.startListener(new startActionListener());  
    }  
}
```

Clasa View

Aceasta clasa reprezinta tot ce inseamna implementarea interfetei grafice facute in acest caz cu java swing.

Clasa incepe prin declararea casutelor de text necesare pentru toate argumentele cerute in problema, si un scrollPane pentru a putea naviga cu usurinta in fereastra generate pentru interfata. Casutele de text au nume sugestive, pentru numarul de client, numarul de cozi, timpul simularii, timpul minim si maxim de sosire, si de procesare, pentru a vedea cat dureaza punerea clientilor in cozi. Se vor genera random timpii de sosire pentru fiecare client si se va afisa atat pe ecran cat si pe fereastra interfetei evolutia cozii si a timpului.

```

public class View extends JFrame {
    private JTextField N = new JTextField("4");
    private JTextField Q = new JTextField("2");
    private JTextField timeLim = new JTextField("60");
    private JTextField tMinA = new JTextField("2");
    private JTextField tMaxA = new JTextField("30");
    private JTextField tMinS = new JTextField("2");
    private JTextField tMaxS = new JTextField("4");
    private JButton submit = new JButton(text: "SUBMIT");
    private JButton start = new JButton(text: "START");
    private JTextPane Rezultat = new JTextPane();
    JScrollPane scrollPane = new JScrollPane(Rezultat);
}

```

Clasa Main

In aceasta clasa avem parametrul destinate pentru view si controller, de aici executandu-se rularea programului.

## 6. Rezultat

Rezultatul e dat de rularea celor 3 teste, voi pune rezultatele acestora:

Pentru primul test:

```

Average waiting time: 2.25
Average service time: 2.5
Pick Hour: 6

```

Pentru al 2 lea test:

```

Average waiting time: 3.34
Average service time: 3.36
Pick Hour: 39

```

Pentru al 3 lea test :

```
Average waiting time: 5.498  
Average service time: 5.499  
Pick Hour: 99
```

## 7. Concluzii

Acest proiect m-a ajutat sa imi dezvolt mai tare abilitatile de scriere in Java. Pentru acest proiect am avut nevoie de o perioada mai lunga de gandire fiind prima oara cand lucrez cu thread-uri.

Ca si dezvoltari ulterioare, consider ca s-ar putea implementa acest algoritm in viata reala, pentru gestionarea cozilor la supermarket, la nivelul unui aeroport sau in alte locuri foarte aglomerate, fiind implementat cu niste cozi automatizate care sa redirectioneze clientii catre coada cu timpul minim de asteptare.

## 8. Bibliografie

M-am ajutat atat de Wikipedia cat si de site-uri de programare: [Geekforgeeks](#)