

# DOCUMENTATIE

## TEMA 2

UTILIZAREA APLICATIEI DE GESTIUNE A  
COZILOR,  
FIRE SI MECANISME DE SINCRONIZARE

NUME STUDENT: Filip-Dud Cristian Călin  
GRUPA: 30228

## CUPRINS

---

1. Obiectivul temei.....	3
2. Analiza problemei, modelare, scenarii, cazuri de utilizare.....	4
3. Proiectare .....	6
4. Implementare .....	10
5. Rezultate .....	17
6. Concluzii .....	17
7. Bibliografie.....	18

# 1. Obiectivul temei

---

## Obiectivul principal:

Proiectați și implementați o aplicație care are ca scop analiza sistemelor bazate pe cozi de așteptare prin simularea a o serie de  $N$  clienți care sosesc pentru servicii, intră în  $Q$  cozi, așteaptă, sunt serviți și în cele din urmă părăsesc cozile, și calcularea timpului mediu de așteptare, a timpului mediu de serviciu și a orei de vârf.

## Obiective secundare:

- Analizarea problemei și identificarea cerințelor
- Proiectarea aplicației de simulare
- Implementarea aplicației de simulare
- Testarea aplicației de simulare

## 2. Analiza problemei, modelare, scenarii, cazuri de utilizare

---

### Cerinte functionale:

- Aplicatia de simulare trebuie sa permita utilizatoriilor sa initializeze simularea
- Aplicatia de simulare trebuie sa permita utilizatoriilor sa porneasca simularea
- Aplicatia de simulare trebuie sa afiseze evolutia cozilor in timp real

### Cerinte nonfunctionale:

- Aplicatia de simulare trebuie sa fie intuitiva si usor de folosit de catre utilizator

### Cazul de utilizare:

**Use Case:** simularea cozilor, vizualizarea evolutiei simularii, vizualizarea statisticilor obtinute

**Actor principal:** utilizatorul

**Scenariul principal de succes:**

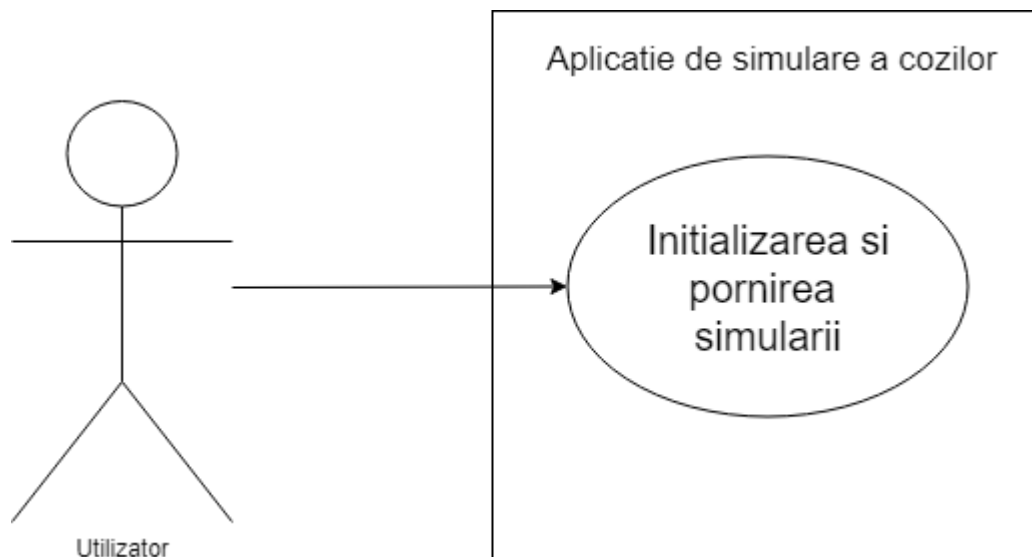
1. Utilizatorul introduce valorile pentru numărul de cozi, numărul de clienți, durata simulării, timpul minim și maxim de sosire, timpul minim și maxim de servire;

2. Utilizatorul apasă butonul de Start;

3. Sunt preluate datele din interfata si incepe simularea, afisand in timp real cozile

**Secvența alternativă:** Datele introduse sunt invalide

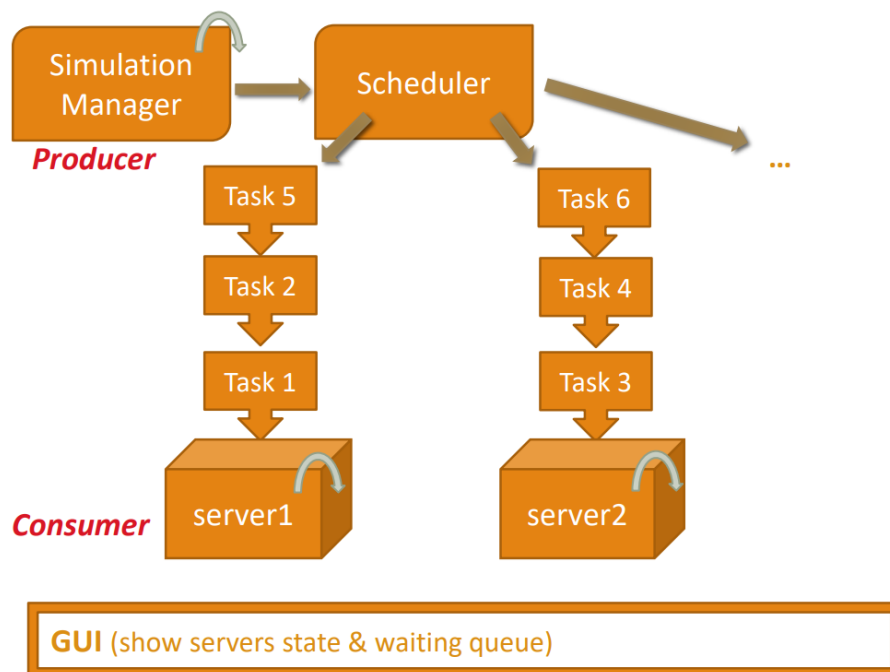
- Utilizatorul a introdus date invalide pentru parametrii simulării (e.g. maximul timpului de sosire este mai mic ca minimul acestuia)
- Aplicația afișează un mesaj de eroare si cere utilizatorului sa introducă date valide;
- Scenariul se întoarce la pasul 1.



## 3. Proiectare


---

### Proiectarea POO



Pachetele in care este structurata aplicatia sunt: GUI (implementeaza interfata GUI a aplicatiei), BusinessLogic (implementeaza modurile de inserare a clientilor in cozi si initializarea acestora cu datele necesare) si Model (contine clasele clienti si cozi care simuleaza aceste lucruri).

Interfata contine campuri de text pentru a introduce datele necesare initializarii simularii, doua zone unde sunt afisate datele in timp real ale simularii si un buton de start.

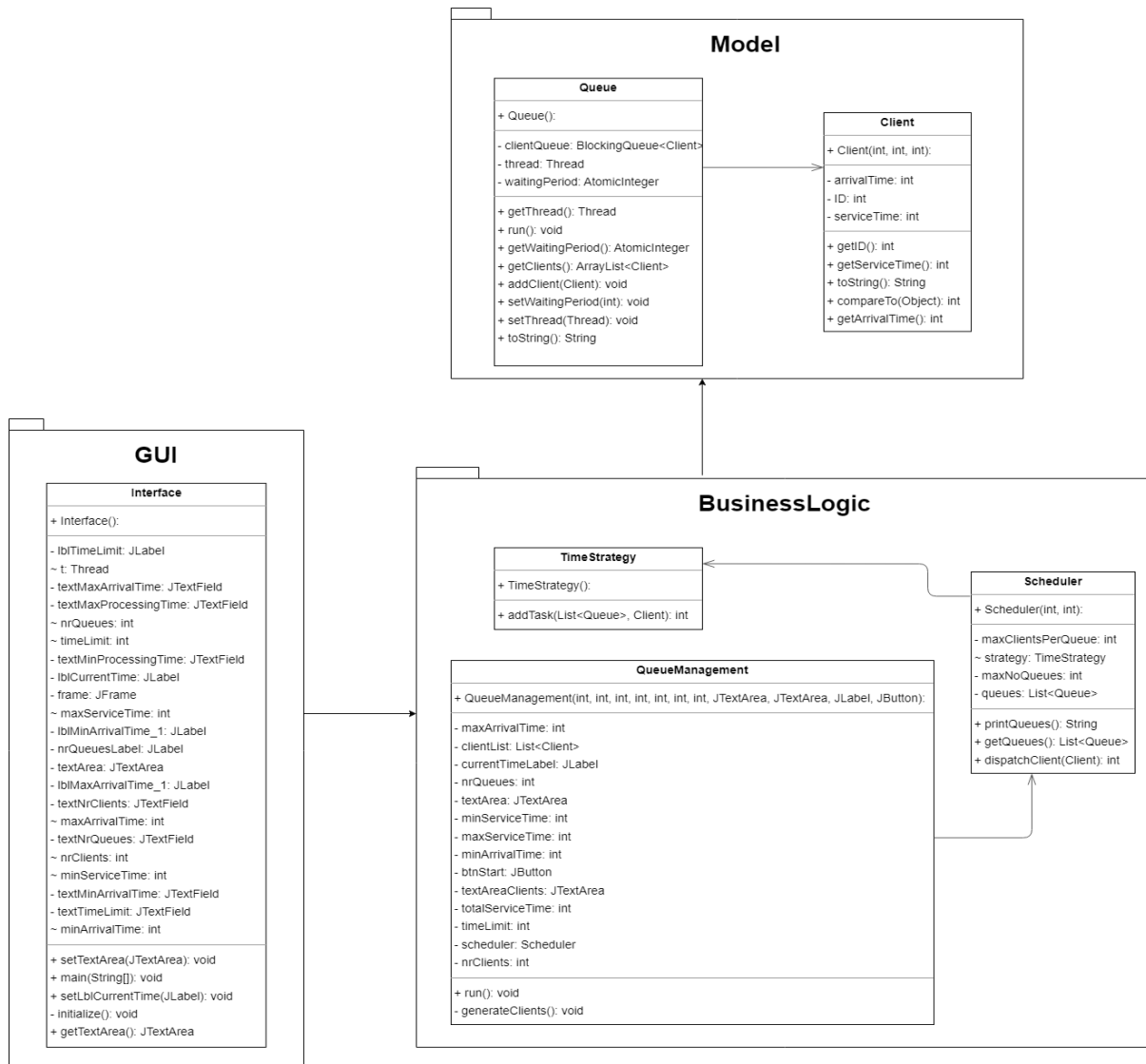


The image shows a screenshot of a software interface for a simulation. The window has a teal header bar with standard window controls (minimize, maximize, close) in the top right corner. Below the header, the interface is divided into several sections. At the top, there are four input fields for initialization parameters: 'Nr of queues', 'Nr of clients', 'Min arrival time', and 'Max arrival time'. Below these are two more input fields: 'Time limit' and 'Min processing time', followed by 'Max processing time'. A 'START' button is located to the right of the first row of input fields. Below the input fields, there is a label 'Current time:' followed by a large, empty rectangular area, likely for displaying real-time simulation data. The entire interface is enclosed in a teal border.

Nr of queues	Nr of clients	Min arrival time	Max arrival time	START
<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	
Time limit	Min processing time		Max processing time	
<input type="text"/>	<input type="text"/>		<input type="text"/>	

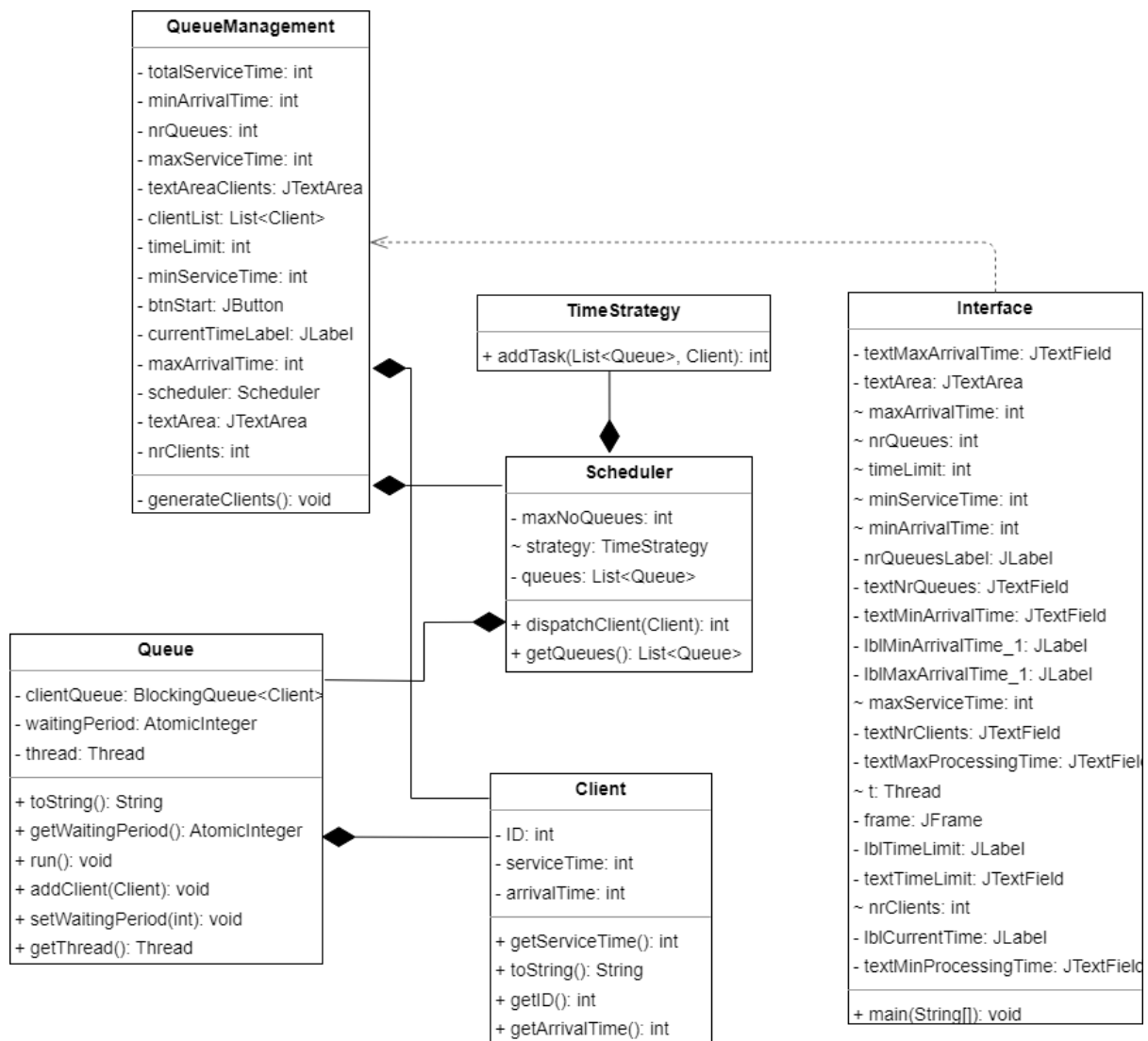
Current time:

Diagrama UML de pachete:





### Diagrama UML de clase:



Structurile date folosite pe care se bazeaza aplicatia sunt colectiile din Java:

- `ArrayBlockingQueue<Client>`: salveaza clientii intr-o coada si asigura faptul ca datele stocate sunt sincronizate in cadrul utilizarii firelor de executie pentru accesarea simultana de informatii de catre acestea.

- `AtomicInteger`: utilizat pentru retinerea timpului de asteptare al unei cozi, asigurand sincronizarea datelor in cadrul utilizarii firelor de executie.

-ArrayList<Queue>: salveaza cozile utilizate pentru stocarea clientilor, asigurand accesibilitate si eficienta ridicata in cadrul memorarii informatiei .

Algoritmul definit pentru introducerea in cozi a clientilor este unul simplu, acesta calculand timpul minim de asteptare dintre toate cozile, returnand acea coada pentru inserearea clientului.

## 4. Implementare

---

Aplicatia este structurata in mai multe pachete, primul fiind pachetul Simulation ce contine clasele necesare simularii: Client, Queue, TimeStrategy, Scheduler, QueueManagement, Interface.

Clasa **Client** reprezinta unitatea care detine datele unui client:

- arrivalTime – reprezinta momentul temporal al simularii in care task-ul trebuie atribuit unui server;
- serviceTime – ne spune cat de mult dureaza procesarea taskului de catre un server, adica cat timp va fi ocupat acesta cu rezolvarea taskului;
- ID – identificatorul unic al fiecarui task;

Metodele clasei sunt gettere, settere, toString si compareTo.

```
public class Client implements Comparable{
    private int ID, arrivalTime, serviceTime;

    public Client(int ID, int arrivalTime, int processingTime) {
        this.ID = ID;
        this.arrivalTime = arrivalTime;
        this.serviceTime = processingTime;
    }

    public int compareTo(Object o) {

        if( ((Client)o).getArrivalTime() > this.arrivalTime)
            return -1;
        if( ((Client)o).getArrivalTime() < this.arrivalTime)
            return 1;
```

```
        return 0;
    }
    public int getID() {
        return ID;
    }

    public int getArrivalTime() {
        return arrivalTime;
    }

    public int getServiceTime() {
        return serviceTime;
    }

    @Override
    public String toString() {
        return "(" + ID + ", " + arrivalTime + ", " + serviceTime + ")";
    }
}
```

Clasa **Queue** care implementeaza interfata Runnable defineste o coada in care se insereaza clientii, definind un fir de executie particular unei cozi. Atributele clasei sunt:

- clientQueue – este o coada creata cu o colectie de tipul ArrayBlockingQueue pentru a sincroniza accesul la date si stoca clienti
- waitingPeriod – este un AtomicInteger care retine in timp real timpul ramas pana la terminarea procesarii clientilor din coada respectiva
- Thread thread – firul de executie asociat unei cozi

Metodele clasei sunt addClient (adauga un nou client in clientQueue), run (proceseaza pe rand clientii din coada), gettere, settere si toString.

```
public void addClient(Client newClient)
{
    clientQueue.add(newClient);
    waitingPeriod.addAndGet(newClient.getServiceTime());
}

public void run() {
    while(true)
    {
        if (clientQueue.size()>0)
        {
            Client x = this.clientQueue.peek();

```

```
        try {
            thread.sleep(x.getServiceTime()*1000);
        } catch (InterruptedException e) {
        }
        clientQueue.remove();
    }
}

public ArrayList<Client> getClients()
{
    ArrayList<Client> clients = new ArrayList<Client>();
    for (Client c: clientQueue)
    {
        clients.add(c);
    }
    return clients;
}
```

Clasa **TimeStrategy** implementeaza o metoda, `addTask`, care are rolul de a gasi coada cu timpul minim de asteptare si a adauga clientul transmis ca parametru in aceasta:

```
public int addTask(List<Queue> queues, Client c)
{
    int minWaitingPeriod = queues.get(0).getWaitingPeriod().get();
    Queue minQueue = queues.get(0);
    for (Queue q: queues)
    {
        if(q.getWaitingPeriod().get() < minWaitingPeriod)
        {
            minWaitingPeriod = q.getWaitingPeriod().get();
            minQueue = q;
        }
    }
    minQueue.addClient(c);
    return queues.indexOf(minQueue);
}
```

Clasa **Scheduler** se ocupa cu managementul cozilor si cu atribuirea clientilor in cozi cu timpul cel mai scurt de asteptare. Atributele clasei sunt:

- `queues` – este lista de servere a simularii, aceasta se initializeaza in constructor in functie de numarul de cozi dorite.
- `maxNoQueues` – retine numarul de cozi
- `strategy` – este un obiect de tipul clasei `TimeStrategy` care distribuie clientii in cozi

Metodele clasei sunt `dispatchClient` (apeleaza metoda `addTask` din `TimeStrategy`), `printQueues` ( afiseaza toate elementele din cozi) si `getQueues`.

```
public Scheduler(int maxNoQueues) {
    this.maxNoQueues = maxNoQueues;
}
```

```

this.queues = new ArrayList<Queue>();
this.strategy = new TimeStrategy();

int k=0;
while(k < maxNoQueues){
    queues.add(new Queue());
    k++;
}

public int dispatchClient(Client c)
{
    return strategy.addTask(queues, c);
}

public List<Queue> getQueues() {
    return queues;
}

public String printQueues()
{
    String queuesString = new String();
    for (Queue q: queues)
    {
        queuesString = queuesString + "Queue " + (queues.indexOf(q) + 1) +
", ";
        queuesString = queuesString + "waiting period: " +
q.getWaitingPeriod() + ": ";
        for(Client c : q.getClients())
        {
            queuesString = queuesString + " " + c + " ";
        }
        queuesString = queuesString + "\n";
    }
    queuesString = queuesString + "\n";
    return queuesString;
}

```

Clasa **QueueManagement** reprezinta clasa principala a simulării care initializează și rulează simularea în metoda run. Ea creează clienții și cozile prin intermediul unui obiect Scheduler. Ea are următoarele atribute:

```

private int nrClients, nrQueues, minArrivalTime, maxArrivalTime;
private int minServiceTime, maxServiceTime, totalServiceTime;
private int timeLimit;
private List<Client> clientList;
private Scheduler scheduler;
private JTextArea textArea;
private JTextArea textAreaClients;
private JButton btnStart;
private JLabel currentTimeLabel;

```

Atributele de tipul Jswing sunt folosite pentru sincronizarea cu interfața grafică.

Metodele principale sunt:

- generateTasks – aceasta metoda se ocupa cu generarea clientilor, fiecare client primind un id, timp de sosire si timp de procesare, timpii fiind generati folosind Math.random

```
private void generateClients()
{
    ArrayList<Client> clientArrayList = new ArrayList<Client>();
    for(int i = 1; i <= nrClients; i++)
    {
        int randomArrivalTime, randomServiceTime;
        randomArrivalTime = (int)Math.floor(Math.random()
* (maxArrivalTime - minArrivalTime + 1) + minArrivalTime);
        randomServiceTime = (int)Math.floor(Math.random()
* (maxServiceTime - minServiceTime + 1) + minServiceTime);
        Client tempClient = new Client(i, randomArrivalTime,
randomServiceTime);
        clientArrayList.add(tempClient);
        totalServiceTime = totalServiceTime + randomServiceTime;
    }
    Collections.sort(clientArrayList);
    clientList = new ArrayList<>();
    System.out.println("LISTA CLIENTILOR:\n");
    for (Client c: clientArrayList)
        System.out.println(c + "\n");
    System.out.println("\n");
    clientList.addAll(clientArrayList);
}
```

- run() – metoda principala a simulării, aceasta asteapta primirea datelor de initializare, genereaza clientii, si incepe simularea propriu-zisa. Se afiseaza in interfata si in consola in timp real cozile si clientii ramasi pentru procesare. La final sunt listate rezultatele simulării.

```
public void run() {

    int realTime = 0;
    int queueId;
    int peakTimeClients = 0, peakTime = 0;
    float avgWaitingTime = 0, avgProcessingTime = 0;

    try {
        generateClients();
        FileWriter myWriter = new FileWriter("log.txt");

        while(realTime <= timeLimit) {

            ArrayList<Client> removeClients = new ArrayList<>();

            for (Client c : clientList)
            {
                if (c.getArrivalTime() == realTime)
                {
                    queueId = scheduler.dispatchClient(c);
                    removeClients.add(c);
                    avgWaitingTime = avgWaitingTime +
(scheduler.getQueues().get(queueId).getWaitingPeriod().get() -
```

```

c.getServiceTime());
        avgProcessingTime = avgProcessingTime +
c.getServiceTime();
    }
}
for (Client c: removeClients)
{
    clientList.remove(c);
}

System.out.println("TIMPUL CURENT: " + realTime + "\n");
myWriter.write("TIMPUL CURENT: " + realTime + "\n");

myWriter.write("Clienti: " + clientList.toString()+"\n");
textAreaClients.setText(clientList.toString());
System.out.println(scheduler.printQueues());
textArea.setText(scheduler.printQueues());
myWriter.write(scheduler.printQueues());

currentTimeLabel.setText("Current time: " + realTime);

int maxClients = 0;
for (Queue q : scheduler.getQueues()) {
    maxClients = maxClients + q.getClients().size();
}
if (maxClients > peakTime) {
    peakTimeClients = maxClients;
    peakTime = realTime;
}

for (Queue q : scheduler.getQueues()) {
    if (q.getWaitingPeriod().get() > 0)
        q.setWaitingPeriod(q.getWaitingPeriod().get() -
1);
}

realTime++;

try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    e.printStackTrace();
}

}

avgWaitingTime = avgWaitingTime/nrClients;
avgProcessingTime = avgProcessingTime/nrClients;

myWriter.write("Timpul mediu de asteptare a fost: " +
avgWaitingTime + "\n" +
        "Timpul mediu de procesare a fost: " +
avgProcessingTime + "\n" +
        "Ora de varf a fost: " + peakTime + "cu un
numar de " + peakTimeClients + " clienti" + "\n");

textArea.setText("Timpul mediu de asteptare a fost: " +
avgWaitingTime + "\n" +
        "Timpul mediu de procesare a fost: " +

```

```
avgProcessingTime + "\n" +
        "Ora de varf a fost: " + peakTime + " cu un
numar de " + peakTimeClients + " clienti" + "\n");

    myWriter.close();
    for (Queue sv: scheduler.getQueues()) {
        sv.getThread().interrupt();
    }
    btnStart.setEnabled(true);
} catch (IOException e) {
    throw new RuntimeException(e);
}
}
```

Clasa **Interface** implementeaza interfata grafica. Contine atribute Jswing si metoda main. Butonul de start are un ActionListener, care atunci cand este apasat preia datele din campurile de text, creaza un obiect de tipul QueueManagement, creaza un thread cu acesta si il porneste.

```
btnStart.addActionListener(new ActionListener()
{
    @Override
    public void actionPerformed(ActionEvent e) {
        try {
            timeLimit = Integer.parseInt(textTimeLimit.getText());
            nrQueues = Integer.parseInt(textNrQueues.getText());
            nrClients = Integer.parseInt(textNrClients.getText());
            minArrivalTime =
Integer.parseInt(textMinArrivalTime.getText());
            maxArrivalTime =
Integer.parseInt(textMaxArrivalTime.getText());
            minServiceTime =
Integer.parseInt(textMinProcessingTime.getText());
            maxServiceTime =
Integer.parseInt(textMaxProcessingTime.getText());

            if (timeLimit < 0 || nrQueues < 0 || minArrivalTime < 0 ||
maxArrivalTime < 0 || minServiceTime < 0 || maxServiceTime < 0)
                throw new Exception();
            if (maxArrivalTime < minArrivalTime || maxServiceTime <
minServiceTime || maxArrivalTime > timeLimit)
                throw new Exception();

            btnStart.setEnabled(false);
            QueueManagement sim = new QueueManagement(timeLimit, nrClients,
nrQueues, minArrivalTime, maxArrivalTime, minServiceTime, maxServiceTime,
textArea, textAreaClients, lblCurrentTime, btnStart);
            t = new Thread(sim);
            t.start();
        } catch (Exception ex)
        {
            JOptionPane.showMessageDialog(null, "Date de intrare
invalide!", "Error", JOptionPane.ERROR_MESSAGE);
        }
    }
});
```



## 5. Rezultate

---

Testul 1:

```
Timpul mediu de asteptare a fost: 0.0  
Timpul mediu de procesare a fost: 2.5  
Ora de varf a fost: 20cu un numar de 1 clienti
```

Testul 2:

```
Timpul mediu de asteptare a fost: 1.06  
Timpul mediu de asteptare a fost: 98.992  
Timpul mediu de procesare a fost: 5.901  
Ora de varf a fost: 200cu un numar de 361 clienti
```

Testul 3:

## 6. Concluzii

---

Dezvoltarea acestei teme m-a ajutat sa invat despre conceptul de multithreading, sincronizarea threadurilor si a variabilelor, de asemenea am aprofundat conceptele de baza ale POO.

În ceea ce privește modalitățile de dezvoltare ulterioară a aplicației, acestea ar putea fi implementate pentru mai multe simulări în paralel, acestea reprezentând mai multe magazine și cozile aferente lor.

## 7. Bibliografie

---

- ❖ <https://www.educative.io/answers/how-to-generate-random-numbers-in-java>
- ❖ <https://app.diagrams.net/>