

# **DOCUMENTATIE**

## **TEMA 2**

NUME STUDENT: HOBAN CRISTIAN MIHAI  
GRUPA: 30227

# **CUPRINS**

<b>CUPRINS</b>	<b>2</b>
<b>1. Obiectivul temei</b>	<b>3</b>
<b>2. Analiza problemei, modelare, scenarii, cazuri de utilizare</b>	<b>3</b>
<b>3. Proiectare</b>	<b>4</b>
<b>4. Implementare</b>	<b>6</b>
<b>5. Rezultate</b>	<b>12</b>
<b>6. Concluzii</b>	<b>13</b>
<b>7. Bibliografie</b>	<b>13</b>

## 1. Obiectivul temei

Obiectivul Principal al temei îl reprezintă realizarea unui program care simulează modul în care un anumit număr de clienți trebuie să se împartă pe mai multe cozi pentru a fi serviți în funcție de timpul de așteptare a celorlalți.

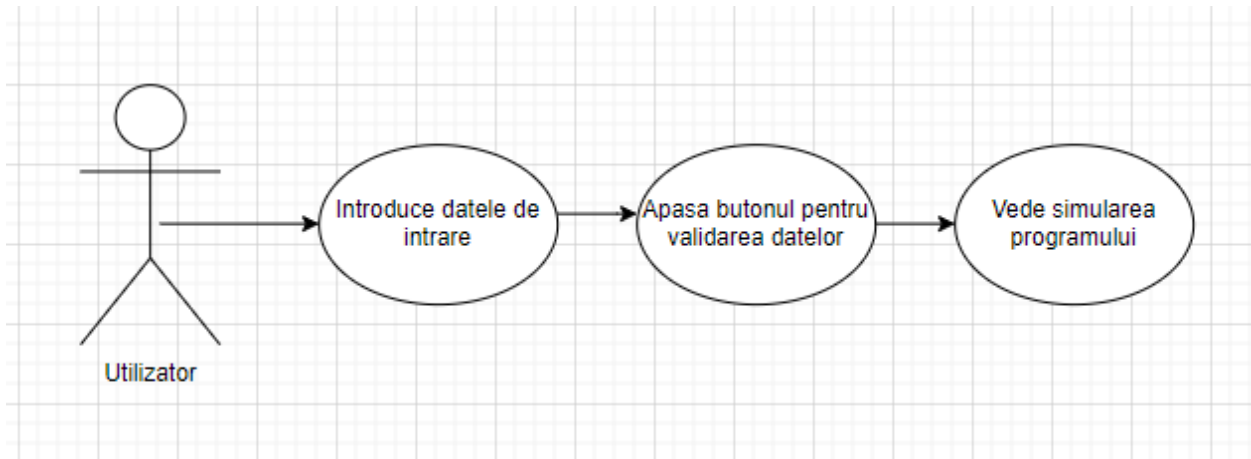
Există mai multe obiective secundare, care, realizate, vor duce la atingerea scopului final. Pe acestea le-am enumerat în tabelul de mai jos:

<i>Obiectiv</i>	<i>Descriere</i>	<i>Capitol în care poate fi găsit</i>
1. Definirea clasei Task care reprezintă un client	Această clasă va avea 3 atribute, ID-ul clientului, timpul la care ajunge, respectiv timpul de servire.	
2. Generarea aleatoare a unui număr de N clienți	Această funcție permite generarea unui număr N de clienți, acest număr fiind introdus de utilizator.	
3. Crearea unui simulator	Acesta va simula modul în care clienții se așează la cozi și ies din acestea	
4. Crearea unei interfețe grafice	Interfața grafică este alcătuită din 2 părți. Una în care utilizatorul va introduce date referitoare la clienți, iar a doua simularea propriu-zisă în funcție de datele introduse anterior	
5. Validarea datelor de intrare	Pentru a face posibilă simularea, trebuie să ne asigurăm că datele de intrare sunt valide, astfel fiecare câmp va fi verificat	

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

Aplicatia trebuie sa permita utilizatorului introducerea de la tastatura a a unui numar in fiecare camp a unui numar corepsunzator etichetei din dreptul lui. Apoi, aceasta trebuie sa verifie validitatea datelor, iar in final, aceasta trebuie sa afiseze simularea in timp real.

Diagrama use-case:



Pasii pe care utilizatorul trebuie sa ii urmeze:

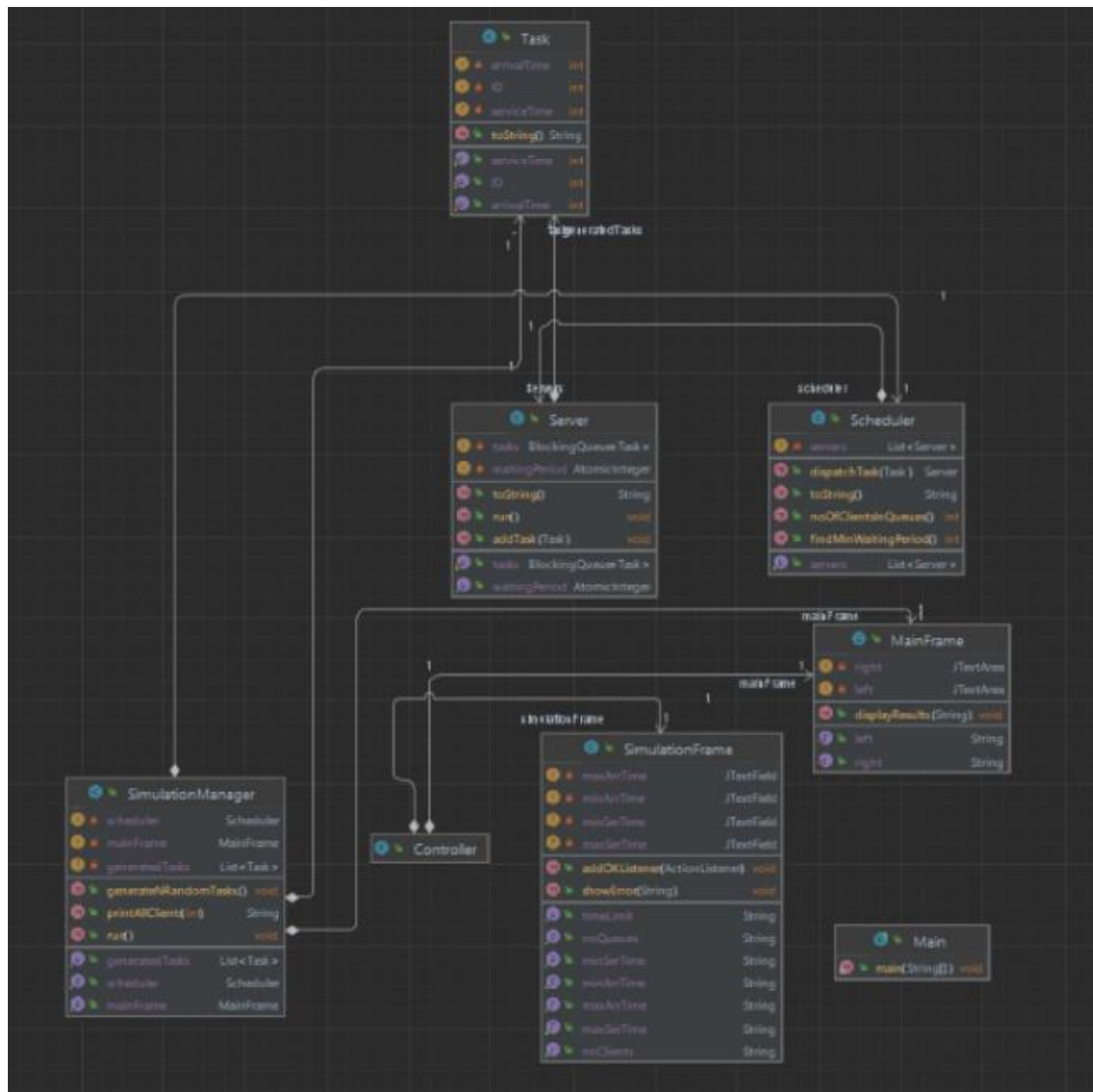
1. Introduce datele de intrare denumite “Number of clients”, “Number of queues”, “Simulation interval”, “Bound for arrival time”, “Bounds for service time”, ultimele 2 avand cate 2 campuri fiecare, acestea reprezentand un interval in cazul amandurora;
2. Apasa butonul “OK”, in urma acesteia, ii vor fi verificate datele pe care le-a introdus anterior
3. Vede fereastra nou-aparuta in care va porni simularea

### 3. Proiectare

Proiectarea OOP reprezinta un model de programare care ofera o buna capacitate de gestionare a unui program.

In acest capitol voi prezenta proiectarea OOP a aplicatiei, in prima faza cu ajutorul unei diagrame UML pentru a fi mai usoara intelegerea modului in care clasele sunt conectate intre ele.

**Diagrama UML:**



Dupa cum se observa din diagrama, aplicatia este structurata in mai multe clase, fiecare din ele apartinand unui pachet:

- **controller**

- Controller
- **logic**
  - Scheduler
  - SimulationManager
- **model**
  - Server
  - Task
- **org.example**
  - Main
- **view**
  - MainFrame
  - SimulationFrame

Pentru a putea implementa simularea cozilor, am avut nevoie in primul rand de a crea clientii, pentru acestia am creat clasa “Task”. Pentru a crea cozile am implementat clasa “Server” care contine un BlockingQueue de clienti(task-uri).

```

1 usages
public class Task {
    4 usages
    private int ID;
    4 usages
    private int arrivalTime;
    4 usages
    private int serviceTime;

    1 usage
    public Task(int ID, int arrivalTime, int serviceTime){
        this.ID = ID;
        this.arrivalTime = arrivalTime;
        this.serviceTime = serviceTime;
    }
}

```

## 4. Implementare

Dupa cum am specificat, am creat clasa “Task” pentru a genera un client si clasa “Server” pentru a genera o coada.

In clasa “Scheduler”, am initializat o lista de cozi(servere), iar in constructorul acestuia am creat atatea servere cate a specificat utilizatorul in datele de intrare, iar pentru fiecare din acestea am pornit cate un fir de executie(thread).

```

public Scheduler(int maxNoServers) {
    servers = new ArrayList<>();
    for (int i = 0; i < maxNoServers; i++) {
        Server s = new Server();
        Thread t = new Thread(s);
        t.start();
        this.servers.add(s);
    }
}

```

Tot in aceasta clasa, am creat functia “dispatchTask” care primeste ca parametru un client(task), si il introduce in coada cu cel mai mic timp de asteptare.

```

public Server dispatchTask(Task t){
    int i = this.findMinWaitingPeriod();
    for(Server s : this.servers){
        int aux = s.getWaitingPeriod().intValue();
        if(aux == i){
            s.addTask(t);
            return s;
            //break;
        }
    }
    return null;
}

```

In clasa “SimulationManager” am creat o metoda care genereaza aleator un numar de N clienti, numarul N fiind preluat din datele introduse de utilizator. Pentru a aloca valori aleatoare am folosit o instanta a clasei “Random”.

```
1 usage
public void generateNRandomTasks(){
    generatedTasks = new ArrayList<>();
    Random random = new Random();
    int arrivalBound = maxArrivalTime - minArrivalTime + 1;
    int serviceBound = maxServiceTime - minServiceTime + 1;
    for(int i = 0; i < numberOfClients; i++){
        int service = random.nextInt(serviceBound) + minServiceTime;
        int arrival = random.nextInt(arrivalBound) + minArrivalTime;
        Task t = new Task( ID: i+1, arrival, service);
        generatedTasks.add(t);
    }
}
```

Metoda “run” din aceasta clasa este metoda care se apeleaza cand principalul fir de executie este pornit. In aceasta, se ia pe rand, client cu client, si se introduc in cozile corespunzatoare cu ajutorul functiei “dispatchTask” prezentata mai sus. Pentru a reprezenta timpul in aceasta simulare, am folosit metoda “sleep” pentru a “ingheta” programul cu o secunda dupa fiecare afisare. Tot in cadrul acestei metode, pentru a afisa la final rezultatele simularii(average waiting time, average service time, peak hour), am introdus niste variabile care memoreaza aceste rezultate.



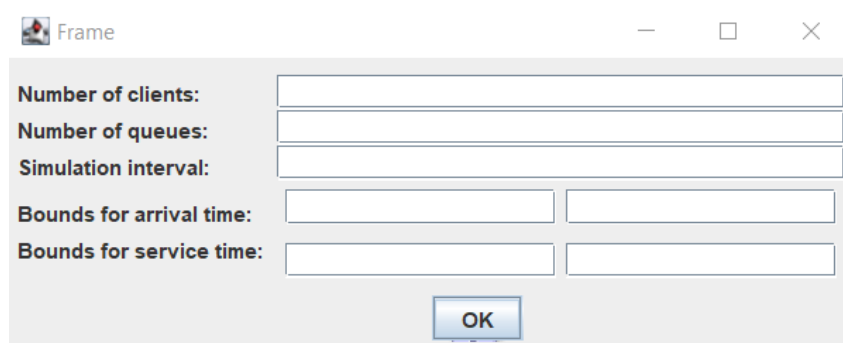
```

public void run() {
    double averageServiceTime = 0;
    double averageWaitingTime = 0;
    int noOfClients = 0;
    int currentTime = 0;
    int maxNoClients = -1;
    try {
        FileWriter file = new FileWriter( fileName: "logTest3.txt");
        while(currentTime <= timeLimit){
            for(Task e:generatedTasks){
                if(e.getArrivalTime() == currentTime){
                    Server s = scheduler.dispatchTask(e);
                    noOfClients++;
                    if(maxNoClients < scheduler.noOfClientsInQueues())
                        maxNoClients = scheduler.noOfClientsInQueues();
                    averageServiceTime += e.getServiceTime();
                    averageWaitingTime += s.getWaitingPeriod().doubleValue();
                }
            }
            mainFrame.setLeft(printAllClients(currentTime));
            String res = "Time " + currentTime + ":\n" + scheduler.toString();
            file.write(res);
            mainFrame.setRight(res);
            /*System.out.println("Time " + currentTime + ":\n");
            System.out.println(scheduler.toString());*/
            try {
                Thread.sleep( millis: 1000);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            currentTime++;
        }
        String output = "Average service time: " + averageServiceTime/noOfClients + "\n" +
            "Average waiting time: " + averageWaitingTime/noOfClients + "\n" +
            "Peak hour: " + maxNoClients;
        mainFrame.displayResults(output);
        file.write(output);
        file.close();
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}

```

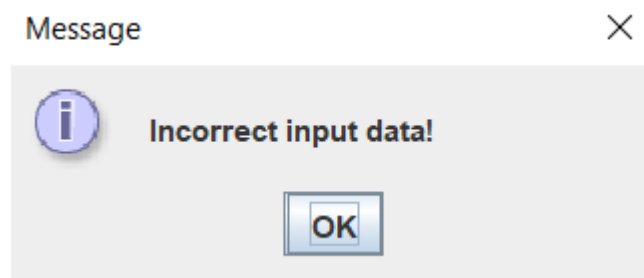
Interfata grafica am realizat-o cu ajutorul a 2 clase.

Prima dintre ele este “SimulationFrame” in care utilizatorul isi va introduce datele de intrare. Pentru campurile in care se introduc date am folosit “JTextField”, iar pentru etichetele campurilor am folosit “JLabel”. Pe aceasta interfata exista si un buton de “OK” care va trebui apasat dupa ce campurile au fost completate.

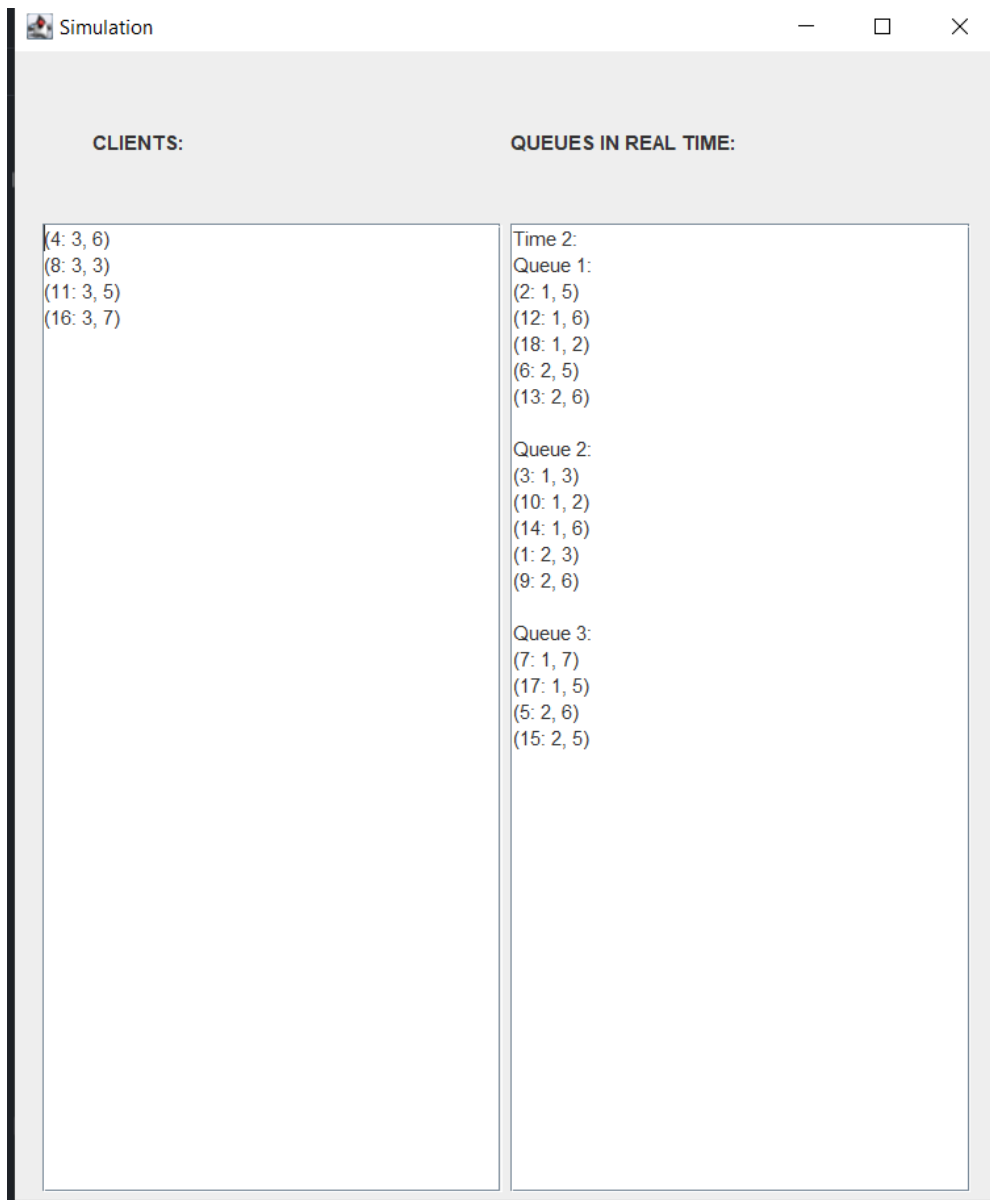


Daca datele introduse de utilizator nu sunt numere intregi, dupa apasarea pe buton va apare un mesaj de eroare.

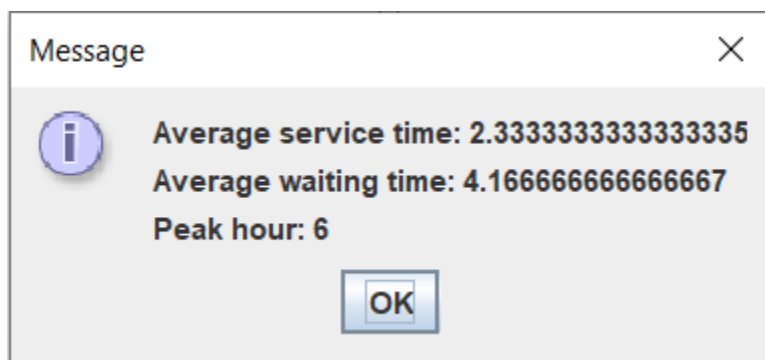
```
1 usage
public void showError(String errorMessage) { JOptionPane.showMessageDialog( parentComponent: this, errorMessage); }
```



Daca datele introduse sunt corecte, se va deschide o noua fereastră, realizata cu ajutorul celeilalte clase “MainFrame”. Aceasta clasa este alcatuita din 2 parti, existand posibilitatea de a da scroll in fiecare din ea in cazul in care textul afisat depaseste limitele chenarului. In partea din stanga vor fi afisati clientii care urmeaza sa intre in coada, iar in partea din dreapta modul in care cozile arata la un anumit timp. Ambele chenare se modifica dinamic, in momentul in care un client intra intr-o coada, acesta nu va mai aparea in stanga, iar in momentul in care un client iese din coada acesta dispare din partea dreapta.



La final, dupa ce timpul de simulare s-a terminat, va aparea un mesaj cu rezultatele simularii, cele pe care le-am mentionat mai sus.



Pentru a face legatura intre interfata grafica si partea functionala a aplicatiei, am implementat clasa "Controller" in care am creat un ascultator pentru butonul "OK". Pentru a verifica daca datele de intrare sunt intr-adevar niste intregi am aplicat asupra fiecarui string introdus de catre utilizator metoda predefinita "parseInt", pe toate acestea introducandu-le intr-un bloc "try" pentru a verifica daca cumva programul va arunca exceptie. Practic, aici este locul in cod unde porneste simularea dupa validarea datelor.

```
try{
    noClients = Integer.parseInt(simulationFrame.getNoClients());
    noQueues = Integer.parseInt(simulationFrame.getNoQueues());
    timeLimit = Integer.parseInt(simulationFrame.getTimeLimit());
    minA = Integer.parseInt(simulationFrame.getMinArrTime());
    maxA = Integer.parseInt(simulationFrame.getMaxArrTime());
    minS = Integer.parseInt(simulationFrame.getMinSerTime());
    maxS = Integer.parseInt(simulationFrame.getMaxSerTime());

    SimulationManager simulationManager = new SimulationManager(timeLimit, maxA, minA, minS, maxS, noQueues, noClients);
    simulationManager.getMainFrame().setVisible(true);
    Thread t = new Thread(simulationManager);
    t.start();
    simulationFrame.dispose();
}
catch(NumberFormatException ex){
    simulationFrame.showError( errorMessage: "Incorrect input data!");
}
}
});
```

## 5. Rezultate

Pentru a putea urmari toate modificarile realizate asupra cozilor, pe langa faptul ca se fac afisari in timp real in interfata, am facut o scriere in fisier pentru pentru fiecare timp. Astfel se poate urmari in detaliu modul in care simularea a functionat.

```
FileWriter file = new FileWriter( fileName: "logTest3.txt");
```

```
String res = "Time " + currentTime + ":\n" + scheduler.toString();
file.write(res);
```

Cerinta specifica 3 teste concrete pe care sa le verificam pe programul nostru, aceste 3 rezultate scriindu-le pe fiecare intr-un fisier separat: "logTest1.txt", "logTest2.txt", "logTest3.txt".

## 6. Concluzii

Pe parcursul realizării acestui proiect, am înțeles modul în care firele de execuție (thread-urile) funcționează, lucru care la început îmi era destul de ambiguu. Ce a mai fost nou a fost scrierea în fișier în cod JAVA, lucru pe care nu l-am folosit până acum.

Ca dezvoltări ulterioare m-am gândit la câteva:

- realizarea unei interfețe în care să fie mai ușoară urmărirea fiecărui client prin înlocuirea acestuia cu un obiect geometrică spre exemplu;
- verificarea datelor de intrare ca numerele din câmpurile pentru intervalele de timp să fie perfect corecte; în sensul în care eu am verificat doar dacă acesta este un număr. Nu am mai verificat dacă primul este mai mic ca următorul.

## 7. Bibliografie

1. <https://dsrl.eu/courses/pt/>
2. <https://www.geeksforgeeks.org/queue-interface-java/>