



# Documentație

## Tema 2: Aplicație pentru gestionarea clienților la cozi într-un magazin



# Cuprins

1. Obiective .....	3
2. Analiza problemei .....	4
3. Proiectare .....	5
4. Implementare .....	7
5. Concluzii și dezvoltări ulterioare .....	11
6. Bibliografie .....	11



# 1. Objective

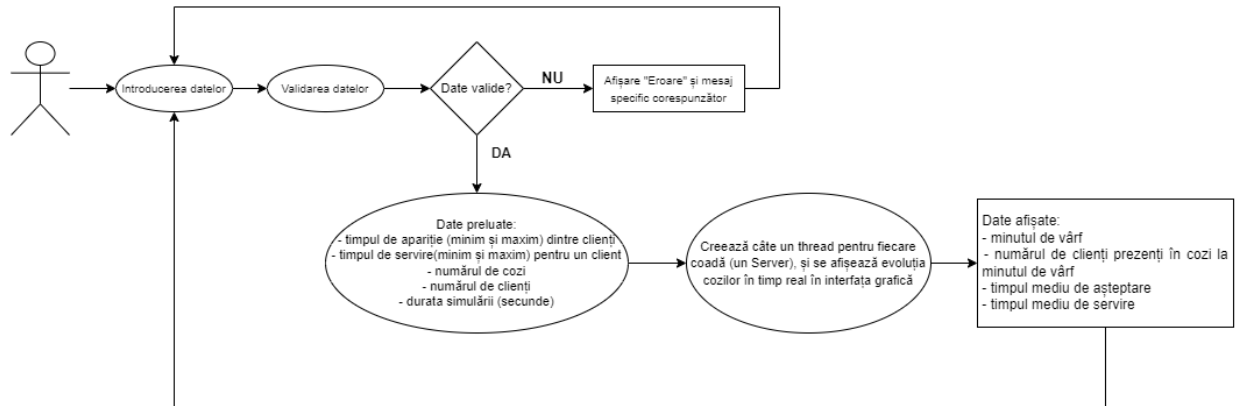
## 1.1 Obiectiv principal

Proiectați și implementați o aplicație de simulare care vizează analiza sistemelor bazate pe cozi pentru determinarea și minimizarea timpului de așteptare al clienților.

## 1.2 Obiective secundare

Obiectiv	Detaliere	Prezentat în capitolul
Dezvoltarea de <i>use cases</i> și scenarii	Reprezentarea diagramelor <i>use case</i> , precum și exemplificarea unor operații implementate	2
Structura de date și clasele	Explicarea și detalierea metodelor folosite în implementarea proiectului	3, 4
Implementarea soluției	Prezentarea funcționalităților interfeței din perspectiva utilizatorului	4

## 2. Analiza problemei



Urmărind diagrama *use case* de mai sus, primul pas este introducerea de date de către utilizator, acestea fiind intervalele de timp de sosire între clienți și între serviri, numărul de cozi, numărul de clienți și durata de simulare. Aceste date sunt procesate începând cu apăsarea butonului **Start**, care începe cu validarea datelor și relațiile dintre acestea (cum ar fi asigurarea că valoarea minimă este mai mică sau egală cu valoarea maximă și că toate casetele sunt completate). În cazul în care datele introduse sunt invalide, se vor afișa mesaje corespunzătoare de eroare. Pentru a putea începe simularea, este necesar ca toate datele să fie valide.

Simularea începe atunci când datele sunt valide, iar evoluția clienților spre casele de marcat poate fi urmărită în timp real, în interfața grafică. Crearea clienților se face pe baza generării aleatoare a timpilor de apariție între aceștia și a timpului necesar pentru servirea fiecărui client și a intervalului de sosire a fiecărui client. Între timp, se ține cont și de timpul total de așteptare și de servire, pentru ca la final să putem genera niște statistici, cum ar fi minutul de vârf, numărul de clienți prezenți în cozi la minutul de vârf, timpul mediu de așteptare și timpul mediu de servire (timpii contorizați împărțiți la numărul de clienți total).



## 3. Proiectare

### 3.1 Introducere

Scopul principal al acestui proiect este de a dezvolta o aplicație de simulare care vizează analiza sistemelor bazate pe cozi pentru determinarea și minimizarea timpului de așteptare al clienților.

Am implementat acest sistem în Java și oferă o interfață grafică de utilizator intuitivă, care să permită utilizatorilor să introducă și să gestioneze datele de intrare astfel încât servirea să fie continuă și eficientă din punct de vedere al timpului, factori importanți când vine vorba de gestionat un număr mare de clienți. De asemenea, sistemul oferă o funcționalitate de afișare a rezultatelor într-un format ușor de înțeles și de interpretat.

## 3.2 Diagrama de clase

În diagrama de clase următoare, sunt prezentate schematic clasele care au realizat această aplicație de gestionare a clienților.

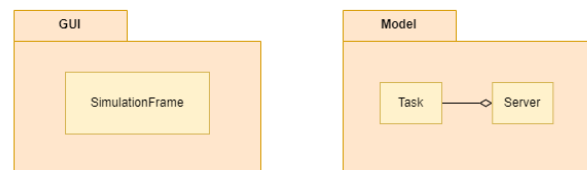
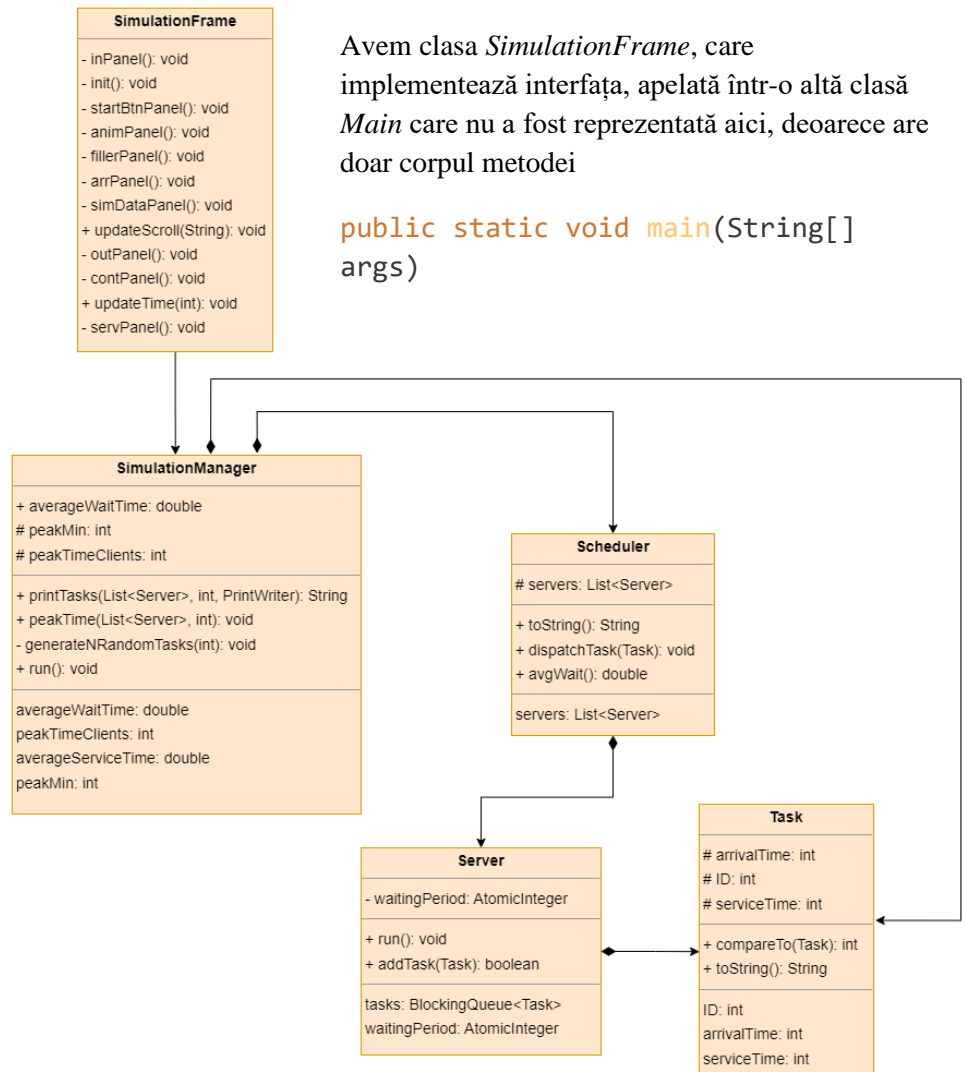
Clasa principală este *SimulationManager*, care implementează interfața *Runnable*. În constructorul său, sunt inițializați parametrii de simulare și o instanță a clasei *Scheduler* este creată pentru a gestiona cozile.

*Scheduler* are rolul de a stabili în ce coadă va merge noul *Task*, strategia implementată fiind ShortestTime, adică următorul client, la moment sosirii lui, va fi așezat la coada care are cel mai mic *waitingPeriod*.

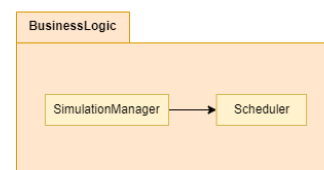
*Server* este clasa care reprezintă o coadă, iar aici are loc “servirea” propriu-zisă, în care Thread-ul doarme și decrementează *serviceTime*-ul clientului.

Avem clasa *SimulationFrame*, care implementează interfața, apelată într-o altă clasă *Main* care nu a fost reprezentată aici, deoarece are doar corpul metodei

```
public static void main(String[] args)
```



Repartizarea claselor în pachete





## 4. Implementarea

### 4.1 Descrierea claselor și metodelor

Datele sunt structurate în următoarele pachete și clase:

- **business.logic**, având clasele *Scheduler* și *SimulationManager*
- **gui**, având clasa *SimulationFrame*
- **model**, având clasele de bază *Server* și *Task*

#### Scheduler

Clasa este responsabilă de gestionarea *Task*-urilor noi și de a le repartiza în *Serverele* corespunzătoare. Am implementat strategia ShortestTime, adică următorul client, la moment sosirii lui, va fi așezat la coada care are cel mai mic *waitingPeriod*.

```
int minWaitingPeriod = Integer.MAX_VALUE;
int index = -1;
for (int i = 0; i < servers.size(); i++) {
    if (servers.get(i).getWaitingPeriod().get() < minWaitingPeriod) {
        minWaitingPeriod = servers.get(i).getWaitingPeriod().get();
        index = i;
    }
}
```

Mai apoi, *Task*-ul transmis ca parametru (*newTask*) va fi adăgat la *Server*-ul găsit:

```
servers.get(index).addTask(newTask);
```

#### SimulationManager

Clasa este responsabilă de gestionarea fiecărui *Task* – client care așteaptă să fie servit. Generează mai întâi o listă de *N Task*-uri aleatoare, *N* fiind numărul de clienți primit de la utilizator, și salvează acești clienți. Rulează cât timp o variabilă contor *currentTime* nu a ajuns încă la *simulationLimit*, o dată de intrare de la utilizator. Parcurgând astfel lista de clienți, care este sortată crescător în funcție de *arrivalTime* încă din metoda de generare, în momentul în care aceasta este egală cu *currentTime*, acest *Task* este preluat și procesat de către metoda dispatchTask din *Scheduler*.



De asemenea, aici se contorizează și timpul mediu de servire, și se adaugă *Task*-ul după procesare într-o listă separată. La finalul parcurgerii listei de clienți în așteptare la acest moment de timp, se va lua acea listă nou creată și se vor șterge aceste *Task*-uri din lista principală, acest lucru fiind făcut după for.

```
while (currentTime < simulationLimit) {
    List<Task> removed = new ArrayList<>(); // pun intr-o lista de toBeRemoved
    for (Task i : generatedTasks) { // in toate task-urile adica clientii waiting,
        if (i.getArrivalTime() == currentTime) { // daca le-a ajuns arrival time u,
            scheduler.dispatchTask(i); // stabileste in ce coada baga tasku 'i', si face
            asta cu shortestTime strategy
            averageServeTime += i.getServiceTime(); // aici incep sa adun service time u
            removed.add(i); // adaug tasku care trebe removed
        }
    }
    generatedTasks.removeAll(removed); // le sterg abia dupa ce am gatat de parcurs toate in
    timpul curent, gen la momentu currentTime
}
(...)
```

Tot în *SimulationManager* are loc scrierea în fișier, folosind un obiect de tip *PrintWriter*. În metoda printTasks, se construiește un StringBuilder pentru a fi mai apoi adăugat în interfața grafică, dar același conținut este în același timp scris și în *printWriter*, mai exact: Timpul curent în care se află execuția, lista clienților care așteaptă, precum și fiecare coadă și clienții care se află înăuntru, dar dacă e goală, se semnalează și acest lucru.

## Server

Această clasă are ca attribute următoarele:

```
private BlockingQueue<Task> clientsQueue;
private AtomicInteger waitingPeriod;
```

Avantajul folosirii unui obiect de tip

BlockingQueue este reprezentat în metoda addTask: metoda offer adaugă un element în coadă, dar caracterul de **Blocking** este dat de faptul că dacă s-a ajuns la capacitatea maximă (prestabilită de noi în constructorul *Server*-ului), aceasta nu mai adaugă, și este semnalat acest lucru.

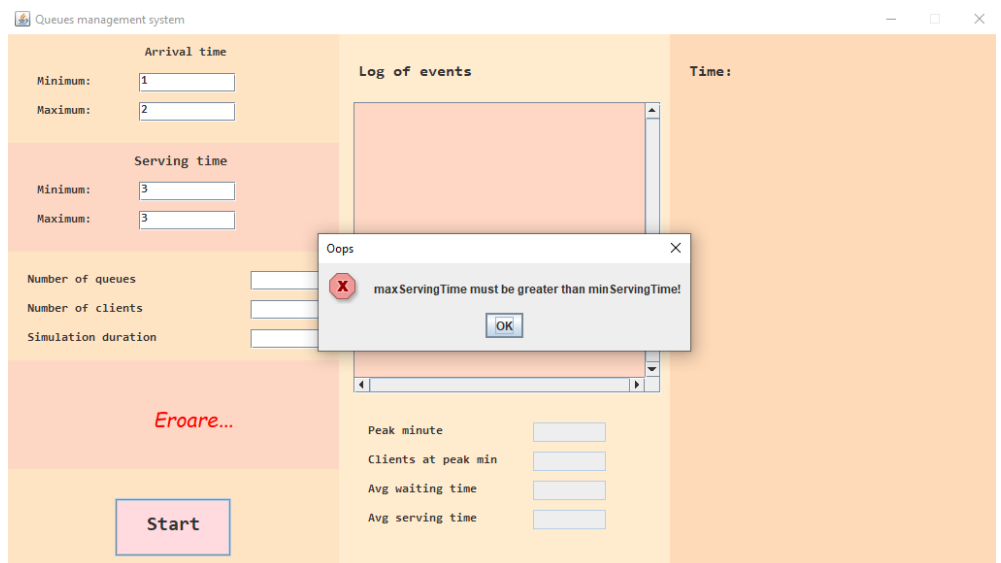
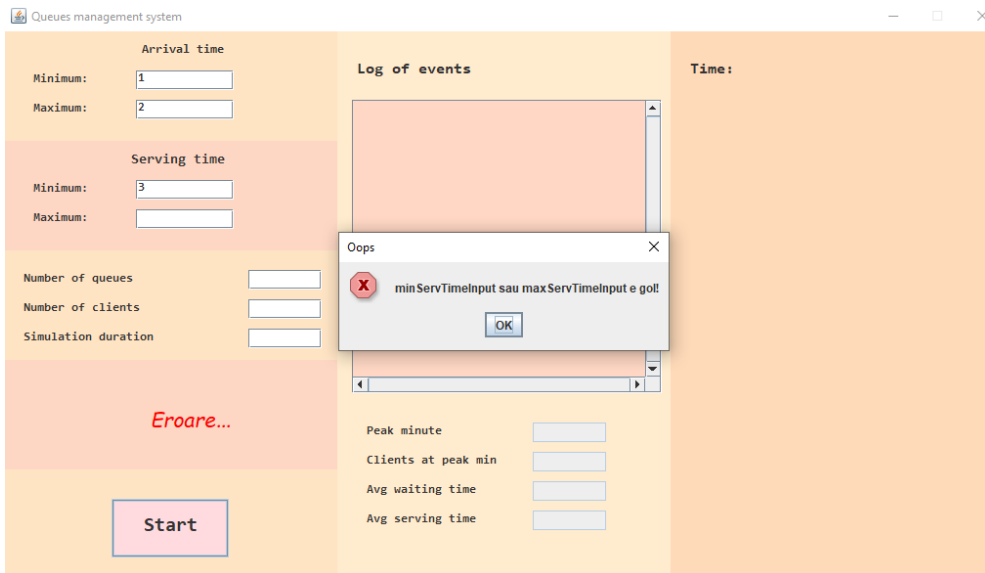
```
public boolean addTask(Task newTask) { // add task to queue + increment waiting period
    if (!clientsQueue.offer(newTask)) {
        System.out.println("Queue is full, " + newTask + " was not added");
        return false;
    }
    waitingPeriod.addAndGet(newTask.getServiceTime());
    return true;
}
```

În plus, în *Server* are loc servirea propriu-zisă a clientului. Fiecare coadă (*Server*) are propria sa coadă de clienți, pe care o parcurge. Ia pe rând primul client, îi scade timpul de servire, până ajunge la 0, apoi scade și waitingPeriod-ul cozii și scoate acest *Task* din coadă.

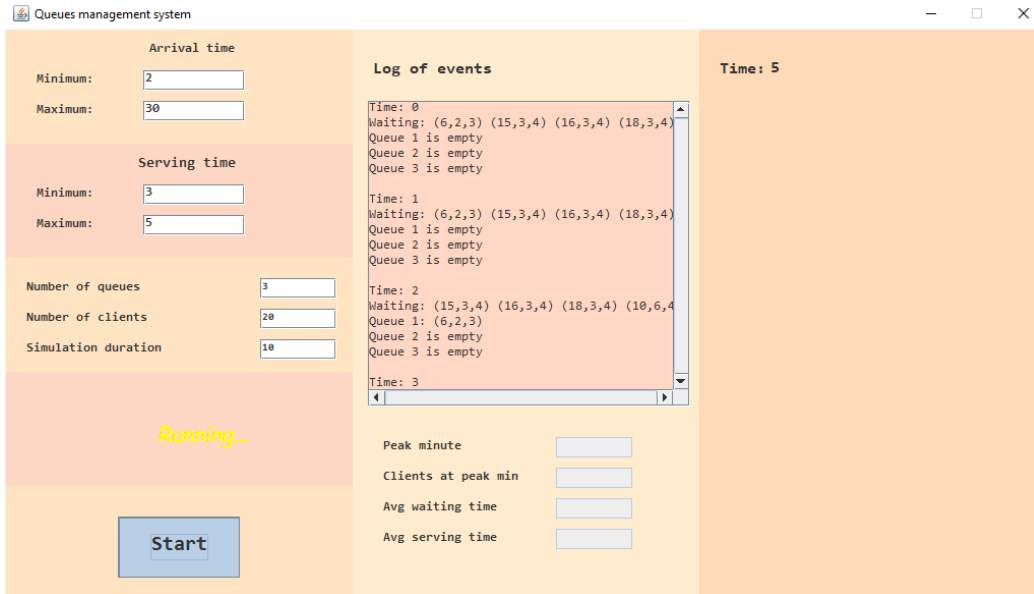


## 4.2 Interfața grafică

Utilizatorul trebuie să introducă anumite valori, fiecare având denumirea în stânga casetelor de input. La apăsarea butonului *Start*, are loc validarea datelor. Erorile posibile sunt semnalate, și un label roșu va fi setat în panel și un mesaj de eroare corespunzător.



Dacă datele sunt corecte, începe execuția programului. Label-ul semnalează ca rulează, timpul din dreapta se scurge în timp real (pentru a urmări mai ușor, acest timp are intervalul  $[1, \text{simDuration}]$ , dar în cod, după cum se poate observa în log și fișierul text, este  $[0, \text{simDuration} - 1]$ ), iar *Log of events* prezintă evoluția cozilor.



**Queues management system**

Arrival time  
Minimum: 2  
Maximum: 30

Serving time  
Minimum: 3  
Maximum: 5

Number of queues: 3  
Number of clients: 20  
Simulation duration: 10

**Running...**

**Start**

**Log of events**

Time: 0  
Waiting: (6,2,3) (15,3,4) (16,3,4) (18,3,4)  
Queue 1 is empty  
Queue 2 is empty  
Queue 3 is empty

Time: 1  
Waiting: (6,2,3) (15,3,4) (16,3,4) (18,3,4)  
Queue 1 is empty  
Queue 2 is empty  
Queue 3 is empty

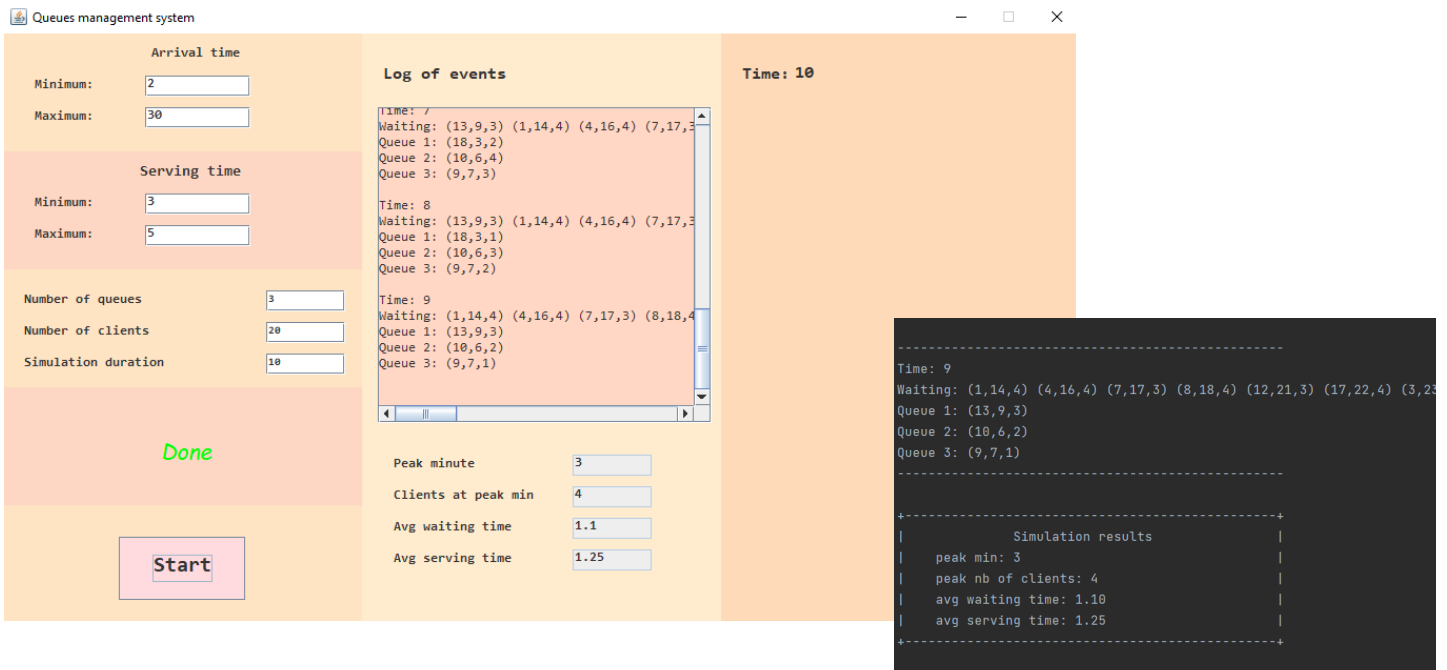
Time: 2  
Waiting: (15,3,4) (16,3,4) (18,3,4) (10,6,4)  
Queue 1: (6,2,3)  
Queue 2 is empty  
Queue 3 is empty

Time: 3

Time: 5

Peak minute  
Clients at peak min  
Avg waiting time  
Avg serving time

La final, se vor auto-completa casetele needitabile pentru statistici, Label-ul semnalează finalul execuției, iar scrierea în fișier este și ea gata.



**Queues management system**

Arrival time  
Minimum: 2  
Maximum: 30

Serving time  
Minimum: 3  
Maximum: 5

Number of queues: 3  
Number of clients: 20  
Simulation duration: 10

**Done**

**Start**

**Log of events**

Time: 7  
Waiting: (13,9,3) (1,14,4) (4,16,4) (7,17,3)  
Queue 1: (18,3,2)  
Queue 2: (10,6,4)  
Queue 3: (9,7,3)

Time: 8  
Waiting: (13,9,3) (1,14,4) (4,16,4) (7,17,3)  
Queue 1: (18,3,1)  
Queue 2: (10,6,3)  
Queue 3: (9,7,2)

Time: 9  
Waiting: (1,14,4) (4,16,4) (7,17,3) (8,18,4)  
Queue 1: (13,9,3)  
Queue 2: (10,6,2)  
Queue 3: (9,7,1)

Time: 10

Peak minute: 3  
Clients at peak min: 4  
Avg waiting time: 1.1  
Avg serving time: 1.25

```

-----
Time: 9
Waiting: (1,14,4) (4,16,4) (7,17,3) (8,18,4) (12,21,3) (17,22,4) (3,23,1)
Queue 1: (13,9,3)
Queue 2: (10,6,2)
Queue 3: (9,7,1)
-----
+-----+
|           Simulation results           |
| peak min: 3                           |
| peak nb of clients: 4                  |
| avg waiting time: 1.10                 |
| avg serving time: 1.25                 |
+-----+
  
```

## 5. Concluzii și dezvoltări ulterioare

În concluzie, am luat deciziile de implementare cele mai eficiente, în limita timpului disponibil. Am acumulat multe cunoștințe noi, și așa avea câteva dezvoltări ulterioare sau îmbunătățiri.

Una din ele ar fi ideea de animație a cozilor în panel-ul din dreapta, în care să avem o figură asemănătoare unui om pentru *Server* și *Task*, fiind mai intuitiv de urmărit clienții astfel, iar mișcarea lor să fie mai fluidă. De asemenea, pentru a asigura bunăstarea clienților noștri, ar fi ideal să evităm aglomerarea, astfel că ar am mai putea avea un buton de adăugare unei case noi, sau să se mai deschidă automat un anumit număr de case necesare fluidizării circulației clienților.

## 6. Bibliografie

- <https://dsrl.eu/courses/pt/>
- <https://introcs.cs.princeton.edu/java/3Ooop/>
- <https://www.javatpoint.com/java-string-format>
- <https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html>
- [http://www.tutorialspoint.com/java/util/timer\\_schedule\\_period.htm](http://www.tutorialspoint.com/java/util/timer_schedule_period.htm)
- <https://www.javacodegeeks.com/2013/01/java-thread-pool-example-using-executors-and-threadpoolexecutor.html>
- <https://stackoverflow.com>

