DOCUMENTATIE

TEMA *2*

NUME STUDENT: Pop Tudor Ștefan
GRUPA: 30226

CUPRINS

DOCUMENTATIE 1

TEMA 2 1

CUPRINS 1

1. Obiectivul temei 3

2. Analiza problemei, modelare, scenarii, cazuri de utilizare 3

3. Proiectare 4

4. Implementare 5

5. Rezultate 14

6. Concluzii 14

1. Obiectivul temei

Obiectivul principal al acestei teme este implementarea unui sistem automat de gestiune a cozilor de așteptare. Se presupune un număr de N clienți, și Q cozi. Clienții sunt determinați de către un triplet care conține un identificator unic, un timp de sosire la coadă, și durata de serviciu.

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

Problema: Gestionarea necorespunzătoare a cozilor duce la timpi mari de așteptare pentru fiecare client, și utilizare ineficientă a resurselor.

Soluția: Un sistem de gestionare a cozilor care implementează mecanisme eficiente de alocare a cozilor.

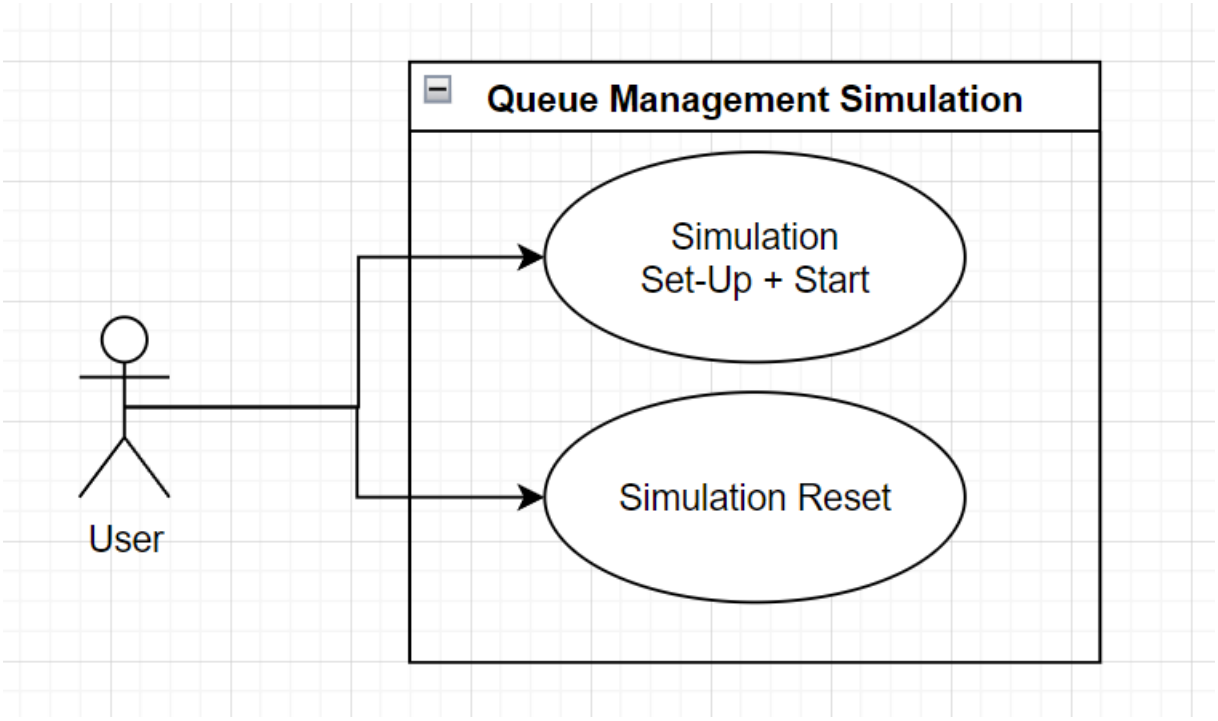

Figura 1. Cazurile de Utilizare.

Primul Caz de Utilizare: Atribuirea de valori + începerea simulării.
Actorul Principal: Utilizatorul
Scenariul de Succes: Utilizatorul introduce valori valide, i.e., numere întregi, în fiecare dintre căsuțele text ale interfeței grafice (număr de clienți, număr de cozi, timp maxim de simulare, timp minim și maxim de sosire, timp minim și maxim de serviciu, respectiv, alege strategia dorită de eficientizare a cozilor. După care, se apasă butonul de "Start Simulation", și programul va începe o simulare pe baza alegerilor făcute. Din secundă în secundă, timpul se va actualiza, și clienții vor fi procesați în funcție de strategia de gestionare aleasă, până la momentul închiderii "magazinului", sau până la procesarea tuturor clienților. Toate modificările vor fi afișate în timp real în jumătatea dreaptă a interfeței grafice.
Scenariul Alternativ: Utilizatorul introduce valori care nu sunt de tip întreg în una dintre căsuțele text din interfața grafică utilizator. În acest caz, programul semnalează o problemă, și afișează mesajul "All entries must be positive integers!" în jumătatea dreaptă a interfeței grafice.

Al Doilea Caz de Utilizare: Resetarea simulării.
Actorul Principal: Utilizatorul
Scenariul de Succes: Utilizatorul apasă pe butonul “Reset”, și interfața golește căsuțele text precum și jumătatea dreaptă a interfeței grafice.

Cerințele Funcționale: Se dorește ca aplicația de simulare să permită utilizatorului curent să poată oferi valori pentru rularea simulării, să înceapă simularea, să vizualizeze evoluția acesteia în timp real, iar, după finalizare, să se afișeze o statistică a zilei, respectiv, ora de vârf, numărul de cerințe îndeplinite (clienți în coadă) la ora de vârf, și timpul mediu de așteptare pentru fiecare client.
Cerințe Nonfuncționale: Se dorește ca aplicația de simulare să fie eficientă cu privire la timpul de execuție al programului în sine, și să furnizeze o interfață grafică primitoare și cu ușurință în utilizare și înțelegere pentru utilizator.

3. Proiectare

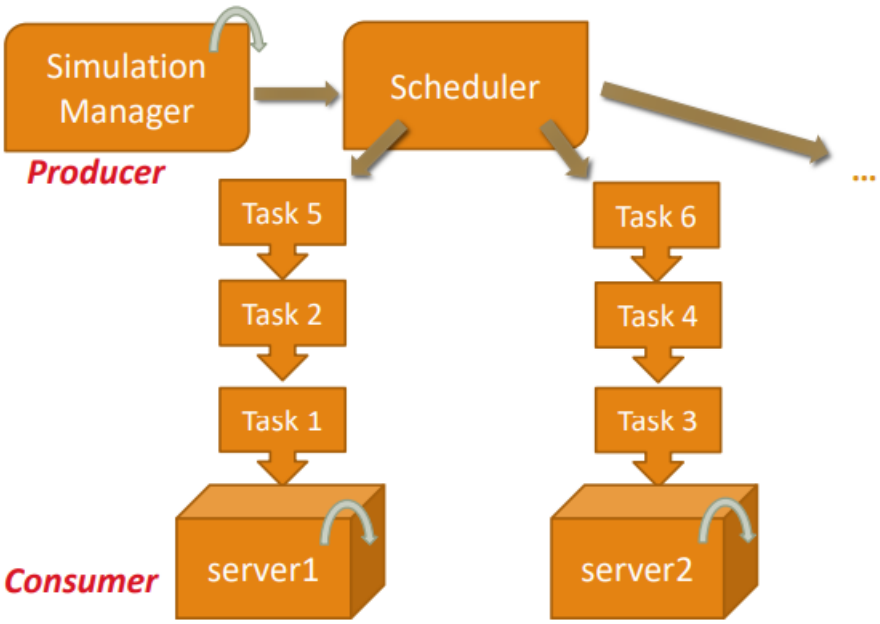

Figura 2. Arhitectura Conceptuală a Aplicației.

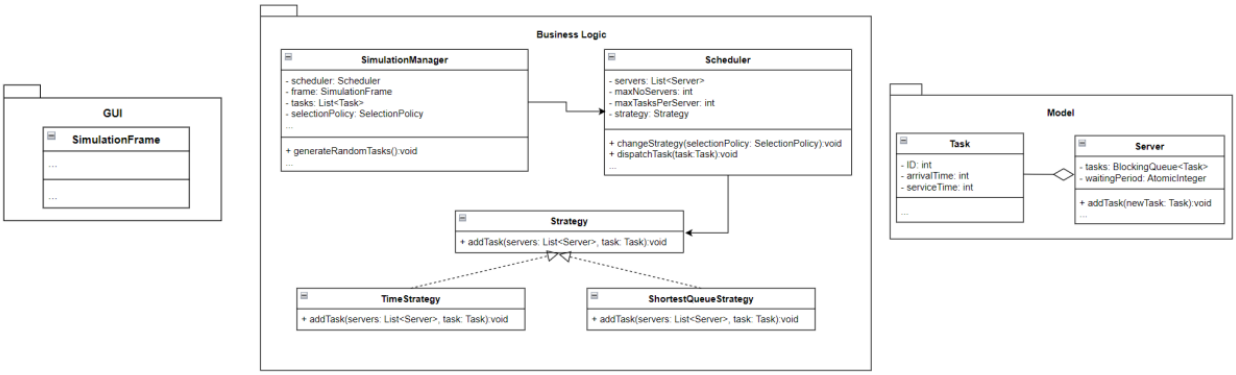

Figura 3. Diagrama de Pachete.

Figura 4. Diagrama UML de Clase a Aplicației.

4. Implementare

I. **SimulationFrame.java:** În cadrul pachetului "GUI", clasa SimulationFrame este responsabilă pentru generarea interfeței grafice utilizator pentru aplicația de simulare. În cadrul acestei clase se găsesc 8 obiecte de tip JLabel, ale căror conținuturi servesc ca fiind titluri pentru 7 obiecte de tip JTextField, și a unui obiect de tip JComboBox, unde se vor insera datele de intrare necesare simulării. Se mai pot găsi două butoane, "Start Simulation", și "Reset", ale căror funcționalitate se explică de la sine. În jumătatea dreaptă a interfeței grafice se găsește un obiect de tip JTextArea, în concordanță cu un obiect de tip JScrollPane. Acestea sunt responsabile pentru afișarea în timp real a fiecărui pas al simulării.

II. **Strategy.java:** În cadrul pachetului "logic", interfața Strategy este responsabilă pentru metodele (ulterior moștenite în TimeStrategy și ShortestQueueStrategy) care vor face inserările și ștergerile fiecărui client în fiecare coadă, după strategia aleasă. Metodele ce vor fi moștenite sunt "addTask" și "getBest".

```
void addTask(List<Server> servers, Task task);
 int getBest(List<Server> servers);
```

III. **TimeStrategy.java**: Din pachetul “logic”, aceasta este una dintre cele două implementări a interfeței Strategy, care decide coada în care următorul client va fi plasat, în funcție de coada cu cel mai scurt timp acumulat de serviciu pentru fiecare client. Metoda “getBest” iterează printre toate cozile de la momentul curent, calculează toți timpii de serviciu, și decide care dintre cozi minimizează acești timpi, după care, metoda “addTask” adaugă următorul client în coada decisă de către “getBest”.

```
public void addTask(List<Server> servers, Task task) {
    int index = getBest(servers);
    servers.get(index).newTask(task);
}
public int getBest(List<Server> servers) {
    int minPeriod = Integer.MAX_VALUE;
    int index = 0;
    for (int i = 0; i < servers.size(); i++) {
        if (!servers.get(i).isClosingTime() && servers.get(i).getWaitingPeriod().get() <
minPeriod) {
            minPeriod = servers.get(i).getWaitingPeriod().get();
            index = i;
        }
    }
    return index;
}
```

IV. **ShortestQueueStrategy.java:** Din pachetul “logic”, aceasta este cealaltă dintre cele două implementări a interfeței Strategy, care plasează următorul client în coada cu număr minim de clienți. În acest caz, “getBest” numără elementele fiecărei cozi, salvează numărul minim, după care, “addTask” adaugă următorul client în coada decisă de către metoda “getBest”.

```
public void addTask(List<Server> servers, Task task) {
    int index = getBest(servers);
    servers.get(index).newTask(task);
}
public int getBest(List<Server> servers) {
    int index = 0;
    Server bestServer = servers.getFirst();
    for (int i = 1; i < servers.size(); i++) {
        if (!servers.get(i).isClosingTime() && servers.get(i).getTasks().size() <
bestServer.getTasks().size()) {
            bestServer = servers.get(i);
            index = i;
        }
    }
    return index;
}
```

V. **Scheduler.java:** Din pachetul “logic”, clasa Scheduler este responsabilă pentru crearea și inițializarea fiecărei cozi, precum și de deciderea strategiei după care se va proceda în cadrul simulării. Constructorul acestei clase este funcția unde se inițializează cozile,

numite “servere”, și se pornește fiecare server ca fiind propria sa entitate, executându-se în paralel cu celelalte servere. În alte cuvinte, fiecare server este un Thread diferit. “getCurrentServiceTime” returnează timpul de serviciu total la momentul curent, adică, suma tuturor timpilor de serviciu a fiecărui server. “howManyTasks” returnează numărul actual de taskuri care au fost expediate, prin însumarea numărului de clienți din fiecare coadă. “changeStrategy” modifică strategia de gestionare a cozilor în funcție de variabila “selectionPolicy”. “dispatchTask” adaugă câte un task nou unui server nou, în funcție de factorii discutați anterior. “clockOutEarly” și “clockOut” sunt funcțiile care semnalează încetarea funcționalității serverelor. “clockOutEarly” se activează dacă toți clienții în așteptare sunt serviți înainte de ora închiderii (timpul maxim de simulare), iar, “clockOut”, în caz contrar (la ora închiderii).

```
public Scheduler(int maxNoServers, int maxTasksPerServer) {
    open = true;
    this.maxNoServers = maxNoServers;
    servers = new ArrayList<>(maxNoServers);
    for (int i = 0; i < maxNoServers; i++) {
        Server server = new Server(maxTasksPerServer);
        servers.add(server);
        Thread th = new Thread(server);
        th.start();
    }
}
public int getCurrentServiceTime() {
    return currentServiceTime;
}
public int howManyTasks() {
    int count = 0;
    for (Server server : servers) {
        count += server.getTasks().size();
    }
    return count;
}
public void changeStrategy(SelectionPolicy selectionPolicy) {
    if (selectionPolicy == SelectionPolicy.TIMESTRATEGY)
        strategy = new TimeStrategy();
    else if (selectionPolicy == SelectionPolicy.SHORTESTQUEUESTRATEGY)
        strategy = new ShortestQueueStrategy();
}
public List<Server> getServers() {
    return servers;
}

public void dispatchTask(Task task) {
    strategy.addTask(servers, task);
    currentServiceTime += task.getServiceTime();
}

public void clockOutEarly() {
    for (Server server : servers)
```

```
        server.setEarlyClosingTime();
    open = false;
}

public void clockOut() {
    for (Server server : servers)
        server.setClosingTime();
    open = false;
}

public boolean serversAreWorking() {
    for (Server s : servers) {
        if(!s.getTasks().isEmpty())
            return true;
    }
    return false;
}
```

VI. **SelectionPolicy.java:** În cadrul pachetului "logic", SelectionPolicy este o enumerație formată din elementele "TIMESTRATEGY", și "SHORTESTQUEUESTRATEGY".

```
public enum SelectionPolicy {
    TIMESTRATEGY, SHORTESTQUEUESTRATEGY
}
```

VII. **Task.java:** În cadrul pachetului "model", Task este clasa care descrie structura clienților ce vor fi expediați către cozile (serverele) corespondente. Această clasă implementează interfața "Comparable<Task>" cu scopul de a sorta crescător clienții în raport cu timpii de sosire ai acestora.

```
private int ID, arrivalTime;
private AtomicInteger serviceTime;
public Task(int ID, int arrivalTime, int serviceTime) {
    this.ID = ID;
    this.arrivalTime = arrivalTime;
    this.serviceTime = new AtomicInteger(serviceTime);
}

public void decrementServiceTime(){
    serviceTime.decrementAndGet();
}

public int getArrivalTime() {
    return arrivalTime;
}
public int getServiceTime() {
    return serviceTime.get();
}
@Override
```

```
public int compareTo(Task task) {
    return Integer.compare(this.arrivalTime, task.arrivalTime);
}
@Override
public String toString() {
    return "(" + ID + ", " + arrivalTime + ", " + serviceTime.get() + ")";
}
```

VIII. **Server.java:** Din cadrul pachetului "model", clasa Server este responsabilă pentru implementarea în cod a cozilor. Anterior a fost menționat faptul că serverele sunt declarate ca și Thread-uri. Prin urmare, clasa Server implementează interfața "Runnable", și, implicit, moștenește metoda "run", care reprezintă criptarea comportamentală a serverelor în sine. Se observă prezența a două atribute, respectiv, "currentTime", și "selfCurrentTime". "currentTime" este o variabilă statică, care criptează timpul curent al simulării (în secunde), iar "selfCurrentTime" este o variabilă care se actualizează doar în concordanță cu "currentTime" la anumite momente de timp bine determint

```
private final BlockingQueue<Task> tasks;
private AtomicInteger waitingPeriod;
private boolean closingTime = false, earlyClosingTime = false;
public static final AtomicInteger currentTime = new AtomicInteger(0);
private int selfCurrentTime = -1;
public Server(int maxTasksPerServer) {
    tasks = new LinkedBlockingQueue<>(maxTasksPerServer);
    waitingPeriod = new AtomicInteger(0);
    selfCurrentTime = currentTime.get();
}

public void setEarlyClosingTime() {
    this.earlyClosingTime = true;
}

public boolean isClosingTime() {
    return closingTime;
}
public static synchronized void resetSimulation() { currentTime.set(0); }
public static synchronized int whatTimeIsIt() { return currentTime.get(); }
public static synchronized void newTime() { currentTime.incrementAndGet(); }
public void setClosingTime() {
    this.closingTime = true;
}

public BlockingQueue<Task> getTasks() {
    return tasks;
}

public AtomicInteger getWaitingPeriod() {
    return waitingPeriod;
}
```

```
public void newTask(Task task) {
    synchronized (tasks) {
        tasks.add(task);
        waitingPeriod.addAndGet(task.getServiceTime());
        if (selfCurrentTime < 0)
            selfCurrentTime = task.getArrivalTime();
    }
}

@Override
public void run() { // got my eye on you
    while (!closingTime && (!earlyClosingTime || !tasks.isEmpty())) {
        if(selfCurrentTime >= currentTime.get()){
            Thread.yield();
            continue;
        }
        selfCurrentTime = currentTime.get();
        if (tasks.isEmpty()) {
            Thread.yield();
            continue;
        }
        if (tasks.peek() != null && tasks.peek().getServiceTime() >= 2)
            tasks.peek().decrementServiceTime();
        else tasks.remove();
        selfCurrentTime = currentTime.get();
        waitingPeriod.decrementAndGet();
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
}

@Override
public String toString() {
    if (tasks.isEmpty())
        return "closed";
    StringBuilder builder = new StringBuilder();
    for (Task task : tasks)
        builder.append(task).append(' ');
    return builder.toString();
}
```

IX. **SimulationManager.java:** Ultima clasă din cadrul acestui proiect. Aceasta se află în pachetul “logic”, și este responsabilă cu rularea simulării în sine.

```
private int maxSimulationTime, minArrivalTime, maxArrivalTime, minServiceTime,
maxServiceTime, nrServers, nrClients;
    private Scheduler scheduler;
    private Thread mainThread;
```

```
    private SelectionPolicy selectionPolicy;
    private List<Task> tasks;
    private FileWriter file;

    public SimulationManager(int N, int Q, int maxSimulationTime, int minArrivalTime, int
maxArrivalTime, int minServiceTime, int maxServiceTime, SelectionPolicy policy) {
        nrClients = N;
        nrServers = Q;
        this.maxSimulationTime = maxSimulationTime;
        this.minArrivalTime = minArrivalTime;
        this.maxArrivalTime = maxArrivalTime;
        this.minServiceTime = minServiceTime;
        this.maxServiceTime = maxServiceTime;
        selectionPolicy = policy;
    }
    public void simulationSetUp() {
        try {
            file = new FileWriter("log.txt");
        }
        catch (IOException e) {
            System.out.println("Upon initializing the File Writer, the program
encountered an IO Exception!");
        }
        scheduler = new Scheduler(nrServers, nrClients);
        scheduler.changeStrategy(selectionPolicy);
        generateRandomTasks();
        mainThread = new Thread(this);
        Server.resetSimulation();
    }

    public void startSim(){
//        System.out.println("simulation started");
        mainThread.start();
    }

    public void generateRandomTasks() {
        Random r = new Random();
        tasks = new ArrayList<>(nrClients);
        for (int i = 0; i < nrClients; i++) {
            int arrTime = r.nextInt(minArrivalTime, maxArrivalTime);
            int servTime = r.nextInt(minServiceTime, maxServiceTime);
            tasks.add(new Task(i + 1, arrTime, servTime));
        }
        Collections.sort(tasks);
    }

    public void log(String s) {
        if (file == null) return;
        try {
```

```
            file.write(s + "\n");
        }
        catch (IOException e) {
            System.out.println("Failed to append to log.txt!");
        }
    }
    @Override
    public void run() {
        double avgWait = 0.0, avgService = 0.0;
        int peakHour = 0, peakTasks = 0;
        while (Server.whatTimeIsIt() < maxSimulationTime && (!tasks.isEmpty() ||
scheduler.serversAreWorking())) {
            while (!tasks.isEmpty() && tasks.getFirst().getArrivalTime() <=
Server.whatTimeIsIt()) {
                scheduler.dispatchTask(tasks.removeFirst());
            }

            log(toString());

            try {
                file.write("Time " + Server.whatTimeIsIt() + "\n");
                file.write("Waiting clients: " + tasks + "\n");
                file.write(scheduler.toString() + "\n");
            }
            catch (IOException e) {
                System.out.println("Failed to append to log.txt!");
            }
            if (scheduler.howManyTasks() > peakTasks) {
                peakHour = Server.whatTimeIsIt();
                peakTasks = scheduler.howManyTasks();
            }

            for (Server server : scheduler.getServers()) {
                avgWait += server.getWaitingPeriod().get();
            }
            //Server.currentTime.incrementAndGet();
            if (tasks.isEmpty())
                scheduler.clockOutEarly();
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            Server.newTime();
        }
        if (scheduler.serversAreWorking()) {
            log("We're Closed!");
            scheduler.clockOut();
        }
        avgWait /= nrClients;
```

```
        log("Average Waiting Time: " + avgWait);
        avgService = (double)scheduler.getCurrentServiceTime() / nrClients;
        log("Average Service Time: " + avgService);
        log("Peak Hour: " + peakHour);
        log("Number of processed tasks during the peak hour: " + peakTasks);
        try {
            file.close();
        }
        catch (IOException e) {
            System.out.println("An issue occurred while attempting to close log.txt!");
        }
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append("Time ").append(Server.whatTimeIsIt()).append('\n');
        synchronized (tasks){
            sb.append("Waiting clients: ").append(tasks).append('\n');
        }
        sb.append(scheduler);

        return sb.toString();
    }

    public void setMaxSimulationTime(int maxSimulationTime) {
        this.maxSimulationTime = maxSimulationTime;
    }

    public void setMinArrivalTime(int minArrivalTime) {
        this.minArrivalTime = minArrivalTime;
    }

    public void setMaxArrivalTime(int maxArrivalTime) {
        this.maxArrivalTime = maxArrivalTime;
    }

    public void setMinServiceTime(int minServiceTime) {
        this.minServiceTime = minServiceTime;
    }

    public void setMaxServiceTime(int maxServiceTime) {
        this.maxServiceTime = maxServiceTime;
    }

    public void setNrServers(int nrServers) {
        this.nrServers = nrServers;
    }
```

```
    public void setNrClients(int nrClients) {
        this.nrClients = nrClients;
    }

    public void setSelectionPolicy(SelectionPolicy selectionPolicy) {
        this.selectionPolicy = selectionPolicy;
    }

    public static void main(String[] args) {
//        SimulationManager manager = new SimulationManager(10, 4, 30, 2, 4, 1, 5,
SelectionPolicy.TIMESTRATEGY);
        new SimulationFrame();

    }
```

5. Rezultate

Se consideră trei teste diferite, cu valorile specificate în cadrul cerinței problemei, respectiv:

Test 1	Test 2	Test 3
N = 4 Q = 2 $t_{simulation}^{MAX}$ = 60 seconds $[t_{arrival}^{MIN}, t_{arrival}^{MAX}] = [2, 30]$ $[t_{service}^{MIN}, t_{service}^{MAX}] = [2, 4]$	N = 50 Q = 5 $t_{simulation}^{MAX}$ = 60 seconds $[t_{arrival}^{MIN}, t_{arrival}^{MAX}] = [2, 40]$ $[t_{service}^{MIN}, t_{service}^{MAX}] = [1, 7]$	N = 1000 Q = 20 $t_{simulation}^{MAX}$ = 200 seconds $[t_{arrival}^{MIN}, t_{arrival}^{MAX}] = [10, 100]$ $[t_{service}^{MIN}, t_{service}^{MAX}] = [3, 9]$

Figura 5. Valorile pentru testarea programului.

Aici, N reprezintă numărul de clienți (task-uri), Q reprezintă numărul de servere, tMaxSimulation reprezintă timpul maxim de simulare, care, atunci când este depășit, se închide aplicația pe baza funcției “clockOut”, tMinArrival, și tMaxArrival reprezintă marginile de sosire între care se pot încadra clienții, iar tMinService, tMaxService reprezintă marginile de timpi de serviciu între care se pot încadra clienții. Rezultatele acestor teste se vor salva în trei fișiere text log, respectiv: simularea 1 se afișează în fișierul “log1.txt”, simularea 2 se afișează în “log2.txt”, și simularea 3 în “log3.txt”. Toate aceste trei fișiere se găsesc în repertoriul acestui proiect pe GitLab.

6. Concluzii

În concluzie, aplicația de proiectare și implementare a acestui sistem de gestiune a cozilor s-a dovedit a fi un proiect de practică foarte bun pentru familiarizarea cu utilizarea de Thread-uri în Java.