

DOCUMENTATIE

TEMA 2

**GHICA MADALINA
GRUPA 30229**

CUPRINS

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

1. Obiectivul temei

Această sarcină implică proiectarea și implementarea unei aplicații de gestionare a cozilor, care atribuie clienților cozi astfel încât timpul de așteptare să fie minimizat. Această aplicație va simula un număr N de clienți care sosesc pentru serviciu, intrând în Q cozi, așteptând, fiind serviți și, în cele din urmă, părăsind coada. Scopul este de a minimiza timpul total petrecut de fiecare client în cozi și de a calcula timpul mediu de așteptare.

Fiecare client este adăugat la coada cu timpul minim de așteptare atunci când timpul său de sosire este mai mare sau egal cu timpul de simulare. Scopul este de a minimiza timpul total de așteptare și de a găsi cea mai bună configurație de cozi.

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

Se dorește crearea unei aplicații care să permită analizarea sistemelor bazate pe cozi de așteptare. Aceasta va avea o interfață grafică intuitivă pentru a introduce datele necesare pentru setarea simulării și selectarea metodei de simulare. Scopul final al aplicației este de a analiza două abordări diferite de management al cozilor prin afișarea în timp real a performanței sistemelor.

Cerințe funcționale:

Aplicația de simulare ar trebui să permită utilizatorilor să configureze simularea.

Aplicația de simulare ar trebui să permită utilizatorilor să înceapă simularea.

Aplicația de simulare ar trebui să afișeze evoluția cozilor în timp real, iar la final să indice timpul mediu de așteptare, ora de vârf și timpul mediu de servire.

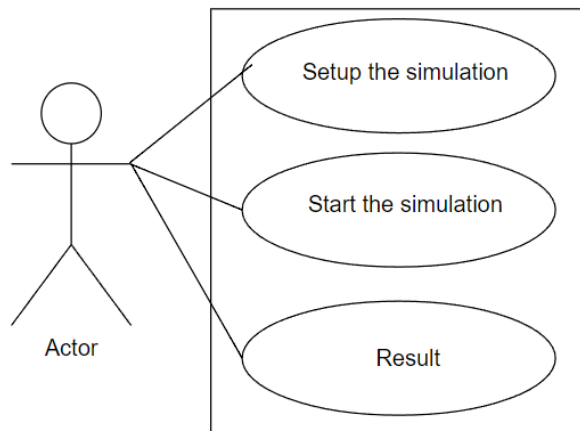
Cerințe non-funcționale:

Calculatorul polinomial trebuie să fie rapid și fiabil.

Aplicația de simulare ar trebui să fie intuitivă și ușor de utilizat.

Cazuri de utilizare:

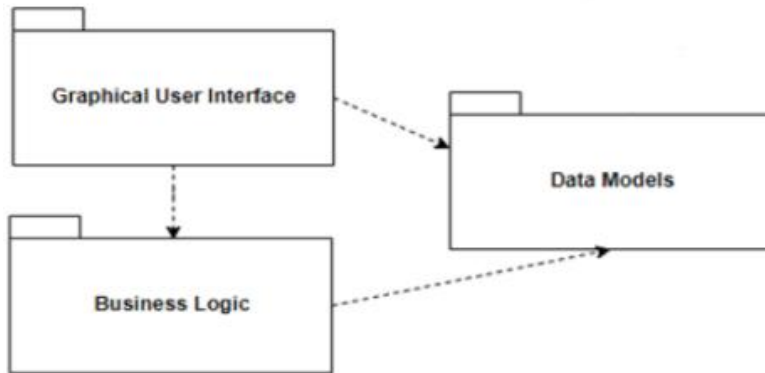
- ➔ Se porneste simularea
- ➔ Se introduc datele
- ➔ Se afiseaza rezultatul final



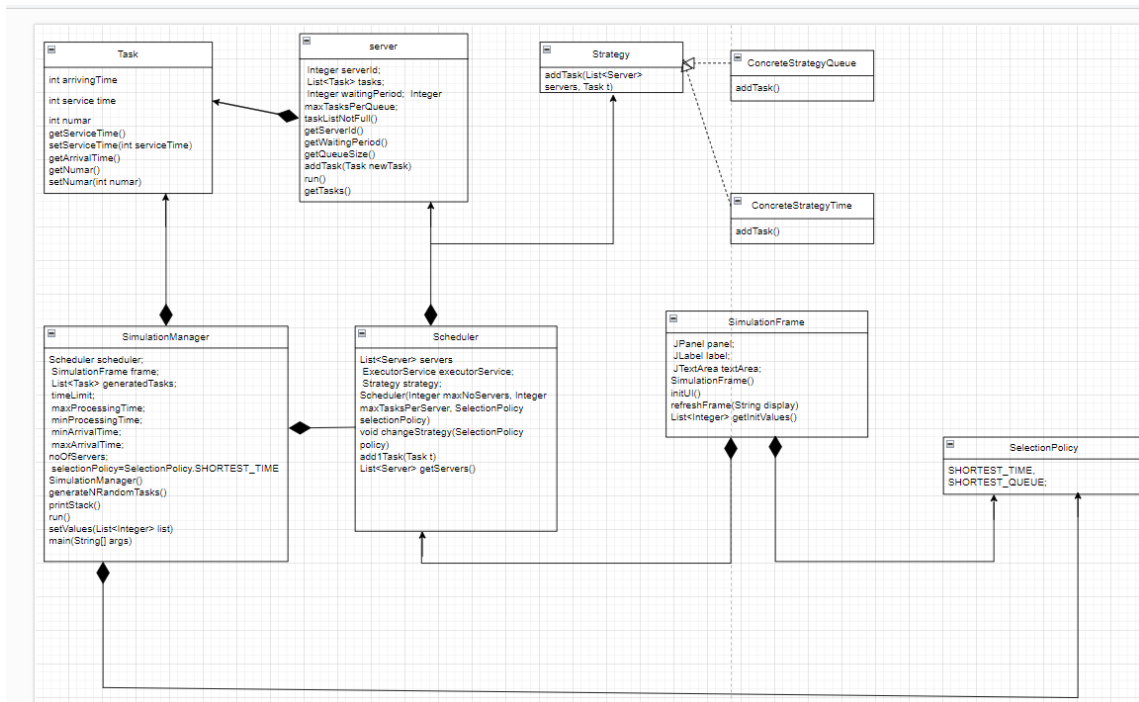
3. Proiectare

Urmatoarele clase fac parte din implementarea unui sistem de simulare a unei cozi de servire, unde un număr de clienți ajung într-un anumit moment la coadă și trebuie să fie serviti de unul sau mai mulți servere într-un anumit mod:

- Clasa `Server` reprezintă un server care poate procesa sarcini (obiecte de tipul `Task`) și are o anumită coadă de așteptare a sarcinilor.
- Clasa `Task` reprezintă o sarcină care trebuie procesată de unul dintre servere.
- Clasa `SimulationFrame` reprezintă o fereastră grafică care afișează starea curentă a simulării, precum și diverse opțiuni și comenzi pentru a controla simularea.
- Interfața `Strategy` reprezintă o strategie de atribuire a sarcinilor unui server.
- Clasa `TimeStrategy` și `ShortestQueueStrategy` sunt două strategii concrete de atribuire a sarcinilor, implementând interfața `Strategy`.
- Clasa `SimulationManager` coordonează simularea și implementează logica generală a sistemului, inclusiv generarea sarcinilor și actualizarea serverelor și sarcinilor în timp real.
- Enum-ul `SelectionPolicy` specifică politica de selecție a serverelor în cadrul strategiilor.
- Clasa `Scheduler` reprezintă scheduler-ul care administrează toate serverele și sarcinile și care se ocupă de a le atribui în funcție de strategia specificată.
- Clasa `ConcreteStrategyTime` și `ConcreteStrategyQueue` reprezintă clasele care implementează strategiile concrete `TimeStrategy` și `ShortestQueueStrategy`.



Diagrame UML



4. Implementare

Clasa Server este definita in pachetul „Model” si implementeaza interfata Runnable.

```

public class Server implements Runnable {
    2 usages
    private Integer serverId;
    9 usages
    private List<Task> tasks;
    4 usages
    private Integer waitingPeriod;
    3 usages
    private Integer maxTasksPerQueue;

    1 usage new *
    public Server(Integer maxTasksPerQueue, Integer serverId) {
        this.maxTasksPerQueue = maxTasksPerQueue;
        this.serverId = serverId;
        this.tasks = new ArrayList<>();
        this.waitingPeriod = 0;
    }

    3 usages new *
    public Boolean taskListNotFull() {
        return tasks.size() < maxTasksPerQueue;
    }

    1 usage new *
    public Integer getServerId() {
        return serverId;
    }

    tasks.remove(index: 0);
    }
    }
    }

    1 usage new *
    public Task[] getTasks() {
        return tasks.toArray(new Task[0]);
    }
}

    public Integer getWaitingPeriod() {
        return waitingPeriod;
    }

    5 usages new *
    public Integer getQueueSize() {
        return tasks.size();
    }

    2 usages new *
    public void addTask(Task newTask) {
        if (tasks.size() < maxTasksPerQueue) {
            tasks.add(newTask);
            waitingPeriod += newTask.getServiceTime();
        }
    }

    new *
    public void run() {
        while (true) {
            try {
                Thread.sleep( millis: 1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            if (tasks.size() > 0) {
                Task currentTask = tasks.get(0);
                currentTask.setServiceTime(currentTask.getServiceTime() - 1);
                waitingPeriod--;
                if (currentTask.getServiceTime() == 0) {
                    tasks.remove(index: 0);
                }
            }
        }
    }
}

```

Clasa conține o listă de sarcini care vor fi procesate de către server, un număr maxim de sarcini permise în coadă, un identificator unic al serverului și o perioadă totală de așteptare a sarcinilor din coadă.

Clasa implementează metode pentru adăugarea unei noi sarcini în coadă, verificarea dacă coada este plină, returnarea identificatorului unic al serverului, a perioadei de așteptare totală a sarcinilor din coadă și a sarcinilor din coadă.

Clasa conține de asemenea o metodă run() care procesează sarcinile din coadă și actualizează starea lor la fiecare secundă.

Clasa task este definită în pachetul „Model” .

```


public class Task {
    2 usages
    private int arrivingTime;
    3 usages
    private int serviceTime;
    2 usages
    private int numar;
    1 usage new *
    public Task(int arrivingTime, int serviceTime) {
        this.arrivingTime = arrivingTime;
        this.serviceTime = serviceTime;
    }
    5 usages new *
    public int getServiceTime() { return serviceTime; }
    1 usage new *
    public void setServiceTime(int serviceTime) {
        this.serviceTime = serviceTime;
    }
    5 usages new *
    public int getArrivalTime() { return arrivingTime; }
    2 usages new *
    public int getNumar() { return numar; }
    1 usage new *
    public void setNumar(int numar) { this.numar=numar; }
}

```

Clasa conține trei variabile membre private: `arrivingTime` (timpul de sosire a sarcinii în coadă), `serviceTime` (timpul necesar pentru a procesa sarcina) și `numar` (un număr unic pentru sarcină). Constructorul clasei acceptă ca argumente timpul de sosire și timpul de procesare al sarcinii. Clasa oferă și o serie de metode publice pentru a accesa și modifica variabilele membre, cum ar fi `getServiceTime()`, `getArrivalTime()`, `setNumar()`, etc. Această clasă este utilizată în contextul unei implementări a unui sistem de cozi pentru a gestiona sarcinile care sosesc în sistem și timpul necesar pentru a le procesa.

Enumeratia `SelectionPolicy` este definita in pachetul „BusinessLogic” .

```

public enum SelectionPolicy {
    2 usages
     SHORTEST_TIME,
    1 usage
    SHORTEST_QUEUE;
}

```

Interfata `Strategy` este definita in pachetul „BusinessLogic”

```

3 pages 2 implementations new *
public interface Strategy {
    1 usage 2 implementations new *
    public void addTask(List<Server> servers, Task t);
}

```

ConcreteStrategyQueue este definită în pachetul „BusinessLogic” și implementează interfața Strategy.

```

1 usage new *
public class ConcreteStrategyQueue implements Strategy{
    1 usage new *
    public void addTask (List<Server> servers, Task t)
    {
        int min = 0;
        int id = -1;
        for(int i=0;i<servers.size();i++){
            if (min==0) {
                if (servers.get(i).taskListNotFull()) {
                    min = servers.get(i).getQueueSize();
                    id = i;
                }
            } else if(min>servers.get(i).getQueueSize())
            {
                min=servers.get(i).getQueueSize();
                id=i;
            }
        }
        servers.get(id).addTask(t);
    }
}

```

Această clasă definește o strategie de adăugare a sarcinilor în cozi într-un sistem multi-server. Metoda addTask() primește două argumente: o listă de servere și o sarcină (Task) care trebuie adăugată într-o coadă.

Metoda parcurge lista de servere și găsește serverul cu cea mai mică coadă de sarcini. Dacă există mai multe servere cu aceeași dimensiune a cozi, se va adăuga sarcina la primul server găsit cu această dimensiune de coadă.

După găsirea serverului potrivit, metoda adaugă sarcina în coada acestuia utilizând metoda addTask() a obiectului Server corespunzător din listă.

ConcreteStrategyTime este definită în pachetul „BusinessLogic” și implementează interfața Strategy.


```

public class ConcreteStrategyTime implements Strategy {
    1 usage new *
    public void addTask (List<Server> servers, Task t)
    {
        int min = 0;
        int id = 0;
        for(int i=0;i<servers.size();i++) {
            if (min==0) {
                if (servers.get(i).taskListNotFull()) {
                    min = servers.get(i).getWaitingPeriod();
                    id = i;
                }
            } else if(servers.get(i).taskListNotFull() && min > servers.get(i).getWaitingPeriod())
            {
                min=servers.get(i).getWaitingPeriod();
                id=i;
            }
        }
        servers.get(id).addTask(t);
    }
}

```

Această clasă conține o metodă addTask(), care adaugă o sarcină într-unul dintre serverele din lista dată ca parametru, folosind o strategie bazată pe timpul de așteptare în coadă. Metoda addTask() parcurge toate serverele din lista dată și determină serverul cu cel mai mic timp de așteptare total în coadă.

Dacă serverul găsit are încă spațiu disponibil în coadă, sarcina este adăugată la coada serverului respectiv. În caz contrar, metoda continuă să parcurgă serverele și să aleagă serverul cu cel mai mic timp de așteptare total în coadă și spațiu disponibil în coadă.

Această metodă implementează o strategie de distribuire a sarcinilor care vizează minimizarea timpului total de așteptare al sarcinilor în cozi.

Clasa Scheduler este definită în pachetul „BusinessLogic”.

Clasa are un constructor care primește numărul maxim de servere, numărul maxim de sarcini pe server și politica de selecție a serverelor (scurtătură pentru "politica de selecție" este "selection policy").

Constructorul creează un ExecutorService cu un număr fix de thread-uri egal cu numărul maxim de servere și initializează un ArrayList cu obiecte Server, fiecare având un identificator unic.

De asemenea, porneste fiecare server în propriul thread, pentru a procesa sarcinile din coada sa. Clasa Scheduler oferă metode pentru a schimba politica de selecție a serverelor, a atribui o sarcină unui server și pentru a obține lista de servere. În plus, clasa oferă și o metodă shutdown() care închide ExecutorService, determinând toate thread-urile să se oprească.

```

public class Scheduler {
    4 usages
    private List<Server> servers = new ArrayList<>();
    3 usages
    private ExecutorService executorService;
    3 usages
    private Strategy strategy;

    1 usage new *
    public Scheduler(Integer maxNoServers, Integer maxTasksPerServer, SelectionPolicy selectionPolicy) {
        executorService = Executors.newFixedThreadPool(maxNoServers);
        for (Integer i = 0; i < maxNoServers; i++) {
            servers.add(new Server(maxTasksPerServer, i));
            executorService.execute(servers.get(i));
        }
        changeStrategy(selectionPolicy);
    }

    1 usage new *
    public void changeStrategy(SelectionPolicy policy) {
        if (policy == SelectionPolicy.SHORTEST_QUEUE) {
            strategy = new ConcreteStrategyQueue();
        } else if (policy == SelectionPolicy.SHORTEST_TIME) {
            strategy = new ConcreteStrategyTime();
        }
    }

    1 usage new *
    public void dispatchTask(Task t) {

```

```

    1 usage new *
    public void dispatchTask(Task t) {
        strategy.addTask(servers, t);
    }

    1 usage new *
    public List<Server> getServers() {
        return servers;
    }

    no usages new *
    public void shutdown() {
        executorService.shutdown();
    }
}

```

Clasa SimulationManager este definita in pachetul „BusinessLogic” si implementeaza interfata Runnable.

Acest cod implementeaz  o simulare a unui sistem de procesare a task-urilor de c tre un num r specificat de servere.  n mod concret, se genereaz  un num r specificat de task-uri cu timpuri de sosire  i de procesare aleatoare  i se organizeaz  cozi de a teptare pentru acestea  n func ie de politica de selec ie specificat . Se afi eaz  starea sistemului la fiecare secund  a simul rii  i se stocheaz  informa iile  ntr-un fi ier text. Simularea se ruleaz  pe un fir de execu ie separat.

```

public class SimulationManager implements Runnable {
    3 usages
    private Scheduler scheduler;
    3 usages
    private SimulationFrame frame;
    9 usages
    private List<Task> generatedTasks;
    2 usages
    Integer timeLimit;
    2 usages
    Integer maxProcessingTime;
    3 usages
    Integer minProcessingTime;
    3 usages
    Integer minArrivalTime;
    2 usages
    Integer maxArrivalTime;
    2 usages
    Integer noOfServers;
    3 usages
    Integer noOfClients;
    1 usage
    SelectionPolicy selectionPolicy=SelectionPolicy.SHORTEST_TIME;

    1 usage new *
    public SimulationManager(){
        this.frame=new SimulationFrame();
        setValues(frame.getInitValues());
        this.scheduler= new Scheduler(noOfServers, maxTasksPerServer 5,selectionPolicy);
    }
}

```

```

        stack=stack+"closed";
    } else {
        Task[] tasks=i.getTasks();
        for(int j=0;j<i.getQueueSize();j++)
            stack=stack+"("+tasks[j].getNumar()+" "+tasks[j].getArrivalTime()+" "+tasks[j].getServiceTime()+"");
    }
    stack=stack+"\n";
}
return stack;
}

new *
@Override
public void run(){
    Integer currentTime=0;
    String display;
    String fin=" ";
    while (currentTime<timeLimit)
    {
        int i=0;
        while(i==0)
        {
            if (generatedTasks.size()!=0){
                Task j=generatedTasks.get(0);
                if (j.getArrivalTime()==currentTime)
                {
                    scheduler.dispatchTask(j);
                    generatedTasks.remove(j);
                }
            } else if (j.getArrivalTime()>currentTime)
            {

```

```

this.scheduler = new Scheduler(noOfServers, maxTasksPerServer, selectionPolicy);
generatedTasks = Collections.synchronizedList(new ArrayList<>());
generateNRandomTasks();
}

1 usage: new *
public void generateNRandomTasks(){
    Random random = new Random();
    int arrival;
    int service;
    for (Integer i=0; i<noOfClients; i++){
        service = random.nextInt( bound: maxProcessingTime-minProcessingTime)+minProcessingTime;
        arrival = random.nextInt( bound: maxArrivalTime-minArrivalTime)+minArrivalTime;
        generatedTasks.add(new Task(service, arrival));
    }
    generatedTasks.sort((Comparator.comparing(Task::getArrivalTime)));
    for (int i=0; i<noOfClients; i++){
        generatedTasks.get(i).setNumar(i);
    }
}

1 usage: new *
private String printStack(){
    String stack = "Waiting clients: ";
    for (Task i: generatedTasks)
    {
        stack = stack + "(" + i.getNumar() + ", " + i.getArrivalTime() + ", " + i.getServiceTime() + ") ";
    }
    stack = stack + "\n";
    for (Server i: scheduler.getServers()){
        stack = stack + "Queue " + (i.getServerId()+1) + ": ";
        if (i.getQueueSize() == 0)
            stack = stack + "closed";
    }
}

```

```

        {
            i--;
        }
    }

    if (generatedTasks.size() == 0)
        i = -1;
}

display = "Time " + currentTime + "\n" + printStack();
fin = fin + display;
System.out.println(display);
frame.refreshFrame(display);
currentTime++;
try{
    Thread.sleep( mills: 1000);
} catch (InterruptedException e) {
    e.printStackTrace();
}
}

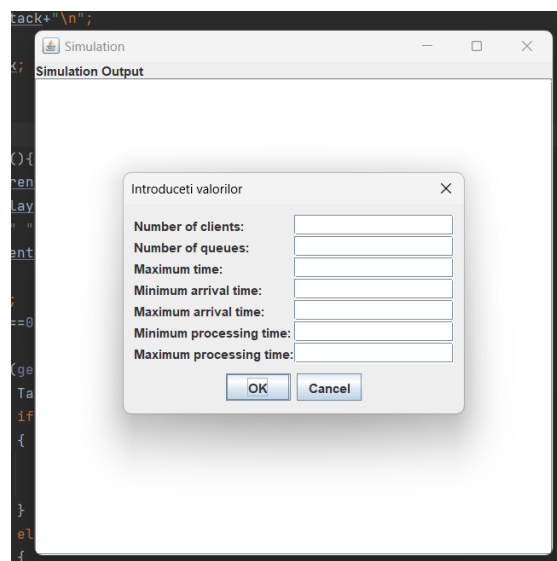
try (FileWriter fileWriter = new FileWriter( fileName: "log.txt"))
{
    fileWriter.write(fin);
}
catch (IOException ioException)
{
    ioException.printStackTrace();
}
}

1 usage: new *
public void setValues(List<Integer> list){
    this.noOfClients = list.get(0);
    this.noOfServers = list.get(1);
    this.timeLimit = list.get(2);
    this.minArrivalTime = list.get(3);
    this.maxArrivalTime = list.get(4);
    this.minProcessingTime = list.get(5);
    this.maxProcessingTime = list.get(6);
}

no usages: new *
public static void main(String[] args){
    SimulationManager gen = new SimulationManager();
    Thread t = new Thread(gen);
    t.start();
}

```

Clasa SimulationFrame este definita in pachetul „GUI” si implementeaza JFrame.



5. Rezultate

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

Simulation

Simulation Output

Time 0

Waiting clients: (0,2,8),(1,2,9),(2,2,24),(3,3,28);

Queue 1: closed

Queue 2: closed

Time 1

Waiting clients: (0,2,8),(1,2,9),(2,2,24),(3,3,28);

Queue 1: closed

Queue 2: closed

Time 2

Waiting clients: (3,3,28);

Queue 1: closed

Queue 2: (0,2,8),(1,2,9),(2,2,24);

Time 3

Waiting clients:

Queue 1: closed

Queue 2: (0,2,7),(1,2,9),(2,2,24),(3,3,28);

Time 4

Waiting clients:

Queue 1: closed

Queue 2: (0,2,6),(1,2,9),(2,2,24),(3,3,28);

Time 5

Waiting clients:

Queue 1: closed

Time 5

Waiting clients:

Queue 1: closed

Queue 2: (0,2,5),(1,2,9),(2,2,24),(3,3,28);

Time 6

Waiting clients:

Queue 1: closed

Queue 2: (0,2,4),(1,2,9),(2,2,24),(3,3,28);

Time 7

Waiting clients:

Queue 1: closed

Queue 2: (0,2,3),(1,2,9),(2,2,24),(3,3,28);

Time 8

Waiting clients:

Queue 1: closed

Queue 2: (0,2,2),(1,2,9),(2,2,24),(3,3,28);

Time 9

Waiting clients:

Queue 1: closed

Queue 2: (0,2,1),(1,2,9),(2,2,24),(3,3,28);

Time 10

Waiting clients:

Simulation

Simulation Output

Queue 2: (0,2,1),(1,2,9),(2,2,24),(3,3,28);

Time 10

Waiting clients:

Queue 1: closed

Queue 2: (1,2,9),(2,2,24),(3,3,28);

Time 11

Waiting clients:

Queue 1: closed

Queue 2: (1,2,8),(2,2,24),(3,3,28);

Time 12

Waiting clients:

Queue 1: closed

Queue 2: (1,2,7),(2,2,24),(3,3,28);

Time 13

Waiting clients:

Queue 1: closed

Queue 2: (1,2,6),(2,2,24),(3,3,28);

Time 14

Waiting clients:

Queue 1: closed

Queue 2: (1,2,5),(2,2,24),(3,3,28);

Time 15

6. Concluzii

În cadrul acestui proiect, am învățat despre thread-uri și am utilizat pentru prima dată variabilele AtomicInteger și BlockingQueue. Am înțeles de ce este important să le folosim în contextul acestei teme și conceptul de operație atomică.

De asemenea, am realizat că testarea unei aplicații poate evidenția detalii care pot îmbunătăți funcționarea acesteia. În viitor, ar putea fi dezvoltate situații neprevăzute pentru a îmbunătăți performanța și interfața grafică a aplicației.

7. Bibliografie

- 1) Generating Random Numbers in a Range - <https://www.baeldung.com/javagenerating-random-numbers-in-range>
- 2) Write to a File - https://www.w3schools.com/java/java_files_create.asp
- 3) <https://dsrl.eu/courses/pt>
- 4) http://www.tutorialspoint.com/java/util/timer_schedule_period.htm