DOCUMENTATIE

TEMA *NUMARUL 2*

NUME STUDENT: Catruc Alexandru- Dan
GRUPA: 30228

CUPRINS

1. Obiectivul temei

Obiectivul principal al acestei teme a fost realizarea unui proiect care sa poata manipula un sistem de cozi precum cel din supermarketuri, restaurante fast-food etc folosind threaduri, oferindu-se informatii despre starea cozilor la fiecare moment de timp.

Obiectivele secundare sunt:

1. Dezvoltarea unei interfete grafice usor de utilizat: interfata trebuie sa fie simpla si intuitiva, in acest caz sa aiba locuri pentru input- urile necesare (numar client, timpul de simulare, numar de cozi si limitele de timp pentru aparitia clientilor si timpul lor de procesare cat timp stau la coada), un buton de start al simularii si un loc pentru rezultate.
2. Distribuirea corecta si eficienta a clientilor: punerea in cozi a clientilor astfel incat sa se obtina un timp de servire mediu cat mai bun.
3. Afisarea live a tuturor cozilor si a starilor acestora in fiecare moment de timp pe parcursul simularii.
4. Afisarea unor raporturi detaliate: in acest caz vom avea afisarea in TextPanel in interfata cu utilizatorul, afisarea in consola in cazul simularii care are loc in clasa Simulation, si afisarea in fisirere .txt a rexultatelor, in cazul nostrum a 3 scenarii de simulare, unul avand loc pe aprox. 3 minute, iar celelalte doua pe cate un minut.
5. Asigurarea ca aplicatia poate sa sustina simulari de cozi si in caz de introducere de date foarte mari.
6. Asigurarea erorilor de input: in cazul in care datele introduse nu sunt corecte se vor afisa mesaje de eroare.
7. Realizarea de teste care sa arate functionalitatea corecta a programului si ca totul functioneaza corect conform standardelor.

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

Cerinte functionale:

1. Generarea clienților: Sistemul ar trebui să genereze un număr specificat de clienți cu sosire aleatoare și timpi de service în limitele date.

2. Alocarea cozii: Sistemul trebuie să aloce clienții la coadă cu timpul minim de așteptare atunci când timpul lor de sosire este mai mare sau egal cu timpul de simulare.

3.Procesare în coadă: sistemul ar trebui să proceseze clienții din fiecare coadă într-o ordine primul venit, primul servit.

4. Controlul simulării: sistemul ar trebui să permită utilizatorilor să pornească, să întrerupă și să oprească simularea.

5. Raportare: Sistemul trebuie să calculeze și să afișeze timpul mediu de așteptare, precum și alte statistici relevante, la sfârșitul simulării.

6. Validarea intrărilor: Sistemul trebuie să valideze intrările utilizatorului pentru a se asigura că se încadrează în constrângerile specificate și să furnizeze mesaje de eroare adecvate atunci când este necesar.

Cerinte non-functionale:

1. Performanță: Sistemul ar trebui să poată gestiona un număr mare de clienți și cozi în mod eficient, cu întârzieri minime în procesarea și afișarea rezultatelor.

2. Scalabilitate: sistemul ar trebui să fie proiectat pentru a se adapta creșterii viitoare, cum ar fi un număr crescut de clienți sau cozi, fără o degradare semnificativă a performanței.

3. Utilizabilitate: interfața cu utilizatorul ar trebui să fie simplă, intuitivă și ușor de navigat, chiar și pentru utilizatorii care nu sunt familiarizați cu sistemele de gestionare a cozilor.

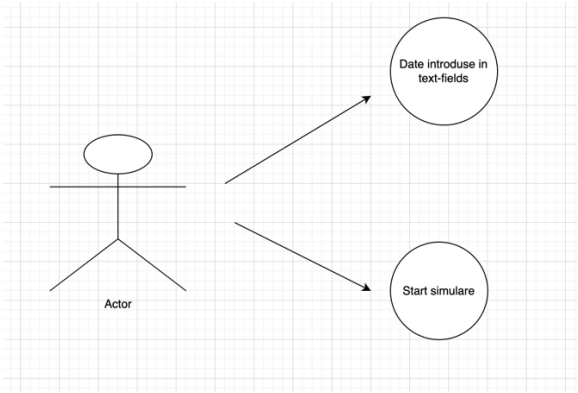
4. Fiabilitate: Sistemul trebuie să fie robust și să ofere rezultate consistente, chiar și în cazul unor erori sau probleme neașteptate în timpul simulării.

5. Mentenabilitatea: codul trebuie să fie bine organizat, modular și documentat corespunzător, astfel încât dezvoltatorilor să fie ușor de întreținut și extins sistemul în viitor.

6. Reactivitate: Interfața de utilizator ar trebui să ofere actualizări în timp real cu privire la starea curentă a cozilor și a clienților pe măsură ce simularea progresează.

Use-cases:

1. Introducerea datelor in text-fields si validarea acestora.
2. Start simularii si analizarea live a datelor pe parcursul rularii programului.



3. Proiectare

Diagrama de pachete:

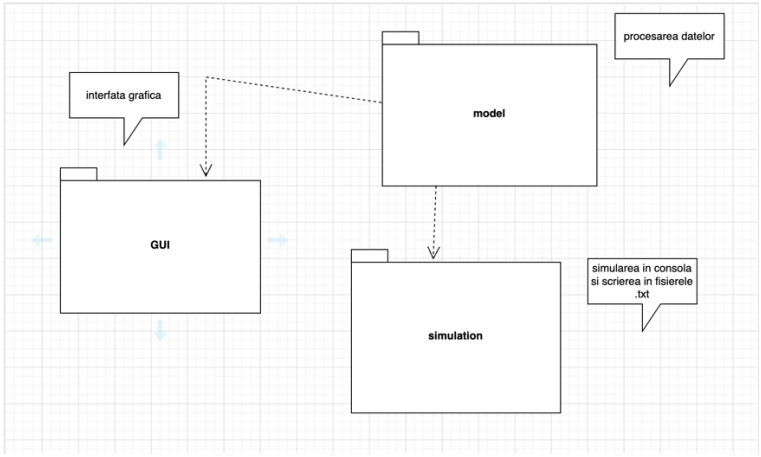
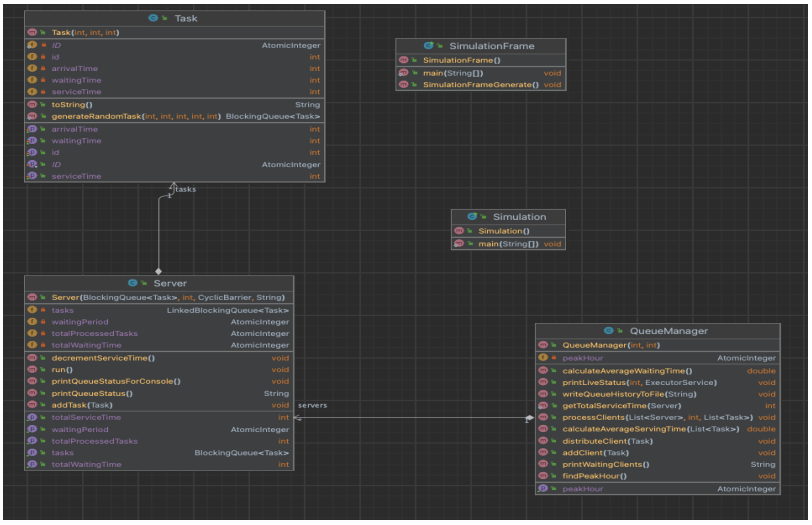


Diagrama de clase:

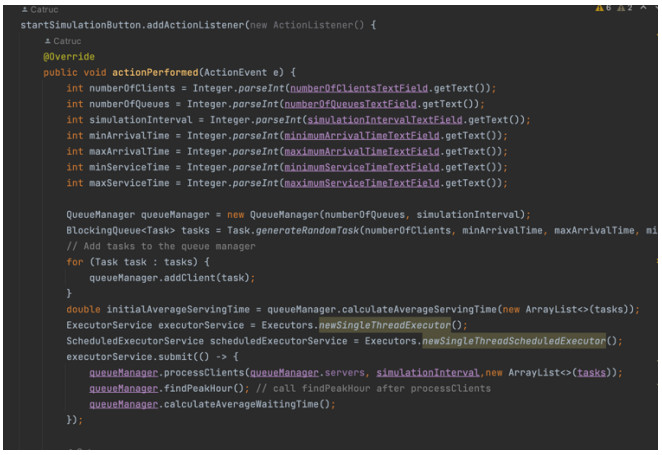


4. Implementare

a. Clasa SimulationFrame:

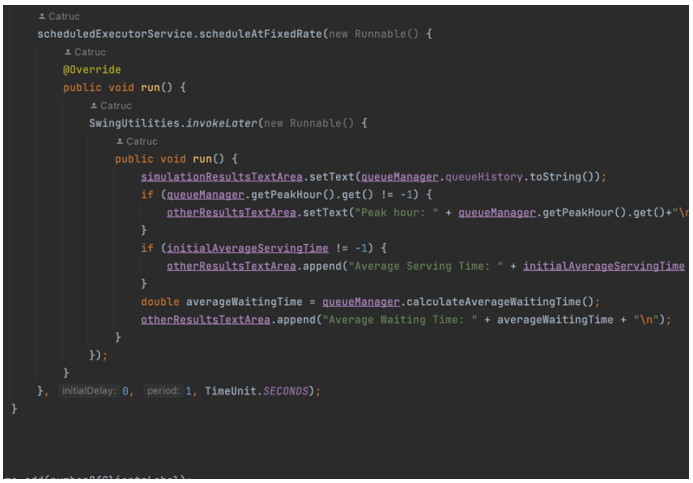
1.SimulationFrameGenerate(): Această metodă configurează cadrul principal al simularii, inclusiv crearea și poziționarea tuturor componentelor necesare ale interfeței grafice, cum ar fi etichetele, câmpurile de text, butoanele și zonele de text. De asemenea, definește ascultătorul de acțiuni pentru butonul startSimulationButton și gestionează configurarea și procesarea simulației.
2.actionPerformed(ActionEvent e): Acesta este ascultătorul de acțiuni pentru butonul startSimulationButton. Este declanșat când utilizatorul face clic pe buton. Acesta preia datele de intrare din câmpurile de text, creează QueueManager și lista de sarcini, apoi procesează clienții în cozi. De asemenea, programează actualizări pentru simulationResultsTextArea și otherResultsTextArea.
3.Task.generateRandomTask(...): Această metodă statică generează o listă de obiecte Task cu timpi de sosire și de servire aleatorii în limitele date. Este utilizată pentru a crea lista de sarcini care vor fi adăugate în QueueManager.
4.QueueManager.addClient(Task task): Această metodă adaugă un client (obiect Task) în coada cu timpul minim de așteptare atunci când timpul de sosire al clientului este mai mare sau egal cu timpul de simulare.
5.QueueManager.processClients(...): Această metodă procesează clienții în fiecare coadă în ordinea sosirii. Este apelată de serviciul executor.
6.QueueManager.findPeakHour(): Această metodă găsește ora de vârf, adică ora cu numărul maxim de clienți deserviți. Este apelată după metoda processClients.
7.QueueManager.calculateAverageWaitingTime(): Această metodă calculează și returnează timpul mediu de așteptare pentru toți clienții din simulare.
8.Runnable(): Aceasta este o implementare anonimă a interfeței Runnable, care este programată să se execute periodic folosind un ScheduledExecutorService. Acesta actualizează simulationResultsTextArea și otherResultsTextArea cu cele mai recente rezultate ale simulării.
9.SwingUtilities.invokeLater(...): Această metodă se asigură că actualizările componentelor UI (simulationResultsTextArea și otherResultsTextArea) sunt executate pe firul Event Dispatch Thread, care este responsabil pentru gestionarea actualizărilor UI în aplicațiile Swing.
10.main(String[] args): Acesta este punctul de intrare al aplicației. Creează o instanță a clasei SimulationFrame și apelează metoda SimulationFrameGenerate() pentru a configura și afișa cadrul principal al simulației.

Codul furnizat configurează o interfață grafică pentru aplicația de gestionare a cozilor, permițând utilizatorilor să introduce date cu scopul de a fi procesate si de a afisa pe un TextPanel rezultatele simularii.



```
startSimulationButton.addActionListener(new ActionListener() {
    Catruc
    @Override
    public void actionPerformed(ActionEvent e) {
        int numberOfClients = Integer.parseInt(numberOfClientsTextField.getText());
        int numberOfQueues = Integer.parseInt(numberOfQueuesTextField.getText());
        int simulationInterval = Integer.parseInt(simulationIntervalTextField.getText());
        int minArrivalTime = Integer.parseInt(minimumArrivalTimeTextField.getText());
        int maxArrivalTime = Integer.parseInt(maximumArrivalTimeTextField.getText());
        int minServiceTime = Integer.parseInt(minimumServiceTimeTextField.getText());
        int maxServiceTime = Integer.parseInt(maximumServiceTimeTextField.getText());

        QueueManager queueManager = new QueueManager(numberOfQueues, simulationInterval);
        BlockingQueue<Task> tasks = Task.generateRandomTask(numberOfClients, minArrivalTime, maxArrivalTime, mi
        // Add tasks to the queue manager
        for (Task task : tasks) {
            queueManager.addClient(task);
        }
        double initialAverageServingTime = queueManager.calculateAverageServingTime(new ArrayList<>(tasks));
        ExecutorService executorService = Executors.newSingleThreadExecutor();
        ScheduledExecutorService scheduledExecutorService = Executors.newSingleThreadScheduledExecutor();
        executorService.submit(() -> {
            queueManager.processClients(queueManager.servers, simulationInterval,new ArrayList<>(tasks));
            queueManager.findPeakHour(); // call findPeakHour after processClients
            queueManager.calculateAverageWaitingTime();
        });
```



b. Clasa QueueManager:

Clasa QueueManager este o clasă care gestionează cozi și servere într-o aplicație de simulare a cozilor.Aceasta se ocupă de distribuirea clienților către servere și calculează statistici legate de performanța sistemului de cozi. Iată o descriere a atributelor și metodelor clasei:

Atribute:

-servers: O listă de obiecte Server, reprezentând serverele care gestionează clienții.
-numServers: Numărul de servere utilizate în simulare.
-waitingClients: O coadă de priorități care conține clienții care așteaptă să fie distribuiți către servere.
-maxSimulationTime: Timpul maxim de simulare.
-barrier: Un obiect CyclicBarrier utilizat pentru a sincroniza serverele.
-queueHistory: Un StringBuilder care păstrează istoricul evenimentelor de coadă.
-tasksPerTimeInterval: O mapare între momentele de timp și numărul total de sarcini din sistem în acel moment.
-peakHour: Un AtomicInteger care reprezintă ora de vârf în simulare.

Metode:

1. QueueManager(int numServers, int maxSimulationTime): Constructorul clasei care inițializează atributelor și creează serverele și thread-urile asociate.

```java
public QueueManager(int numServers, int maxSimulationTime) {
    this.numServers = numServers;     // Number of servers
    this.maxSimulationTime = maxSimulationTime;  // Maximum simulation time
    servers = new ArrayList<>();  // List of servers
    waitingClients = new PriorityQueue<>(Comparator.comparingInt(Task::getArrivalTime));  // List of waiting clients
    barrier = new CyclicBarrier(numServers);   // Cyclic barrier for the servers
    this.queueHistory = new StringBuilder();    // String builder for the queue history
    tasksPerTimeInterval = new HashMap<>();    // Map for the tasks per time interval
    peakHour = new AtomicInteger( initialValue: 0);   // Atomic integer for the peak hour

    for (int i = 0; i < numServers; i++) {
        BlockingQueue<Task> taskQueue = new LinkedBlockingQueue<>();  // Create a queue for each server
        String serverName = "QUEUE"+(i+1);   // Create a name for each server
        Server server = new Server(taskQueue, maxSimulationTime, barrier,serverName);  // Create a server with the queue
        servers.add(server);  // Add the server to the list of servers
        Thread serverThread = new Thread(server);  // Create a thread for the server
        serverThread.setName(serverName);  // Set the name of the thread
        serverThread.start();  // Start the thread
    }

}
```

2.getPeakHour(): Returnează ora de vârf în simulare.
3.addClient(Task task): Adaugă un client în coada de așteptare waitingClients.
4.printWaitingClients(): Returnează un șir de caractere care reprezintă clienții care așteaptă să fie distribuiți către servere.
5.getTotalServiceTime(Server server): Calculează și returnează timpul total de servire pentru un server dat.
6.distributeClient(Task client): Distribuie un client către serverul cu cele mai puține sarcini sau cu timpul minim de servire.

```java
public synchronized void distributeClient(Task client) {   // Distribute the client to the server with the least tasks
    if (client.getArrivalTime() <= maxSimulationTime) {  // If the client's arrival time is less than the maximum simulation tim
        int minTasks = Integer.MAX_VALUE;   // Minimum number of tasks
        int minServiceTime = Integer.MAX_VALUE;   // Minimum service time
        Server bestServer = null;   // Best server

        for (Server server : servers) {   // For each server
            int currentTasks = server.getTasks().size();  // Get the number of tasks
            int currentServiceTime = getTotalServiceTime(server);   // Get the total service time

            if (currentTasks < minTasks || (currentTasks == minTasks && currentServiceTime < minServiceTime)) {  // If the curre
                minTasks = currentTasks;   // Set the minimum number of tasks to the current number of tasks
                minServiceTime = currentServiceTime;   // Set the minimum service time to the current service time
                bestServer = server;   // Set the best server to the current server
            }
        }

        if (bestServer != null) {   // If the best server is not null
            bestServer.addTask(client);   // Add the client to the best server
        }
    }
}
```

7.findPeakHour(): Găsește și setează ora de vârf în simulare.
8.printLiveStatus(int currentTime, ExecutorService executorService): Afișează și înregistrează starea curentă a cozilor și a clienților în așteptare.

```
1 usage   1 Catruc
public void printLiveStatus(int currentTime,ExecutorService executorService){
    int totalTasks = servers.stream().mapToInt(server -> server.getTasks().size()).sum();
    tasksPerTimeInterval.put(currentTime, totalTasks);
    System.out.println("Time: " + currentTime);
    queueHistory.append("Time: ").append(currentTime).append(System.lineSeparator());
    String waitingClientsStatus = printWaitingClients();
    System.out.println(waitingClientsStatus);
    queueHistory.append(waitingClientsStatus).append(System.lineSeparator());
    for (Server server : servers) {
        Future<?> future = executorService.submit(server::printQueueStatusForConsole);  // Print the queue status o
        try {
            future.get();
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }
    }
    servers.forEach(server -> {
        queueHistory.append(server.printQueueStatus()).append(System.lineSeparator());
    });
}
```

9.processClients(List<Server> servers, int maxSimulationTime, List<Task> tasks): Procesează clienții și actualizează starea sistemului de cozi în funcție de timpul de simulare.

```
public void processClients(List<Server> servers, int maxSimulationTime, List<Task> tasks) {
    ExecutorService executorService = Executors.newFixedThreadPool(numServers);  // Create a fixed thread pool with the numbe
    int currentTime = 0;  // Current time
    int stop = 1;  // Stop variable
    double averageServingTime = calculateAverageServingTime(tasks);  // Calculate the average serving time
    while (!waitingClients.isEmpty() || currentTime < maxSimulationTime||stop==1) {   // While there are waiting clients or t
        while (!waitingClients.isEmpty() && waitingClients.peek().getArrivalTime() == currentTime) {   // While there are wai
            Task client = waitingClients.poll();   // Get the first client
            distributeClient(client);   // Distribute the client
        }
        printLiveStatus(currentTime,executorService);

        try {
            Thread.sleep( millis: 1000); // Sleep for 1 second to simulate 1 second of simulation time
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        currentTime++;   // Increment the current time
        stop = 0;   // Set the stop variable to 0
    }
    executorService.shutdown();   // Shutdown the executor service
    if (stop == 0) {
        queueHistory.append("Simulation finished").append(System.lineSeparator());
        findPeakHour();
        queueHistory.append("Peak hour: ").append(peakHour).append(System.lineSeparator());
        queueHistory.append("Average serving time: ").append(averageServingTime).append(System.lineSeparator());
        double averageWaitingTime = calculateAverageWaitingTime(); // Calculate the average waiting time
        queueHistory.append("Average waiting time: ").append(averageWaitingTime).append(System.lineSeparator()); // Append th
        System.out.println(averageWaitingTime);
```

10.writeQueueHistoryToFile(String fileName): Salvează istoricul cozilor într-un fișier cu numele specificat.
11.calculateAverageServingTime(List<Task> tasks): Calculează și returnează timpul mediu de servire pentru lista de sarcini dată.
12.calculateAverageWaitingTime(): Calculează și returnează timpul mediu de așteptare pentru toți clienții din sistemul de cozi.

În general, clasa **QueueManager** este responsabilă pentru gestionarea serverelor și a cozilor de clienți într-o aplicație de simulare.

c. clasa Server

Clasa Server reprezintă un server într-un sistem bazat pe cozi, unde sarcinile sunt procesate. Implementează interfața Runnable, permițându-i să fie executată ca un fir de execuție separat.

Variabile de instanță:

1.tasks: Un obiect LinkedBlockingQueue pentru a stoca sarcinile care așteaptă să fie procesate de server.
2.waitingPeriod: Un AtomicInteger care reprezintă perioada de așteptare.
3.maxSimulationTime: Un întreg ce reprezintă timpul maxim de simulare.
4.barrier: Un obiect CyclicBarrier utilizat pentru a sincroniza serverele.
5.serverName: Un șir de caractere ce reprezintă numele serverului.
6.totalWaitingTime: Un AtomicInteger care reprezintă timpul total de așteptare.
7.totalProcessedTasks: Un AtomicInteger care reprezintă numărul total de sarcini procesate.

Metode:

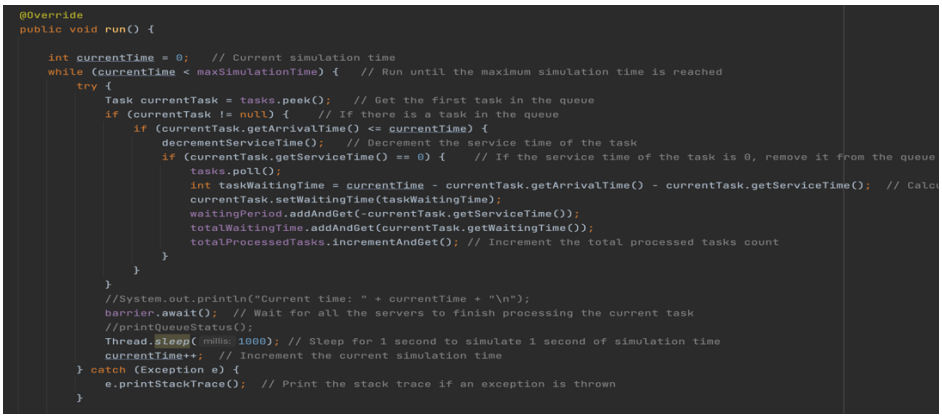
1.Constructorul clasei Server inițializează variabilele de instanță și creează un LinkedBlockingQueue nou pentru a stoca sarcinile.

```
1 usage   Catruc
public Server(BlockingQueue<Task> tasks, int maxSimulationTime,CyclicBarrier barrier,String serverName) {
    this.tasks = new LinkedBlockingQueue<>();
    this.maxSimulationTime = maxSimulationTime;
    this.waitingPeriod = new AtomicInteger( initialValue: 0);
    this.barrier=barrier;
    this.serverName=serverName;
    this.totalWaitingTime = new AtomicInteger( initialValue: 0);
    this.totalProcessedTasks = new AtomicInteger( initialValue: 0);
}
```

2.addTask(Task task): Adaugă o sarcină în coada de sarcini.
3.getTasks(): Returnează coada de sarcini.
4.getWaitingPeriod(): Returnează perioada de așteptare.
5.getTotalWaitingTime(): Returnează timpul total de așteptare.
6.decrementServiceTime(): Decrementează timpul de servire al sarcinii curente, dacă există vreo sarcină în coadă.

```
1 usage   Catruc
public void decrementServiceTime() {
    if (!tasks.isEmpty()) {
        Task currentTask = tasks.peek();
        if (currentTask.getServiceTime() > 0) {
            currentTask.setServiceTime(currentTask.getServiceTime() - 1);
        }
    }
}
```

7.run(): Suprascrie metoda run() a interfeței Runnable. Conține logica principală de procesare a sarcinilor în timpul simulării.



```java
@Override
public void run() {

    int currentTime = 0;   // Current simulation time
    while (currentTime < maxSimulationTime) {   // Run until the maximum simulation time is reached
        try {
            Task currentTask = tasks.peek();   // Get the first task in the queue
            if (currentTask != null) {    // If there is a task in the queue
                if (currentTask.getArrivalTime() <= currentTime) {
                    decrementServiceTime();   // Decrement the service time of the task
                    if (currentTask.getServiceTime() == 0) {    // If the service time of the task is 0, remove it from the queue
                        tasks.poll();
                        int taskWaitingTime = currentTime - currentTask.getArrivalTime() - currentTask.getServiceTime();  // Calc
                        currentTask.setWaitingTime(taskWaitingTime);
                        waitingPeriod.addAndGet(-currentTask.getServiceTime());
                        totalWaitingTime.addAndGet(currentTask.getWaitingTime());
                        totalProcessedTasks.incrementAndGet(); // Increment the total processed tasks count
                    }
                }
            }
            //System.out.println("Current time: " + currentTime + "\n");
            barrier.await();  // Wait for all the servers to finish processing the current task
            //printQueueStatus();
            Thread.sleep( millis: 1000); // Sleep for 1 second to simulate 1 second of simulation time
            currentTime++;  // Increment the current simulation time
        } catch (Exception e) {
            e.printStackTrace();  // Print the stack trace if an exception is thrown
        }
```

8.getTotalServiceTime(): Returnează timpul total de servire al tuturor sarcinilor din coadă.
9.printQueueStatus(): Returnează un șir de caractere ce reprezintă starea curentă a cozii și numele serverului.
10.printQueueStatusForConsole(): Afișează starea curentă a cozii și numele serverului în consolă.
11.getTotalProcessedTasks(): Returnează numărul total de sarcini procesate.

d. clasa Task:

Clasa Task reprezintă o sarcină care trebuie procesată de un server într-un sistem bazat pe cozi.

Variabile de instanță:

ID: Un AtomicInteger static care reprezintă ID-ul global al sarcinilor. Acesta se incrementează de fiecare dată când este creată o sarcină nouă.
id: Un întreg care reprezintă ID-ul local al sarcinii.
arrivalTime: Un întreg care reprezintă timpul de sosire al sarcinii.
serviceTime: Un întreg care reprezintă timpul necesar pentru a procesa sarcina.
waitingTime: Un întreg care reprezintă timpul de așteptare al sarcinii în coadă înainte de a fi procesată.

Metode:

1.Constructorul clasei Task inițializează variabilele de instanță cu valorile primite ca argumente.
2.Getteri și setteri pentru variabilele de instanță: getWaitingTime(), setWaitingTime(int waitingTime),
3.getId(), getArrivalTime(), getServiceTime(), setID(AtomicInteger ID), setId(int id), setArrivalTime(int arrivalTime), setServiceTime(int serviceTime).
4.generateRandomTask(int numberOfTasks, int minArrivalTime, int maxArrivalTime, int minServiceTime, int maxServiceTime): Această metodă statică generează un număr specificat de sarcini aleatoare cu timp de sosire și timp de servire între limitele date și le adaugă într-o coadă de tip LinkedBlockingQueue. Returnează coada de sarcini generate.
5.toString(): Suprascrie metoda toString() pentru a returna un șir de caractere ce reprezintă obiectul Task, incluzând ID-ul, timpul de sosire și timpul de servire.

e. clasa Simulation:

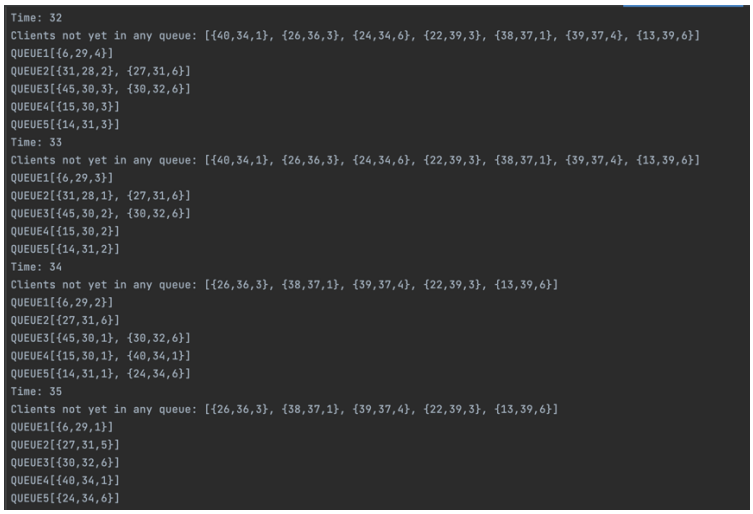
Clasa Simulation este punctul de intrare al unei aplicații Java care simulează un sistem de cozi. În cadrul metodei main, sunt create un manager de cozi (QueueManager), sarcini aleatoare (Task) și apoi sunt adăugate

în managerul de cozi. Apoi, sarcinile sunt procesate de serverele din managerul de cozi, iar istoricul cozilor este salvat într-un fișier pentru analiză ulterioară.

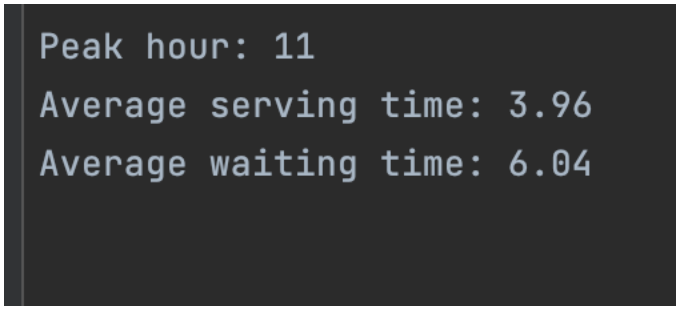
5. Rezultate

In aceasta tema nu au fost necesare testari cu ajutorul Junit dar au fost afisate rezultatele in fisiere .txt care prezinta starea cozilor la fiecare moment de timp (in acest caz o unitate de timp = 1 secunda).

Un exemplu de testare cu ajutorul afisarii rezultatelor in .txt file:



In prima parte a fisierului se vor afisa clientii care inca nu sunt in nicio coada si starea cozilor actualizata plus momentul de timp.



In ultima parte se vor afisa peak hour, average serving time si average waiting time, care se calculeaza pe parcursul simularii si se afiseaza la final.

Rezultatele sunt conform asteptarilor deoarece pentru managementul cozilor au fost luate in considerare toate cazurile care ar putea da peste cap analiza problemei precum: daca doi clienti intra intr-o coada in acelasi moment de timp, acestia sa fie pusi in aceeasi coada, sau daca cozile au numar egal de client, noul client sau noii clienti care urmeaza sa fie bagati in coada, sa fie pusi unde timpul de asteptare este cel mai scurt. Legat de ora de varf, se va afisa una singura, acolo unde au fost cei mai multi clienti intr-un anumit moment de timp. In cazul in care vor fi doua ore de varf, se va afisa doar una, adica prima gasita.

6.Concluzii

In timpul realizarii acestei teme s-au invatat in primul rand multe lucruri noi precum: lucrul cu threaduri, folosirea a noi interfete precum cea Runnable (care are utilitatea in manipularea threadurilor), scrierea unor informatii in fisiere text (lucru care pana acum nu l-am facut in mediul Java). Pe langa acestea au mai fost si tipurile de date precum AtomicInteger sau structurile de date cum ar fi BlockingQueue pe care nu le-am mai folosit pana la realizarea acestui proiect.

Ca si implementare pot sa zic ca a fost cu adevarat o provocare la inceput pana am realizat cum sa fac legatura intre clase si cum sa manipulez acele taskuri (clientii) in diferite servere (cozile).

In concluzie, acest proiect de Queue Management a fost cu adevarat de foarte mare ajutor pentru invatarea de noi lucruri si mai ales cele de manipulare a threadurilor cu care nu am mai lucrat deloc inainte, stiind despre ele doar o scurta teorie.

7.Bibliografie

1. Runnable Interface: https://www.upgrad.com/blog/runnable-interface-in-java/#:~:text=A%20runnable%20interface%20in%20Java,Threads%20must%20implement%20this%20interface.
2. Java Threads: https://www.w3schools.com/java/java_threads.asp
3. Random in Java: https://www.javatpoint.com/how-to-generate-random-number-in-java
4. Multithreading: https://www.javatpoint.com/multithreading-in-java
5. Queues: https://www.softwaretestinghelp.com/java-queue-interface/
6. JTextArea: https://www.javatpoint.com/java-jtextarea
7. CyclicBarrier: https://www.baeldung.com/java-cyclic-barrier