

DOCUMENTAȚIE

TEMA 2

NUME STUDENT: Marcu Ariana-Mălina
GRUPA: 30222

CUPRINS

1.	Obiectivul temei	3
2.	Analiza problemei, modelare, scenarii, cazuri de utilizare	3
3.	Proiectare.....	4
4.	Implementare.....	6
5.	Rezultate.....	9
6.	Concluzii	10
7.	Bibliografie	10

1. Obiectivul temei

Obiectivul principal al temei este implementarea unei aplicatii de gestionare a cozilor, in JAVA, care atribuie clienților cozi, astfel încât timpul de așteptare este minimizat.

Obiective secundare:

- Definirea cerințelor pentru aplicația de management al cozilor. Proiectarea aplicației de management al cozilor, inclusiv definirea algoritmului pentru minimizarea timpului de așteptare, pași care vor fi detaliați în următoarele două capitole ale documentației.
- Implementarea aplicației de management al cozilor, inclusiv definirea structurii de date pentru reprezentarea cozilor și clienților (clasele Server și Client – capitolul 4)
- Testarea aplicației și evaluarea performanțelor acesteia în ceea ce privește minimizarea timpului de așteptare (mai multe în capitolul 5)

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

Cozile sunt folosite frecvent pentru a modela domenii din lumea reală. Obiectivul principal al unei cozi este de a furniza un loc pentru ca un "client" să aștepte înainte de a primi un "serviciu". Managementul sistemelor bazate pe cozi este interesat de minimizarea timpului în care "clientii" așteaptă în cozi înainte de a fi serviți. Una dintre modalitățile de a minimiza timpul de așteptare este de a adăuga mai multe servere, adică mai multe cozi în sistem, dar acest lucru crește costurile furnizorului de servicii.

Aplicația de gestionare a cozilor ar trebui să simuleze (prin definirea unui timp de simulare *tsimulation*) o serie de N clienți care vin pentru serviciu, intra în Q cozi, așteaptă, sunt serviți și în cele din urmă părăsesc cozile. Toți clienții sunt generați random și sunt caracterizați de trei parametri: **ID** (un număr între 1 și N), **tarrival** (timpul de simulare când sunt pregătiți să intre în coadă) și **tservice** (intervalul de timp sau durata necesară pentru a servi clientul; adică timpul de așteptare când clientul se află în fața cozii). Aplicația urmărește timpul total petrecut de fiecare client la coadă și calculează timpul mediu de așteptare. Fiecare client este adăugat la coadă cu timpul minim de așteptare când *tarrival* este mai mare sau egal cu timpul de simulare ($tarrival \geq tsimulation$).

Cerintele functionale pentru aplicația de gestionare a cozilor ar fi următoarele:

- ✓ Generarea a N clienți, fiecare cu un ID unic, un timp de sosire și un timp de servire.
- ✓ Simularea a N clienți care ajung pentru servire și așteaptă în cozi.
- ✓ Adăugarea clienților în cozi cu minimul timp de așteptare disponibil.
- ✓ Atribuirea clienților către unul din mai multe servere (cozi), astfel încât timpul de așteptare să fie minimizat.
- ✓ Înregistrarea timpului total petrecut de fiecare client în cozi și calcularea timpului mediu de așteptare pentru toți clienții.
- ✓ Eliberarea clienților care au fost serviți și părăsirea cozilor.

Cerintele non-funcționale pentru această aplicație ar fi:

- ❖ Performanța: aplicația trebuie să fie capabilă să gestioneze N clienți într-un timp rezonabil și să minimizeze timpul de așteptare.
- ❖ Fiabilitate: aplicația trebuie să fie robustă și să poată face față unor situații neprevăzute sau erori în sistem.
- ❖ Scalabilitate: aplicația trebuie să poată suporta creșterea numărului de clienți fără a afecta performanța.
- ❖ Simplitate: Interfața utilizatorului trebuie să fie simplă și intuitivă pentru a putea fi utilizată de orice utilizator fără cunoștințe avansate de programare.
- ❖ Portabilitate: aplicația trebuie să poată fi rulată pe mai multe platforme fără a fi necesare modificări semnificative ale codului sursă.

O aplicație de gestionare a cozilor ar putea fi utilă într-o varietate de contexte din viața reală, cum ar fi: servicii bancare sau poștale, centre medicale (în aceste cazuri, pacienții trebuie să aștepte adesea în cozi pentru a vedea un medic sau pentru a efectua un test medical, servicii de asistență tehnică sau de suport: atunci când clienții așteaptă să primească asistență de la un operator sau un tehnician. În general, orice situație în care oamenii trebuie să aștepte într-o coadă pentru a primi un serviciu sau o asistență ar putea beneficia de o astfel de aplicație de gestionare a cozilor.

3. Proiectare

The screenshot shows a graphical user interface for a queue management simulation. It features several input fields for parameters: N , $T_{MinArrival}$, $T_{MinService}$, Q , $T_{MaxArrival}$, $T_{MaxService}$, and $T_{MaxSimulation}$. A blue "START" button is positioned to the right of the input fields. Below the inputs is a section labeled "LOG OF EVENTS:" followed by a large, empty rectangular box for displaying the simulation log. The interface is styled with a light gray background and standard Windows-style window controls (minimize, maximize, close) in the top right corner.

Mai sus avem interfața grafică realizată iar mai jos diagrama UML a codului sursă:

Pachetele in care mi-am organizat codul sunt:

1. Pachetul GUI(Graphical User Interface): contine clasa SimulationFrame.
2. Pachetul BusinessLogic: cu clasele ConcreteStrategyQueue, ConcreteStrategyTime, Input, Scheduler, SelectionPolicy, SimulationManager, Strategy.
3. Pachetul Model: cu cele doua clase de baza Server(Queue) si Client(Task).
4. Separat avem clasa Main si diferitele input-uri pentru testare.

4. Implementare

1. CLASA CLIENT

Clasa de baza, care contine settere si gettere pentru crearea unui obiect de tipul Client, cu cele 3 atribute ale sale cerute mai sus si constructorul. De asemenea, avem si metoda toString, pentru o afisare intuitiva.

```
private int ID;
private int arrivalTime;
private int serviceTime;

public Client(int ID, int arrivalTime, int serviceTime) {
    this.ID = ID;
    this.arrivalTime = arrivalTime;
    this.serviceTime = serviceTime;
}
```

2. CLASA SERVER

Clasa Server este responsabilă pentru gestionarea cozilor și procesarea clienților. Metoda addClient adaugă un client în coada serverului, metoda run procesează clienții în ordinea sosirii lor și metoda toString returnează o reprezentare sub forma de șir de caractere a clienților din coada serverului. Metodele getter și setter permit accesul la coada de clienți și la perioada de așteptare. Clasa implementează interfața Runnable, ceea ce înseamnă că poate fi utilizată ca un fir de execuție.

```
private BlockingQueue<Client> clienti;
private AtomicInteger waitingPeriod;

public void run() {
    while(!clienti.isEmpty()){
        try {
            Thread.sleep(1000);
            setWaitingPeriod(new AtomicInteger(waitingPeriod.intValue()-1));
            Client client1 = clienti.peek(); //primul care e in coada
            synchronized (client1){
                client1.setServiceTime(client1.getServiceTime()-1);
            };
            if(client1.getServiceTime() == 0) //a fost servit
            {
                clienti.remove(client1);
            }
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
}
```

3. CLASA CONCRETSTRATEGYQUEUE

Clasa ConcreteStrategyQueue implementează interfața Strategy și definește strategia de a adăuga un client nou în coada serverului cu cel mai mic număr de clienți. Aceasta are o metodă care primește o listă de servere și un client, determină serverul cu cel mai puțini clienți și adaugă clientul în coada acelui server.

```
@Override
public void addClient(List<Server> servers, Client c)
{
    Server minS = getMinServer(servers);
    minS.addClient(c);
}
public Server getMinServer(List<Server> servers)
{
    Server min = null;
    Integer nrClienti = Integer.MAX_VALUE;
    for(Server s : servers){
        if(s.getClients().length < nrClienti){
            min = s;
            nrClienti = s.getClients().length;
        }
    }
    return min;
}
```

4. CLASA CONCRETSTRATEGYTIME

Pe același principiu, și această clasă implementează interfața Strategy și definește o strategie de adăugare a unui client la servere(cozi) bazată pe coada cu cel mai mic timp de așteptare. Metoda addClient primește o listă de servere și un client și adaugă clientul la serverul cu cel mai mic timp de așteptare. Metoda getMinTimeServer primește o listă de servere și returnează serverul cu cel mai mic timp de așteptare, determinat de valoarea atomică waitingPeriod a fiecărui server.

5. CLASA INPUT

Se ocupă de citirea și stocarea datelor dintr-un fișier de intrare și prelucrarea lor într-un format utilizabil în simularea de cozi. Aceasta citește dintr-un fișier de intrare formatat, inițializând proprietățile corespunzătoare obiectului de intrare. Proprietățile includ numărul de clienți, numărul de cozi, timpul total de simulare, intervalul de sosire și serviciu pentru clienți, precum și un obiect FileWriter care este utilizat pentru a scrie într-un fișier de ieșire. Fișierul de ieșire conține informații despre clienți, cum ar fi timpul de sosire, durata serviciului și timpul de așteptare. Aceste date ulterior vor fi folosite în simularea sistemului de cozi. Datele care vor fi citite de utilizator:

```
private Integer nrClients;
private Integer nrQueues;
private Integer tSimulation;
private Integer minArrivalTime;
private Integer maxArrivalTime;
private Integer minServiceTime;
private Integer maxServiceTime;
```

6. CLASA SCHEDULER

Clasa Scheduler reprezintă componenta centrală a sistemului de simulare a unei cozi de așteptare. Aceasta gestionează mai multe servere, fiecare cu o capacitate maximă de clienți, și îi atribuie pe aceștia în funcție de o strategie specifică, pe baza politicii de selecție a serverului cu cea mai scurtă coadă sau cea mai scurtă perioadă de așteptare.

În plus, Scheduler se ocupă și de crearea de noi servere în cazul în care toate cele existente sunt ocupate. Această clasă are metode pentru adăugarea de clienți la coadă, găsirea serverului cu cele mai mici valori și afișarea stării tuturor serverelor. Scheduler este esențial pentru simularea unei cozi de așteptare și asigurarea unui flux optim de clienți.

```
private List<Server> servers;
private int maxNoServers;
private int maxClientsPerServer;
private Strategy strategy;
private SelectionPolicy policy;
```

7. ENUM-UL SELECTIONPOLICY

Această clasă definește o listă de constante de selecție a politicii pentru program. Cele două constante definite reprezintă două politici de selecție diferite pe baza lungimii cozii și timpului de așteptare al clientului. Această clasă este utilizată în clasa Scheduler pentru a specifica strategia de selecție a serverului pentru a procesa un client nou.

```
package BusinessLogic;

public enum SelectionPolicy {
    SHORTEST_QUEUE, SHORTEST_TIME;
}
```

8. CLASA SIMULATIONMANAGER

Reprezintă nucleul aplicației și coordonează toate operațiunile necesare pentru simularea procesului de așteptare a clienților într-un magazin cu mai multe cozi de așteptare și mai mulți operatori (servere). În constructor, se realizează inițializarea obiectelor de intrare și se generează o listă de clienți aleatori cu intervale de sosire și de servire alese aleatoriu, din intervalul menționat ca input. Acești clienți sunt păstrați într-un câmp al clasei numit generatedClients.

Metoda run() este responsabilă cu simularea procesului și se execută într-un fir de execuție separat. În această metodă, se realizează următoarele operații:

- În fiecare iterație, se actualizează timpul curent și se scrie în fișierul de ieșire mesajul corespunzător.
- Se verifică dacă un client nou a sosit și, dacă este cazul, se adaugă la coada potrivită din cadrul obiectului scheduler.
- Se afișează în consolă lista de clienți așteptând și starea serverelor.
- Se verifică dacă toți clienții și-au terminat servirea și s-a ajuns la un punct în care nu mai există clienți în așteptare sau în deservire.
- Se pune firul de execuție în așteptare pentru o secundă.

Metoda printClients() este responsabilă cu afișarea listei de clienți aflați în așteptare și se folosește atât în interiorul metodei run(), cât și în metoda SimulationFrame.update() din clasa SimulationFrame pentru actualizarea interfeței grafice.

```
public SimulationManager(String fileName, String fileOutputName) {
    input = new Input(fileName, fileOutputName);
    generateRandomClients();
    System.out.println(generatedClients);
    scheduler = new Scheduler(input.getNrQueues(), input.getNrClients());
}
```


9. CLASA MAIN

Această clasă conține metoda principală main care creează un obiect SimulationManager cu fișierul de intrare "input1.txt" și fișierul de ieșire "output1.txt". Apoi, este creat un fir de execuție nou, trecând obiectul SimulationManager ca parametru și pornindu-se firul de execuție utilizând metoda start(). Acest lucru va declanșa simularea magazinului și va rula până când toți clienții au fost serviți și toate cozile sunt goale.

```
public static void main(String[] args) throws IOException, InterruptedException
{
    SimulationManager gen = new SimulationManager("input1.txt", "output1.txt");
    Thread t = new Thread(gen);
    t.start();
}
```

5. Rezultate

În final, pentru a testa cu ușurință codul meu, m-am folosit de datele de intrare cerute și am obținut rezultate corecte în ceea ce privește 'Log of events'-ul cerut. Se poate observa la final și average waiting time-ul, precum și câteva din cozile care vor apărea în fișierul de ieșire "output1.txt", atunci când rulam cu "input1.txt".

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 $t_{simulation}^{MAX} = 200 \text{ seconds}$ $[t_{arrival}^{MIN}, t_{arrival}^{MAX}] = [10, 100]$ $[t_{service}^{MIN}, t_{service}^{MAX}] = [3, 9]$

```
Time = 0
Waiting clients: (1, 22, 4)(2, 15, 4)(3, 5, 4)(4, 16, 3)

Queue: 1 closed
Queue: 2 closed

Time = 1
Waiting clients: (1, 22, 4)(2, 15, 4)(3, 5, 4)(4, 16, 3)

Queue: 1 closed
Queue: 2 closed

Time = 2
Waiting clients: (1, 22, 4)(2, 15, 4)(3, 5, 4)(4, 16, 3)

Queue: 1 closed
Queue: 2 closed

Time = 3
Waiting clients: (1, 22, 4)(2, 15, 4)(3, 5, 4)(4, 16, 3)

Queue: 1 closed
Queue: 2 closed

Time = 4
Waiting clients: (1, 22, 4)(2, 15, 4)(3, 5, 4)(4, 16, 3)

Queue: 1 closed
Queue: 2 closed

Time = 5
Waiting clients: (1, 22, 4)(2, 15, 4)(4, 16, 3)

Queue: 1 (3, 5, 4)
Queue: 2 closed

Time = 6
Waiting clients: (1, 22, 4)(2, 15, 4)(4, 16, 3)

Queue: 1 (3, 5, 3)
Queue: 2 closed
```

...

```
Time = 23
Waiting clients:

Queue: 1 (1, 22, 3)
Queue: 2 closed

Time = 24
Waiting clients:

Queue: 1 (1, 22, 2)
Queue: 2 closed

Time = 25
Waiting clients:

Queue: 1 (1, 22, 1)
Queue: 2 closed

Time = 26
Waiting clients:

Queue: 1 closed
Queue: 2 closed
Average waiting time: 3.75
```

6. Concluzii

În concluzie, acest proiect de simulare a unui sistem de cozi este un exemplu excelent de aplicare a conceptelor teoretice într-un context practic. Acesta ne demonstrează importanța analizei și optimizării sistemelor de cozi în diverse domenii, cum ar fi industria, afacerile sau chiar în viața de zi cu zi. Prin implementarea unui astfel de sistem, am putut explora diferite politici de selectare a cozilor, strategii de alocare a clienților și tehnici de simulare, în scopul de a îmbunătăți eficiența și performanța generală a sistemului.

De asemenea, acest proiect ne-a oferit oportunitatea de a dezvolta abilități practice în programarea orientată pe obiecte, gestionarea fișierelor, lucrul cu fire de execuție și integrarea diferitelor clase și module într-un proiect coerent și funcțional. Posibilitati de dezvoltari ulterioare ar putea fi: o eficiență mai mare a programului, o organizare mai bună a codului și mai ales o interfață grafică pe măsura, care să ușureze înțelegerea utilizatorului și să simuleze în timp real ceea ce acum se vede doar în consolă și în fișierul de ieșire.

7. Bibliografie

1. *Bruce Eckel, Thinking in Java (4th Edition), Publisher: Prentice Hall PTR Upper Saddle River, NJ United States, ISBN: 978-0-13-187248-6 Published: 01 December 2005.*
2. *What are Java classes? - www.tutorialspoint.com*
3. *[Java Threads - GeeksforGeeks](http://www.geeksforgeeks.org/java-threads/)*
4. *[Java Read Files \(w3schools.com\)](http://www.w3schools.com/java/java_read_files.asp)*
5. *[java - How do I create a file and write to it? - Stack Overflow](https://stackoverflow.com/questions/1038483/java-how-do-i-create-a-file-and-write-to-it)*
6. *https://dsrl.eu/courses/pt/materials/PT2023_A2_S1.pdf*