

DOCUMENTATIE

TEMA 2

**Aplicație de gestiune a cozilor cu ajutorul
thread-urilor si mecanismelor de sincronizare**

NUME STUDENT: Jișă Diana-Maria
GRUPA: 30224

CUPRINS

1. Obiectivul temei	3
2. Analiza problemei, modelare, scenarii, cazuri de utilizare.....	4
3. Proiectare	5
4. Implementare	7
5. Rezultate	11
6. Concluzii.....	13
7. Bibliografie.....	14

1. Obiectivul temei

Această temă de laborator urmărește crearea unei aplicații de gestionare eficientă a clienților, astfel încât aceștia să nu petreacă prea mult timp în așteptarea la cozi. În cadrul aplicației, utilizatorul poate configura diferite setări, cum ar fi durata simulării, numărul de clienți, numărul de cozi și intervalele de sosire și servire a clienților. Prin intermediul interfeței grafice a aplicației, utilizatorul poate introduce aceste informații și aplicația va monitoriza timpul total petrecut de fiecare client în coadă, calculând timpul mediu de așteptare. Scopul final al acestei teme este de a oferi o soluție pentru optimizarea sistemelor bazate pe cozi, prin reducerea timpului de așteptare al clienților.

Obiective secundare:

- Analiza problemei și identificarea cerințelor funcționale și non-funcționale ale aplicației de gestionare a cozilor (Cap. 2)
- Proiectarea soluției utilizând paradigma OOP, diagrame UML de clase și pachete, și definirea structurilor de date și a interfețelor necesare. (Cap. 3)
- Implementarea aplicației de gestionare a cozilor și a interfaței grafice a utilizatorului (Cap. 4)
- Prezentarea scenariilor de testare (Cap. 5)
- Concluzii și sugestii pentru îmbunătățiri ulterioare (Cap. 6)
- Referințe bibliografice (Cap. 7)

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

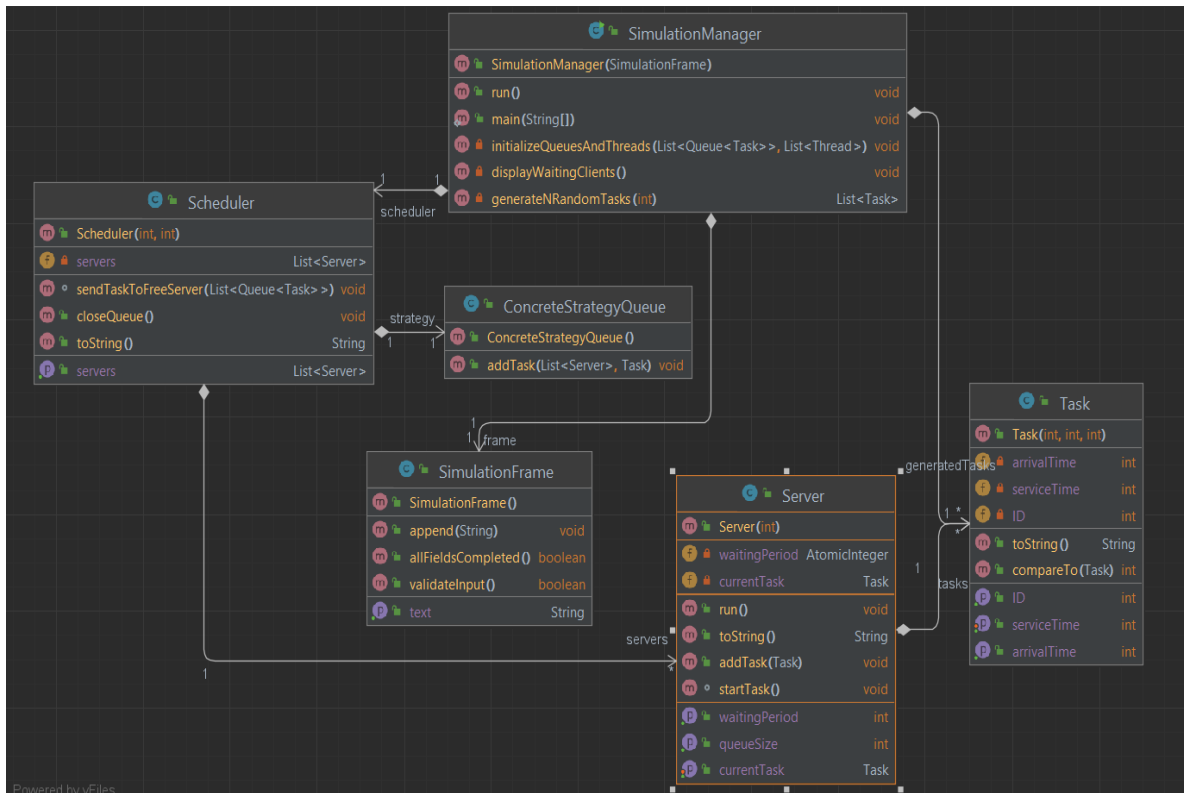
Cozile sunt folosite pentru a modela domeniul din lumea reală, oferind un loc pentru clienți, unde să aștepte înainte de a primi un serviciu. Obiectivul principal al unei cozi este de a minimiza timpul de așteptare al clienților înainte de a fi serviți. În aceste sisteme, fiecare client este asociat cu o casă de servire, iar clienții trebuie să aștepte înainte de a primi serviciul dorit. Pentru a minimiza timpul de așteptare, clienții sunt direcționați către casa de servire cu cel mai mic număr de clienți.

Scopul proiectului este de a simula sosirea unei serii de clienți într-un anumit interval de timp și direcționarea lor către casa de servire cu cel mai mic număr de clienți. Pentru a realiza acest lucru, trebuie să cunoaștem momentul de sosire al fiecărui client și timpul necesar pentru a primi serviciul dorit. Este important să cunoaștem și timpul de așteptare estimat pentru fiecare client nou sosit la fiecare casă de servire.

După ce utilizatorul accesează interfața programului, acesta va trebui să introducă anumiți parametri care vor influența modul în care se va desfășura simularea. Este important să se rețină că, în cazul în care se aleg valori mari pentru acești parametri, simularea va avea un timp de rulare mai îndelungat, deoarece implementarea nu este optimizată pentru performanță în astfel de situații. Cu toate acestea, programul poate fi utilizat cu succes în toate scenariile, iar utilizatorii sunt încurajați să își ajusteze parametrii în funcție de nevoile lor specifice.

3. Proiectare

Diagrama UML



În general, clasele din pachetul `org.example.Model` sunt folosite pentru a reprezenta obiectele și datele necesare pentru simulare, clasa `Scheduler` este folosită pentru a gestiona planificarea serverelor și coziilor, iar clasa `SimulationManager` este folosită pentru a inițializa și rula simularea sistemului de cozi. Clasa `SimulationFrame` este folosită pentru a afișa informații relevante în interfața grafică a simulării.

Relația dintre clasa `SimulationManager` și celelalte clase din pachetul `org.example.Model` este aceea că clasa `SimulationManager` folosește obiecte de tip `Task` și `Server` din acest pachet. În plus, clasa `Scheduler` din pachetul `org.example.BusinessLogic` este folosită de către clasa `SimulationManager` pentru a gestiona serverele și coziile din sistemul de cozi.

Relația dintre clasa `SimulationManager` și clasa `SimulationFrame` din pachetul `org.example.GUI` este aceea că clasa `SimulationManager` extinde clasa `JFrame`, deci reprezintă fereastra de simulare a sistemului de cozi. Aceasta primește un obiect de tip `SimulationFrame` în

constructor și utilizează variabilele din acesta pentru a prelua datele de configurare ale sistemului de cozi și pentru a afișa informații relevante în fereastra de simulare.

În ceea ce privește structurile de date folosite, clasa utilizează mai multe colecții pentru a stoca datele necesare pentru simulare:

- `List<Task> generatedTasks`: o listă de obiecte de tip `Task` ce reprezintă sarcinile generate random ce trebuie procesate de către sistemul de cozi.
- `List<Queue<Task>> queues`: o listă de cozi, fiecare coadă reprezentând coada de așteptare pentru un server din sistemul de cozi.
- `List<Thread> queueThreads`: o listă de fire de execuție, fiecare fir de execuție reprezentând un server din sistemul de cozi.

4. Implementare

Pentru a implementa această aplicație de administrare a cozilor, am implementat 6 clase, pe care le-am împărțit, în funcție de funcționalitatea lor, în 3 pachete distincte: **Model**, **BusinessLogic** și **GUI**.

Clasa Task

Aceasta este o clasă Java din pachetul " **package org.example.Model** ", care conține informațiile necesare pentru procesarea fiecărui client. Clasa Task are trei câmpuri: ID, arrivalTime și serviceTime. Aceste câmpuri reprezintă ID-ul unic al clientului, momentul la care acesta ajunge și respectiv timpul necesar pentru a fi servit. Clasa implementează interfața Comparable și suprascrive metoda compareTo pentru a compara clienții în funcție de momentul sosirii lor. Clasa oferă, de asemenea, metode de acces pentru câmpul serviceTime și o metodă toString pentru a afișa detaliile clientului.

Clasa Server

Aceasta este cealaltă clasă Java din pachetul " **package org.example.Model** ", care reprezintă un server(coadă) care poate procesa clienți. Clasa Server are trei câmpuri: tasks, waitingPeriod și currentTask. Câmpul tasks este o coadă blocantă de clienți care așteaptă să fie procesați de către server. Câmpul waitingPeriod reprezintă timpul total pe care clienții îl așteaptă în coadă, iar currentTask reprezintă clientul pe care serverul îl procesează în prezent.

Clasa oferă mai multe metode, inclusiv: addTask, startTask, getCurrentTask, setCurrentTask, getQueueSize, getWaitingPeriod. Totodată în interiorul acestei clase este implementată și metoda run, care este apelată atunci când firul de execuție al serverului este pornit. Aceasta preia continuu sarcini din coada tasks, le procesează și actualizează câmpul waitingPeriod corespunzător. Dacă coada este goală, firul de execuție al serverului așteaptă până când o nouă sarcină este adăugată în coadă. Odată ce o sarcină este finalizată, câmpul currentTask este setat la null.

Clasa Scheduler

Aceasta este o clasă Java din pachetul "**org.example.BusinessLogic**". Clasa este numită "Scheduler" și reprezintă o logică de gestionare a unui sistem de cozi.

Clasa conține o listă de servere și o listă de cozi asociate. De asemenea, are o strategie de coadă definită de clasa "ConcreteStrategyQueue" și un număr maxim de servere și clienți per server. Constructorul clasei initializează serverele și coziile, creând un fir de execuție pentru fiecare server și apoi le pornește. De asemenea, se setează strategia de coadă și numărul total de clienți serviți.

Metoda "sendTaskToFreeServer" trimite clienții către serverele libere. Verifică fiecare server și verifică dacă acesta are un client pe care îl procesează. Dacă nu are nicio client curent, îi alege un client din prima coadă nevidă găsită și o atribuie serverului. Dacă serverul are un client, verifică dacă aceasta a fost finalizată și, în caz afirmativ, o elimină și incrementă numărul total de clienți serviți. Metoda "closeQueue" elimină cozile goale din lista de cozi. Metoda "toString" returnează o reprezentare sub formă de șir de caractere a stării curente a cozilor și sarcinilor. Pentru fiecare coadă, se afișează numărul cozi, lista clienților și serverul asociat.

Clasa ConcreteStrategyQueue

Clasa ConcreteStrategyQueue se află în pachetul "**org.example.BusinessLogic**". Această clasă are o singură metoda, addTask, care primește lista de servere și un client, iar scopul ei este de a adauga clientul la coada cu cel mai mic număr de clienți.

```
2 usages
public class ConcreteStrategyQueue {

    public void addTask(List<Server> servers, Task t) {
        Server shortestQueueServer = servers.get(0);
        for (Server server : servers) {
            if (server.getQueueSize() < shortestQueueServer.getQueueSize()) {
                shortestQueueServer = server;
            }
        }
        shortestQueueServer.addTask(t);
    }
}
```


Clasa SimulationManager

Aceasta este o clasă Java din pachetul " **package org.example.BusinessLogic** ", care reprezintă o implementare a unui manager de simulare pentru un sistem de cozi. Sistemul constă dintr-un set de servere care procesează clienți care sosesc la intervale aleatoare de timp. Funcția principală a managerului este de a distribui clienții între servere, luând în considerare timpul de sosire și timpul de servire al fiecărei client, dar și de a colecta statistici cu privire la performanța sistemului.

Clasa SimulationManager extinde clasa JFrame și implementează interfața Runnable. Aceasta conține mai multe câmpuri, inclusiv obiectul SimulationFrame utilizat pentru a afișa simularea într-o interfață grafică, limita de timp pentru simulare și parametrii de controlare a timpului de sosire și a timpului de servire al clienților. Conține, de asemenea, un obiect Scheduler, care gestionează serverele și o Listă de obiecte Task care reprezintă clienții de procesat.

Constructorul SimulationManager inițializează câmpurile obiectului și creează serverele și firele necesare pentru a procesa sarcinile. De asemenea, generează o listă de obiecte Task cu timpi aleatorii de sosire și serviciu, sortează lista în funcție de timpul de sosire și timpul de serviciu al sarcinilor și afișează lista de clienți așteptând în fereastra de simulare.

Metoda initializeQueuesAndThreads() inițializează cozile și firele pentru fiecare server. Creează un obiect AtomicBoolean pentru a urmări dacă mai există clienți rămași, creează o coadă pentru fiecare server și creează un fir pentru a procesa clienții pentru fiecare server. Firele sunt în buclă continuă, verificând dacă există clienți în coadă și le atribuie serverului. Când nu mai sunt clienți în coadă de procesat și nu mai sunt clienți care urmează să sosescă, firele se termină. Metoda displayWaitingClients() afișează lista de clienți așteptând în fereastra de simulare.

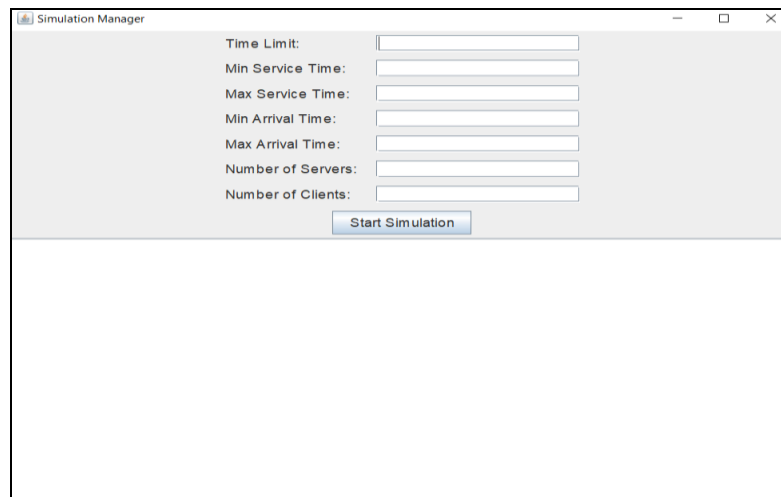
Metoda run() este bucla principală a simulării. Inițializează cozile și firele, afișează lista de clienți așteptând și rulează până când limita de timp este atinsă sau nu mai sunt clienți de procesat. În fiecare iterație a buclei, verifică clienții care au sosit și le atribuie serverelor, colectând statistici privind timpul de așteptare și timpul de serviciu al fiecărui client. De asemenea, colectează statistici privind numărul de clienți serviți și ora de vârf a simulării. În cele din urmă, scrie statisticile într-un fișier, dar le și afișează pe interfața grafică.

Clasa SimulationFrame

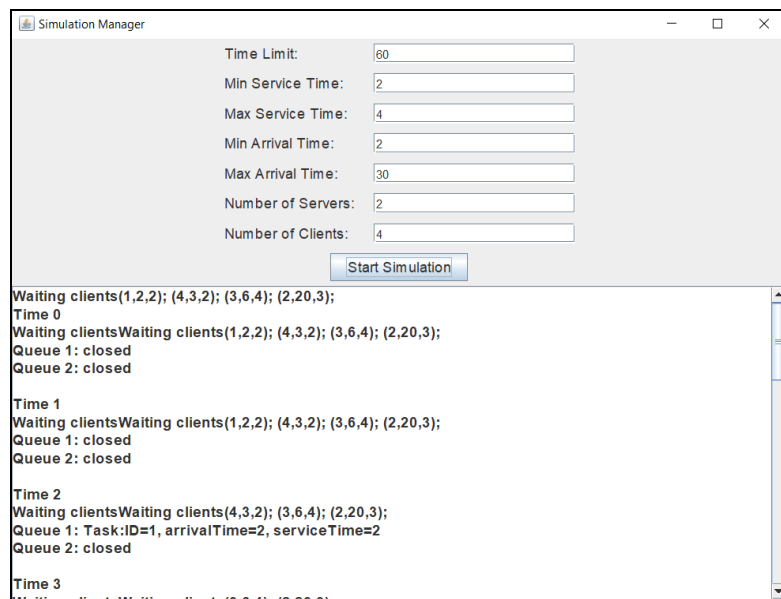
Aceasta este o clasă Java din pachetul " **package org.example.GUI** ". Această clasă extinde clasa "JFrame" din biblioteca Swing și reprezintă o interfață grafică cu utilizatorul (GUI) care permite utilizatorului să introducă diferiți parametri pentru o simulare a unui sistem de cozi și să înceapă simularea.

Interfața conține mai multe câmpuri pentru introducerea parametrilor, precum timpul limită, timpul maxim și minim de serviciu, timpul maxim și minim de sosire, precum și numărul de servere și clienți. De asemenea, există un buton "Start Simulation" care inițiază simularea atunci când este apăsat.

În clasa "SimulationFrame" este inclusă și o metodă numită "validateInput", care verifică dacă inputurile introduse de utilizator sunt valide. Aceasta verifică dacă toate câmpurile sunt completate și dacă valorile introduse sunt numere întregi pozitive.



The screenshot shows a window titled "Simulation Manager". It contains several input fields for simulation parameters: "Time Limit:", "Min Service Time:", "Max Service Time:", "Min Arrival Time:", "Max Arrival Time:", "Number of Servers:", and "Number of Clients:". Each field is followed by a text input box. Below these fields is a button labeled "Start Simulation".



The screenshot shows the same "Simulation Manager" window, but now the "Start Simulation" button has been pressed, and the output area below the input fields is populated with simulation results. The output is as follows:

```
Waiting clients(1,2,2); (4,3,2); (3,6,4); (2,20,3);
Time 0
Waiting clientsWaiting clients(1,2,2); (4,3,2); (3,6,4); (2,20,3);
Queue 1: closed
Queue 2: closed

Time 1
Waiting clientsWaiting clients(1,2,2); (4,3,2); (3,6,4); (2,20,3);
Queue 1: closed
Queue 2: closed

Time 2
Waiting clientsWaiting clients(4,3,2); (3,6,4); (2,20,3);
Queue 1: Task:ID=1, arrivalTime=2, serviceTime=2
Queue 2: closed

Time 3
```

5. Rezultate

Am efectuat teste multiple ale acestui program și am constatat că a furnizat rezultate corecte pentru fiecare dintre ele. Mai jos este prezentat rezultatul primului test, care este precizat și în cadrul cerinței. Rezultatele pentru celelalte două teste din cerință sunt disponibile într-un fișier, pe care îl voi încărca separat.

Test 1

N = 4

Q = 2

tsimulation MAX = 60 seconds

[tarrival MIN, tarrival MAX] = [2, 30]

[tservice MIN, tservice MAX]= [2, 4]

```
Waiting clients(1,6,3); (4,9,3); (2,10,2); (3,15,3);
Time 0
Waiting clients(1,6,3); (4,9,3); (2,10,2); (3,15,3);
Queue 1: closed
Queue 2: closed
```

```
Time 1
Waiting clients(1,6,3); (4,9,3); (2,10,2); (3,15,3);
Queue 1: closed
Queue 2: closed
```

```
Time 2
Waiting clients(1,6,3); (4,9,3); (2,10,2); (3,15,3);
Queue 1: closed
Queue 2: closed
```

```
Time 3
Waiting clients(1,6,3); (4,9,3); (2,10,2); (3,15,3);
Queue 1: closed
Queue 2: closed
```

```
Time 4
Waiting clients(1,6,3); (4,9,3); (2,10,2); (3,15,3);
Queue 1: closed
Queue 2: closed
```

```
Time 5
Waiting clients(1,6,3); (4,9,3); (2,10,2); (3,15,3);
Queue 1: closed
Queue 2: closed
```

```
Time 6
Waiting clients(4,9,3); (2,10,2); (3,15,3);
Queue 1: Task:ID=1, arrivalTime=6, serviceTime=3
Queue 2: closed
```

```
Time 7
Waiting clients(4,9,3); (2,10,2); (3,15,3);
Queue 1: Task:ID=1, arrivalTime=6, serviceTime=2
Queue 2: closed
```

```

Time 8
Waiting clients(4,9,3); (2,10,2); (3,15,3);
Queue 1: Task:ID=1, arrivalTime=6, serviceTime=1
Queue 2: closed

Time 9
Waiting clients(2,10,2); (3,15,3);
Queue 1: Task:ID=4, arrivalTime=9, serviceTime=3
Queue 2: closed

Time 10
Waiting clients(3,15,3);
Queue 1: Task:ID=4, arrivalTime=9, serviceTime=2
Queue 2: Task:ID=2, arrivalTime=10, serviceTime=2

Time 11
Waiting clients(3,15,3);
Queue 1: Task:ID=4, arrivalTime=9, serviceTime=1
Queue 2: Task:ID=2, arrivalTime=10, serviceTime=1

Time 12
Waiting clients(3,15,3);
Queue 1: closed
Queue 2: closed

Time 13
Waiting clients(3,15,3);
Queue 1: closed
Queue 2: closed

Time 14
Waiting clients(3,15,3);
Queue 1: closed
Queue 2: closed

Time 15
Waiting clients
Queue 1: Task:ID=3, arrivalTime=15, serviceTime=3
Queue 2: closed

Time 16
Waiting clients
Queue 1: Task:ID=3, arrivalTime=15, serviceTime=2
Queue 2: closed

Time 17
Waiting clients
Queue 1: Task:ID=3, arrivalTime=15, serviceTime=1
Queue 2: closed

Time 18
Waiting clients
Queue 1: closed
Queue 2: closed

Average waiting time: 2.7500
Average service time: 2.7500
Peak hour: 6

```

În urma testelor efectuate, am constatat ca aplicația de gestionare a cozilor funcționează corect. Prin urmare, testarea a fost esențială pentru dezvoltarea unui program fiabil. Aceasta a asigurat ca aplicația funcționează corect si că poate fi utilizată in mod eficient si intuitiv de către utilizatori.

6. Concluzii

În concluzie, crearea acestei aplicații de administrare a cozilor a fost o provocare captivantă din punct de vedere tehnic și o oportunitate de a învăța mai multe despre modelarea sistemelor bazate pe cozi și sincronizarea programării. Sistemul dezvoltat poate simula cu succes un număr mare de clienți care așteaptă în cozi, iar timpul mediu de așteptare pentru fiecare client a fost evaluat cu succes.

În viitor, există posibilitatea de a îmbunătăți acest proiect prin implementarea de noi funcționalități, cum ar fi prioritizarea clienților sau utilizarea unor algoritmi mai eficienți pentru alocarea clienților la cozi. În plus, interfața grafică poate fi îmbunătățită prin adăugarea de elemente vizuale interactive, care să ilustreze mai bine funcționarea algoritmului și să ofere o experiență de utilizare mai intuitivă și mai plăcută.

În general, acest proiect mi-a permis să îmi dezvolt abilitățile tehnice și cunoștințele despre modelarea sistemelor bazate pe cozi și sincronizare în programare, precum și să obțin o experiență practică în proiectarea și dezvoltarea de aplicații pentru gestionarea cozilor.

7. Bibliografie

1. What is Thread-Safety and How to Achieve it? (<https://www.baeldung.com/java-thread-safety>)
2. Thread Safety in Java (<https://www.digitalocean.com/community/tutorials/thread-safety-in-java>)
3. <https://www.youtube.com/watch?v=px4W-HXRWkK>
4. Queue Interface in Java (<https://www.geeksforgeeks.org/queue-interface-java/>)
5. Java Programming Tutorial Multithreading & Concurrent Programming (https://www3.ntu.edu.sg/home/ehchua/programming/java/j5e_multithreading.html)