DOCUMENTAȚIE

TEMA 2

NUME STUDENT: IFTENE IOAN-FLORIN

GRUPA: 30221

# CUPRINS

[1. Obiectivul temei 3](#_Toc95297885)

[2. Analiza problemei, modelare, scenarii, cazuri de utilizare 3](#_Toc95297886)

[3. Proiectare 4](#_Toc95297887)

[4. Implementare 4](#_Toc95297888)

[5. Rezultate 6](#_Toc95297889)

[6. Concluzii 7](#_Toc95297890)

[7. Bibliografie 7](#_Toc95297891)

# Obiectivul temei

Obiectivul principal al temei este de a proiecta și implementa o aplicație Java care să simuleze un sistem de gestiune al cozilor de clienți, astfel încât timpul de așteptare al fiecărui client să fie minim. Clienții ar trebui așezați automat la cozile la care au de așteptat cel mai puțin, înainte de a fi serviți.

Obiectivele secundare care vor ajuta la atingera obiectivului principal sunt:

|  |  |
| --- | --- |
| **Obiectiv secundar** | **Capitolul în care e detaliat** |
| Generarea aleatoare a clienților | 4 |
| Multithreading: un thread pentru fiecare coadă + folosirea de structuri de date care să asigure o bună funcționalitate a threadurilor | 4 |
| Generarea unui fișier text, în care să apară succesiunea de evenimente petrecute în cadrul cozilor, la fiecare moment de timp | 4 |
| Crearea unei interfeței grafice formată din două ferestre, una în care utilizatorul să poată seta detaliile simulării(timpul maxim de simulare, timpii minimi și maximi la care pot ajunge clienții și timpii minimi și maximi de procesare/ de servire a clienților, numărul de cozi și numărul de clienți) și cealaltă fereastră să fie folosită pentru a afișa evoluția în timp real a cozilor de clienți | 3,4 |
| Afișarea rezultatelor simulării (timpul mediu de așteptare, timpul mediu de servire și ora de vârf pentru intervalul de timp în care a avut loc simularea) atât în interfața grafică, cât și în fișierul text creat pentru succesiunea evenimentelor | 4 |
| Testarea aplicației pentru setul de teste din tabelul de mai jos | 5 |

Tabel de teste:

|  |  |  |
| --- | --- | --- |
| Testul 1 | Testul 2 | Testul 3 |
| * N = 4 * Q = 2 * timpul maxim de simulare = 60 de secunde * intervalul de timp în care pot ajunge clienții = [2, 30] * intervalul de procesare pentru un client = [2, 4] | * N = 50 * Q = 5 * timpul maxim de simulare = 60 de secunde * intervalul de timp în care pot ajunge clienții = [2, 40] * intervalul de procesare pentru un client = [1, 7] | * N = 1000 * Q = 20 * timpul maxim de simulare = 200 de secunde * intervalul de timp în care pot ajunge clienții = [10, 100] * intervalul de procesare pentru un client = [3, 9] |

# Analiza problemei, modelare, scenarii, cazuri de utilizare

Pentru a folosi aplicația, utilizatorul va introduce mai întâi cele 7 informații referitoare la detaliile simulării (timpul maxim de simulare, timpii minimi și maximi la care pot ajunge clienții și timpii minimi și maximi de procesare/ de servire a clienților, numărul de cozi și numărul de clienți) în fereastra de Setup. Ele vor fi preluate ca și stringuri de către aplicație, la apăsarea butonului *Start,* vor fi verificate și, dacă totul e bine, se va deschide fereastra corespunzătoare simularii bazate pe datele introduse. Dacă datele introduse nu sunt corecte, se va afișa unul din mesajele "Verificati datele introduse! Asigurati-vă că valorile introduse sunt numere întregi, pozitive!", "Valorile introduse trebuie sa fie numere intregi, pozitive!", "Timpii minimi trebuie sa fie mai mici decat timpii maximi!", în funcție de cazul de eroare.

* Scenariul principal, în care aplicația efectuează cu succes ceea ce se dorește este următorul:

Pasul 1: Utilizatorul deschide aplicația

Pasul 2: Utilizatorul introduce corect datele simulării în fereastra *SimulationSetup*

Pasul 3: Utilizatorul apasă butonul *Start*

Pasul 4: Aplicația reușește să preia cele 7 stringuri din câmpurile de intrare și să le atribuie cîmpurilor corespunzătoare unor valori întregi din *SimulationManager*

Pasul 5: Se deschide fereastra *SimulationFrame*

Pasul 6: Simularea începe și utilizatorul poate vedea în timp real evoluția cozilor, până când timpul curent atinge timpul limită, sau până când nu mai rămân clienți de gestionat

* Scenarii alternative pot apărea în următoarele situații:

1. Utilizatorul introduce altceva decât nuimere întregi în oricare din cele 7 câmpuri din *SimulationSetup*. În acest caz:
2. utilizatorului îi va apărea o fereastră cu mesajul "Verificati datele introduse! Asigurati-vă că valorile introduse sunt numere întregi, pozitive!";
3. se revine la pasul 2 din scenariul principal.
4. Utilizatorul introduce un număr negativ în cel puțin unul din cele 7 câmpuri sau scrie 0 în dreptul valorii corespunzătoare numărului de cozi. În această situație:
5. utilizatorului îi va apărea o fereastră cu mesajul "Valorile introduse trebuie sa fie numere intregi, pozitive!";
6. se revine la pasul 2 din scenariul principal.
7. Utilizatorul introduce o valoare mai mare într-un câmp destinat unui timp minim decât în câmpul pentru timpul maxim corespunzător. În această situație:
8. utilizatorului îi va apărea o fereastră cu mesajul "Timpii minimi trebuie sa fie mai mici decat timpii maximi!";
9. se revine la pasul 2 din scenariul principal.

# Proiectare

Diagrama UML de clase:

----------------------de inserat diagrama-----------------------------

Am început proiectarea prin crearea clasei *Client*, care reprezintă obiectul cu care operează aplicația. Am conceput apoi clasa *Server*, care reprezintă obiectul corepunzător unei cozi, fiind alcătuită dintr-o listă de clienți. A treia clasă creată este clasa *Scheduler*. Ea constă dintr-o listă formată din toate cozile simulării.

Clasele *SimulationSetup* și *SimulationFrame* reprezintă clasele care alcătuiesc interfața grafică a aplicației. *SimulationSetup* este fereastra cu care se deschide aplicația, în care utilizatorul introduce datele și pornește simularea, iar *SimulationFrame* este fereastra în care se poate vedea evoluția cozilor, în timp real și rezultatele obținute în urma simulării.

Clasa SimulationManager este clasa principală a aplicației. Din această clasă se generează aleator N clienți și se efectuează simularea propriu-zisă, cu actualizarea informației afișate în interfața grafică.

# Implementare

Clasele folosite pentru realizarea aplicației și principalele metode din fiecare sunt următoarele:

1. Clasa Client

Această clasă este alcătuită din 3 câmpuri pentru a reține principalele caracteristici ale unui obiect de tip client: id-ul (id) , timpul de sosire (tArrival) și timpul de procesare/ de servire (tService), toate cele 3 fiind variabile de tip int.

Clasa conține metoda *toString*, care returnează o variabilă de tip String, folosită ulterior pentru a afișa detaliile referitoare la clienți. În plus, această clasă conține settere și gettere pentru toate cele 3 câmpuri care o descriu.

Totodată, clasa implementează *Comparable*, metoda *compareTo* fiind implementată astfel încât, atunci când este apelată metoda *Collections.sort()* pentru o listă de clienți, aceasta să fie sortată în ordinea crescătoare a timpilor de sosire ai clienților din listă.

1. Clasa Server

Această clasă este alcătuită din 2 câmpuri: o coadă de clienți, de tip BlockingQueue și un timp de așteptare la coadă, de tip AtomicInteger. Am folosit aceste tipuri de date pentru a evita probleme legate de sincronizare între threaduri. În această clasă am creat metoda *addClient*, prin care se adaugă un nou client în coadă și se actualizează timpul de așteptare. O altă metodă a clasei este metoda *toString,* folosită pentru a putea afișa coada și conținutul ei.

Clasa implementează Runnable, metoda run având rolul de a actualiza timpul de servire al clientului care este la rând în fiecare secundă care trece sau de a scoate clientul curent din coadă dacă procesarea lui a fost finalizată (timpul de servire a ajuns la 0).

1. Clasa Scheduler

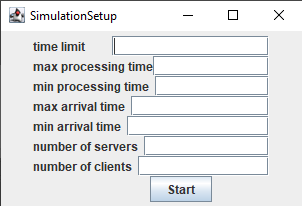
Este alcătuită dintr-o listă de cozi (*servers*) și o listă de Threaduri (*threads)* în care se vor reține threadurile corespunzătoare cozilor din listă. În constructorul clasei, care primește ca și parametru numărul cozilor, se creează atâtea cozi câte indică valoarea primită, se introduc în lista *servers* și se creează și pornește câte un thread pentru fiecare coadă, thread care va fi adăugat în lista *threads.*

Clasa conține metoda *getMinWaitT*, care returnează coada cu cel mai mic timp de așteptare. Această metodă va fi folosită în *SimulationManager* pentru a decide la ce coadă se va adăuga noul client. Alte metode importante ale clasei sunt *checkIfThereAreClients* - care verifică dacă există clienți în cozi, *unservedClients* - care returnează câți clienți nu au ajuns încă la rând pentru a se începe procesul de servire a lor, *howManyClients* -care returnează numărul de clienți din cozi, metodă ce va fi folosită pentru a determina ora de vârf. Totodată, această clasă conține și metoda *interruptThreads*, care parcurge lista de Threaduri pornite pentru cozi și le întrerupe – această metodă fiind apeată la finalizarea simulării.

1. Clasa SimulationSetup

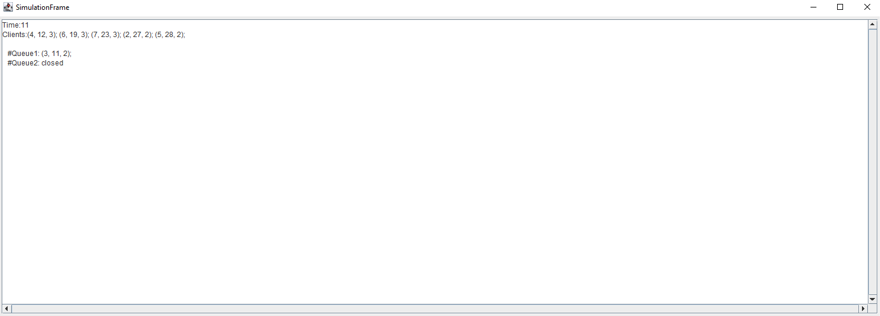
În această clasă este implementată fereastra cu care se dechide aplicația, cea în care utilizatorul introduce datele premergătoare simulării. Este alcătuită din 7 casete text de tip JTextField, câte una pentru fiecare atribut pe care utilizatorul trebuie să îl introducă, și un buton *start* de tip JButton, pe care utilizatorul va apăsa pentru a transmite aplicației că poate colecta datele introduse și că dorește să se înceapă simularea. Totodată, parte din atributele clasei sunt și o variabilă de tip *SimulationManager* și una de tip *SimulationFrame,* a căror utilitate va fi descrisă ulterior.

În cadrul constructorului clasei, sunt așezate casetele de text, butonul și 7 JLabel (unul pentru fiecare casetă-text) în interiorul unui JPanel, după cum se poate observa mai jos:



Clasa conține metoda *showSim*, prin care se ascunde fereastra curentă, se pornește un Thread corespunzător clasei *SimulationManager* și se face vizibilă fereastra *SimulationFrame*, cea în care se va putea urmări simularea.

1. Clasa SimulationFrame

În această clasă este implementată fereastra în care se vor afișa simularea în timp real și rezultatele obțiute în urma ei. Este alcătuită dintr-un JTextArea, în cadrul căruia se vor scrie datele. Constructorul clasei pune cîmpul JTextArea într-un JScrollPane, pentru ca utilizatorul să poată derula dacă sunt prea multe informații pentru a putea fi văzute în fereastra care se deschide. JScrollPane-ul este apoi adăugat într-un JPanel, care va fi văzut de către utilizator. Metoda clasei, *setVwText*, este folosită pentru a schimba textul afișat utilizatorului în JTextArea.

1. Clasa SimulationManager

Câmpurile clasei sunt reprezentate de cele 7 atribute ale simulării - timpul maxim de simulare (*timeLimit*), timpul maxim de procesare al unui client (*maxProcessingTime*), timpul minim de procesare (*minProcessingTime*), timpul maxim la care un client poate veni la coadă (*maxArrivalTime*), timpul minim la care se poate veni la coadă (*minArrivalTime*), numărul de cozi (*numberofServers*), numărul de clienți (*numberOfClients*); o varibilă de tip *Scheduler –* în care se vor salva cozile; una de tip *SimulationSetup* – pentru a face vizibilă prima fereastră a aplicației atunci când se instanțiază un obiect de tip *SimulationManager;* și o listă de clienți formată din obiectele de tip *Client* generate aleator în cadrul metodei *generateNRandom* (metoda folosește *Math.random()*)*,* din interiorul clasei *SimulationManager*.

În cadrul acestei clase este implementat un ActionListener pentru apăsarea butonului *start* din interfața *SimulationSetup.* În cadrul metodei *actionPerformed* sunt tratate scenariile alternative descrise în capitolul 2 referitoare la apăsarea butonului *start*.

Clasa implementează *Runnable*.În cadrul metodei *run* se creează (sau se golește) fișierul text în care se vor scrie evenimentele și rezultatele simulării, se inițializează o variabilă care va contoriza timpul (*currentTime*) și în interiorul unei instrucțiuni de tip *while* se iau clienții generați aleator din *generatedClients* care au timpul de sosire egal cu timpul curent, se caută coada din *scheduler* cu cel mai mic timp de așteptare pentru fiecare client, se adaugă clientul la acea coadă, se modifică variabilele folosite pentru calcularea timpului mediu de așteptare și de servire, se verifică dacă timpul curent reprezintă cumva ”ora de vârf” și apoi se scrie conținutul curent al cozilor, lista de clienți în așteptare și timpul curent atât în fișierul text creat, cât și în interfața grafică. Se așteaptă trecerea unei secunde cu ajutorul metodei Thread.sleep(1000), se incrementează timpul curent și se repetă totți pașii anteriori, până când timpul curent este mai mic decât timpul maxim de simulare (*timeLimit*) sau până când nu mai sunt clienți nici la cozi, nici în lista de așteptare. La ieșirea din *while,* se scriu în fișier și în interfață rezultatele obținute (timpul mediu de așteptare, timpul mediu de servire și ora de vârf) și se intrerup thread-urile aflate în stare activă..

# Rezultate

Aplicația a fost testată folosind datele prezente în al doilea tabel din Capitolul 1. O serie de rezultate obținute în urma celor 3 teste este prezentată în tabelul de mai jos. Fișierele text cu succesiunea de evenimente pentru fiecare din cele 3 teste pot fi găsite în cadrul proiectului, în aceeași locație în care se află și prezenta documentație.

|  |  |  |
| --- | --- | --- |
| Testul 1 | Testul 2 | Testul 3 |
| Average waiting time: 3.25  Average service time: 3.25  Peak hour: 16 --> 2 clients | Average waiting time: 7.32  Average service time: 4.22  Peak hour: 18 --> 13 clients | Average waiting time: 88.024  Average service time: 3.8685  Peak hour: 100 --> 716 clients |

# Concluzii

# Bibliografie

1. <https://dsrl.eu/courses/pt/materials/A2_Support_Presentation.pdf>
2. <https://howtodoinjava.com/java/io/java-append-to-file/>
3. <https://www.tutorialspoint.com/Java-Program-to-Append-Text-to-an-Existing-File>