Tema numărul 2

Aplicație de gestiune a cozilor folosind thread-uri

A picture containing logo, graphics, symbol, design

Description automatically generated

Nume: Șerban

Prenume: Sebastian – Mihai

Grupa: 30223

Specializare: Calculatoare și Tehnologia Informației

Contents

[1. Obiectivul temei 3](#_Toc136433826)

[2. Analiza problemei și Modelare 4](#_Toc136433827)

[3. Scenarii și cazuri de utilizare 5](#_Toc136433828)

[4. Implementare 6](#_Toc136433829)

[5. Concluzii 8](#_Toc136433830)

[6. Bibliografie 9](#_Toc136433831)

# Obiectivul temei

Principalul obiectiv al temei este acela de a implementa o aplicație de gestiune a cozilor dintr-un magazin folosind fire de execuție și mecanisme de sincronizare. Această aplicație trebuie să fie capabilă să execute următoarele operații:

* Să lanseze în execuție, în paralel, mai multe fire de execuție care să preia funcționalitățile cozilor.
* Să genereze un set de date aleator pe baza unor parametrii introduși în interfața grafică.
* Să genereze un jurnal ce urmărește evoluția cozilor în fiecare moment al simulării.
* Să calculeze timpul mediu de așteptare în coadă, timpul mediu de servire și moment cel mai aglomerat al simulării.
* Să update-ze în timp constant evoluția cozilor în interfața grafică

# Analiza problemei și Modelare

În faza incipientă, am analizat problema și am identificat cerințele funcționale pe care le-am enumerat în capitolul precedent.

De asemenea, am identificat și alte cerințe precum nevoia de o interfață grafică ce este capabilă să primească toți parametrii, să creez clase care să încapsuleze cozile și clienții într-un mod potrivit, să generez atât fișiere de tip text pentru afișarea evoluției cozilor pe tot parcursul simulării.

Totodată, am descoperit nevoia de clasificare a atributelor din fiecare clasă, în atribute ce necesită atenție mai specială, deoarece sunt utilizate în multithreading și atribute ce pot fi predefinite cu tipuri de dată primitive, ce nu necesită o atenție deosebită.

Necesitatea unor clase care au drept scop stocarea datelor despre clienți și a cozilor a dus la apariția a doua clase separate și anume: clasa Client care încapsulează toate informațiile legate de clienți și clasa Server care încapsulează toate informațiile și logica ce se pune în legătură cu cozile.

În urma celor două observații mai sus menționate am observat următorul lucru: trebuie să utilizez structuri de date ce suportă acțiuni concurente asupra lor pentru a putea definii cozile de aștepare, iar timpul inter de așteptare al fiecărei cozi este absolut necesar să fie de tip AtomiInteger pentru a asigura incrementarea și decrementarea acestuia conform logicii pe care dorim să o urmărim și anume fiecare fir de execuție să update-ze corect și persistent variabila ce stochează date despre timpul de așteptare.

O altă necesitate a fost stabilirea unei strategii de alegere a inserării unui client într-o coadă, iar pentru acest lucru am ales să folosesc o strategie ce se bazează pe o tehnică greedy, fiecare client fiind adăugat în coada care are timpul de așteptare cel mai mic la momentul de timp t în care doresc să realizez inserarea.

A screenshot of a computer

Description automatically generated with medium confidenceÎn urma cerințelor funcționale și a tuturor observațiilor mai sus menționate am proiectat un număr de 3 pachete și 6 clase și a rezultat următoarea diagramă de clase:

# Scenarii și cazuri de utilizare

Pentru această aplicație am identificat 3 scenarii care se pot întâlnii:

* Introducerea greșită a parametrilor de la tastatură sau neintroducerea lor deloc.
* Lansarea simulării cu parametrii corespunzători.
* Lansarea unei noi simulări în urma unei alte simulări încheiate cu succes.

În urma acestor scenarii am realizat următoarele cazuri de utilizare:

* În cazul în care nu se introduc datele corect din aplicație, simularea nu pornește, iar pe ecran se afișează mesajul: „Invalid Input”.
* În cazul în care se lansează simularea pentru prima dată, cu parametrii buni, se scrie evoluția în acel panel de tip JScrollPanel pentru a se putea observa în timp real evoluția cozilor.
* În cazul în care se lansează o nouă simulare, iar aceasta are parametrii corespunzători, se golește panel-ul JScrollPanel și se scrie, în timp real, evoluția noilor cozi.

# Implementare

Pachetul GUI:

Conține o singură clasă, și anume clasa SimulationFrame care construiește interfața grafică a aplicației.

Pachetul Model:

Conține clasele Client și Server care reprezintă clasele ce stochează date despre clienți și cozi.

Clasa Server implementează interfața Runnable pentru a descrie funcționalitatea firelor de execuție în raport cu cozile.

Public void run()

Această metodă descrie funcționalitatea thread-urilor: oprește thread-ul pentru 1 secundă, dacă coada nu este goală, se decrementează timpul de servire a clientului ce este servit în momentul t. Dacă servirea clientului s-a încheiat, elimină clientul din coadă.

public synchronized void addClient(Client client)

Această metodă asigură adăugarea unui client în coadă, fără a se suprapune cu alte fire de execuție.

Pachetul BLL:

Conține toată logica de execuție a aplicației, prin intermediul claselor SimManager, Scheduler, Strategy.

Clasa Strategy se ocupă cu alegerea cozii în care se introduce un client, pe baza metodei addClient.

public void addClient(List<Server> servers, Client client)

Aplică un algoritm de tip greedy ce determină coada în care clientul transmis prin parametrul client este introdus. Această strategie se bazează pe selectarea cozii cu timp minim de așteptare, iar dacă există două cozi cu același timp minim, o alege pe prima pe care o întâlnește.

Clasa Scheduler se ocupă cu încapsularea datelor din clasele Client, Server și Strategy pentru a putea fi utilizate mai cu ușurință.

Clasa SimManager se ocupă cu thread-ul principal ce lansează în execuție simularea și păstrează timpul global de simulare al aplicației.

public SimManager(SimulationFrame frame)

Acest constructor primește ca parametrul fereastra aplicației cu interfața grafică și se asigură că datele introduse sunt corecte pentru a se putea lansa în execuție simularea.

public void run()

Metodă ce are ca scop gestionarea tuturor cozilor prin intermediul scheduler-ului pe care l-am descris mai sus.

private void generate()

Metodă ce se ocupă cu generarea tuturor clienților folosind parametrii introduși de către utilizator în interfața grafică. Datele sunt generate aleator.

private void computeAverageWaitingTime()

Metodă ce se ocupă cu calculul timpului mediu de așteptare pentru ca acesta să poată fi afișat la finalul.

# Concluzii

Acum că am ajuns la finalul proiectului aș putea spune că sunt mulțumit de modul în care am livrat aplicația și sunt foarte bucuros de faptul că am reușit să lucrez cu fire de execuție și că am învățat atât de multe lucruri despre lacăte, variabile volatie, atomice, colecții sincronizate și colecții ce suportă operații concurente.

Ca și îmbunătățire, consider că aș mai putea lucra la calitatea diagramelor UML pentru a fi și mai explicite decât cele pe care le-am introdus până în prezent în documentațiile de la disciplina Tehnici de Programare.

De asemenea, dacă ar fi să mă reîntorc la implementarea acestei aplicații aș spune că un lucru de îmbunătățit ar fi crearea de cazuri mai speciale pentru validarea datelor, poate chiar introducerea de validatori.

# Bibliografie

[Fundamental Programming Techniques (dsrl.eu)](https://dsrl.eu/courses/pt/)

<https://www.geeksforgeeks.org/blockingqueue-interface-in-java/>

<https://www.w3schools.com/java/java_files_create.asp>

<https://www.geeksforgeeks.org/collections-synchronizedcollection-method-in-java-with-examples/>

<https://www.baeldung.com/java-blocking-queue>

<https://www.w3schools.com/java/java_threads.asp>

<https://www.digitalocean.com/community/tutorials/java-blockingqueue-example>

<https://www.digitalocean.com/community/tutorials/thread-sleep-java>