DOCUMENTATIE

TEMA *2*

NUME STUDENT: Alixandrescu Octavian

GRUPA: 30223

# 

# CUPRINS

[1. Obiectivul temei 3](#_Toc95297885)

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

[3. Proiectare 4](#_Toc95297887)

[4. Implementare 5](#_Toc95297888)

[5. Testare 12](#_Toc95297888)

[6. Concluzii 13](#_Toc95297890)

[7. Bibliografie 13](#_Toc95297891)

# Obiectivul temei

Proiectarea și implementarea unei aplicații de simulare a unui sistem de cozi bazat pe determinarea si minimizarea timpului de așteptare al clienților

Cozile sunt utilizate în mod obișnuit pentru a modela domeniile lumii reale. Obiectivul principal al unei cozi este de a oferi un loc pentru ca un „client” să aștepte înainte de a primi un „serviciu”. Managementul sistemelor bazate pe cozi este proiectat să minimizeze timpul pe care „clienții” lor îl așteaptă în cozi înainte de a fi serviți. O modalitate de a minimiza timpul de așteptare este adăugarea mai multor servere, adică mai multe cozi în sistem (fiecare coadă este considerată ca având un procesor asociat), dar această abordare crește costurile furnizorului de servicii.

Aplicația ar trebui să simuleze (prin definirea unui timp de simulare) o serie de N clienți care sosesc pentru service, care intră în Q cozi, așteaptă, sunt serviți și în final părăsesc cozile. Toți clienții sunt generați la pornirea simularii și sunt caracterizați de trei parametri: ID (un număr între 1 și N), (timpul de simulare când sunt gata să meargă la coadă; adică ora la care clientul a terminat cumpărăturile) și ( 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 în cozi și calculează timpul mediu de așteptare. Fiecare client este adăugat la coadă cu un timp de așteptare minim atunci când timpul său este mai mare sau egal cu timpul de simulare ( ≥ ).

Următoarele date ar trebui considerate date de intrare pentru aplicație și ar trebui să fie inserate de utilizator în interfața de utilizator a aplicației:

- Numar de clienti (N);

- Numărul de cozi (Q);

- Interval de simulare ();

- Ora minimă și maximă de sosire (≤≤);

- Durata minimă și maximă de service (≤≤);

Un număr de threaduri Q vor fi lansate pentru a procesa în paralel clienții. Un alt thread va fi lansat pentru a menține timpul de simulare și pentru a distribui fiecare client la coada cu cel mai mic timp de așteptare când ≥.

# Analiza problemei

# În programarea concomitentă, există două unități de bază de execuție: procese și fire. Pe un computer rulează mai multe fire și procese active, chiar dacă există doar un procesor cu un singur nucleu. Acest lucru se realizează prin caracteristica sistemului de operare de time-slicing. În Java, programarea concomitentă se ocupă de fire.

Pentru a defini un thread, trebuie să extindem Thread și să implementăm interfața Runnable. Fiecare thread suprascrie metoda run() => aceasta va fi executată în timpul ciclului de viață al firului. Atomicitatea va fi necesară pentru anumite variabile care ar putea fi accesate simultan de mai multe fire, asigurând astfel siguranța firului. În același scop, va trebui să folosim colecții sigure pentru fire cum ar fi colecții sincronizate (exemplu: Collections.synchronizedList(new ArrayList<>())) sau colecții concurente (exemplu: BlockingQueue).

Având în vedere faptul că vom avea nevoie și de o interfață grafică pentru a afișa distribuția clienților în timp real, vom folosi modelul de design Model-View-Controller.

# Proiectare

Diagrama UML de clase a proiectului:

A picture containing diagram

Description automatically generated

Se observă că structura proiectului nu este una foarte complexă. Proiectul conține opt clase, fiind structurat în patru pachete după cum urmează: pachetul View: clasa SetupView și clasa SimulationView; pachetul Model: clasa Scheduler, clasa Task și clasa Server; pachetul Controller: clasa Controller și clasa Simulator

# Implementare

**4.1 Pachetul ”Model”**

Acest pachet conține următoarele clase:

* Task
* Server
* Scheduler

Text

Description automatically generated

Clasa „ Task ” reprezintă clientul, care se caracterizează prin id, ora de sosire (arrivalT) și timpul necesar sarcinii pentru a fi procesată (serviceT). De asemenea, folosim o variabilă statică pentru numărul de clienți și o incrementăm de fiecare dată când este generat un alt client, pentru a obține id-ul. Metodele cheie aici sunt metoda toString() pe care o înlocuim, metoda compareTo(), cu care vom sorta mai târziu listele de clienți după ora sosirii. Nu setam id-ul clientilor in constructor, ci doar orele de service si sosire (cu valori aleatorii in intervalul dat), deoarece vom seta id-urile dupa ce vom genera si sorta listele de clienti.

Text

Description automatically generated

Clasa „ Server ” reprezintă coada, iar atributele acesteia sunt o coadă de blocare pentru sarcinile aflate în prezent pe acel server, un număr întreg atomic folosit pentru a stoca suma timpilor de serviciu pentru toți clienții aflați în prezent în coadă și un id pe care îl setăm cu ajutorul a unei alte variabile statice, la fel ca în clasa Task. Constructorul setează id-ul cozii, inițializează „ tasks ” ca un „ ArrayBlockingQueue<>() ” a cărui dimensiune este parametrul constructorului și este numărul maxim de clienți care pot fi în server în orice moment. De asemenea, inițializam numărul atomic „ waitingPeriod ” la 0. Metodele pe care le folosim aici sunt „ addTask(Task newTask) ” în care pur și simplu adăugăm o sarcină la coada de blocare și incrementăm perioada de așteptare cu timpul de serviciu al noului adăugat task, „ toString() ” în care afișăm toate sarcinile aflate în prezent pe server, „ getWaitingPeriod() ” și metoda „ run() ”, deoarece această clasă implementează interfața „ Runnable ”.

Text

Description automatically generated

În metoda „run()” folosim o buclă while care se execută constant și în această buclă oprim firul de execuție pentru un număr de secunde egal cu timpul de service al primului client din coadă și apoi scoatem acel client din coadă. , așa cum a fost procesat în acel timp. De asemenea, scădem perioada de așteptare a serverului cu timpul de service al clientului care a fost procesat. Întreaga bucată de cod necesită un bloc try-catch, deoarece metoda „sleep()” poate arunca o excepție de tipul „InterruptedException”.

Text

Description automatically generated

Clasa „Scheduler” este folosită pentru a distribui clienții la cozi. Astfel avem nevoie de o listă cu toate serverele și de un număr întreg care să stocheze numărul maxim de sarcini pe server, pe care le vom trimite ca parametru constructorului fiecărui server;

În constructor inițializam lista ca o listă de matrice a cărei dimensiune este parametrul acestui constructor (“nrServers”) și apoi inițializam serverele și creăm fire de execuție pentru fiecare dintre ele, pe care le pornim și noi.

Text

Description automatically generated

Metoda „dispatchTask()” găsește perioada minimă de așteptare de la toate serverele, astfel încât să știm care coadă va fi prima care se va goli. Aceasta este coada unde trimitem noua sarcină. De asemenea, dorim ca această metodă să returneze timpul de așteptare al clientului nou adăugat, deoarece mai târziu vom dori să calculăm timpul mediu de așteptare după întreaga simulare.

**3.2 Pachetul ”View”**

Acest pachet conține următoarele clase:

- SetupView;

- SimulationView.

Interfața este utilizată astfel încât utilizatorul să poată introduce toți parametrii simulării:

- numărul de cozi;

- numarul de clienti;

- durata simulării;

- ora minimă de sosire pentru toți clienții;

- ora maximă de sosire pentru toți clienții;

- timpul minim de servire pentru toti clientii;

-timpul maxim de servire pentru toti clientii.

Această clasă extinde clasa „JFrame”, iar metodele acesteia sunt doar constructorul, ascultătorul pentru butonul „RUN SIMULATION” și getters pentru toate șirurile din câmpurile de text.

Clasa SimulationView este pentru cadrul pe care dorim să apară după ce este apăsat butonul „RUN SIMULATION”. Atributele acelei clase sunt două matrice de etichete, una pentru servere și una pentru clienții în așteptare, o etichetă pentru ora curentă și, desigur, JPanel. În constructorul acestei clase (care are 2 parametri: numărul de servere și numărul de clienți generați) inițializam etichetele și panoul de conținut. Eticheta de timp este inițializată cu textul „TIME=0” și toate celelalte etichete sunt inițializate cu textul „”, deoarece textul lor va fi setat în controler cu metodele „toString()” ale serverelor și sarcinilor.

**3.3 Pachetul ”Controller”**

Acest pachet conține următoarele clase:

- Simulator

- Controller

Clasa „Simulator” implementează interfața „Runnable” și este firul principal al întregii aplicații. Atributele sale sunt:

- Numerele întregi pentru toți parametrii de simulare introduși de utilizare în vizualizarea setup;

- un planificator;

- o coadă de blocare pentru toți clienții generați;

- timpul mediu de așteptare pentru toți clienții în simulare;

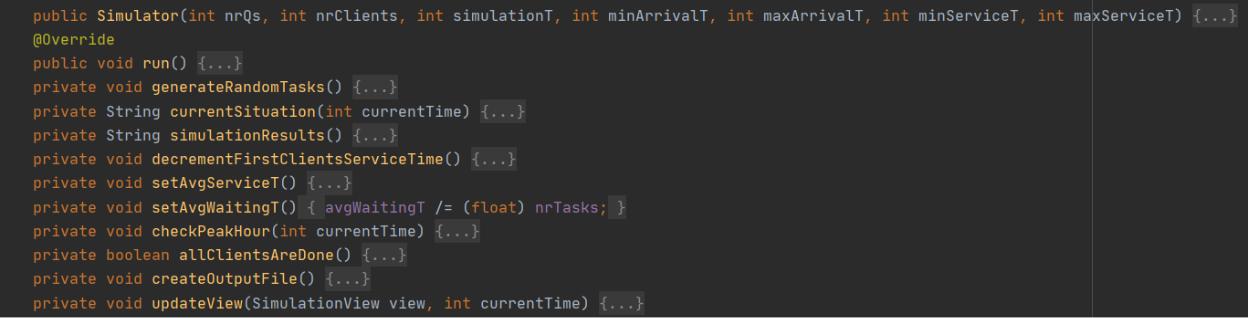
- timpul mediu de service pentru toți clienții din simulare;

- ora de vârf;

- Întregul timp de service al tuturor clienților aflați la cozi în timpul orelor de vârf;

- un obiect „PrintWriter” pentru scrierea logului de simulare într-un fișier „.txt”;

- un obiect „SimulationView”.



În constructor inițializam toți parametrii simulării, planificatorul, vizualizarea și apelăm metodele „generateRandomTasks()” și „setAvgServiceT()”.

În metoda „generateRandomTasks()” obiectivul este de a crea clienți nrTasks și de a le sorta după ora de sosire, motiv pentru care clasa „Task” scrisă implementează interfața „Comparable”. Nu putem sorta direct o coadă de blocare, așa că folosim o colecție auxiliară: o listă de matrice. Astfel, sarcinile pe care le generăm sunt adăugate la lista de matrice, pe care o sortăm. După aceea, setăm id-ul fiecărui client în ordinea orei de sosire și îl adăugăm la coada de blocare.

Text

Description automatically generated

Metoda „setAvgServiceT()” calculează pur și simplu media timpului de service al fiecărui client, deoarece vom dori să afișăm acest rezultat în jurnalul generat la sfârșitul simulării.

Metoda „run()” este metoda de bază a acestei clase. Inițializam întregul „currentTime” care va reprezenta secunda curentă în timpul simulării, apelăm metoda „createOutputFile()” care creează pur și simplu fișierul „.txt” în care vom afișa rezultatele și apoi începem bucla „while” a cărei condiție este ca ora curentă să nu fi atins încă valoarea dată pentru timpul de simulare. În această buclă, trecem prin coada de blocare a sarcinilor generate și apelăm planificatorului să trimită sarcinile a căror oră de sosire a ajuns la ora curentă. Valoarea returnată de metoda „dispatch(Task t)” este adăugată la variabila „avgWaitingT” și apoi sarcina expediată este eliminată din coada de blocare. În continuare, apelăm metoda „checkPeakHour(currentTime)” care calculează suma timpilor de serviciu pentru toți clienții aflați în prezent în cozi și va găsi valoarea maximă a acestor rezultate. După aceasta, apelăm metoda care scrie situația curentă a cozilor în fișierul de ieșire.

Text

Description automatically generated

Următoarea metodă apelată aici este „updateView(view, currentTime)”, care actualizează textul dintre toate etichetele din simulare. Această metodă mai întâi apelează metoda „allClientsDone()” care verifică dacă toate cozile de blocare a sarcinilor sunt goale. Dacă da, simularea trebuie să se încheie și nicio etichetă nu este actualizată. Altfel, eticheta timpului este actualizată cu ora curentă, etichetele cu clienții în așteptare sunt actualizate prin bifarea cozii „generatedTasks”, care va avea sarcini eliminate din ea în timpul simulării și, în sfârșit, etichetele serverelor sunt actualizate.

Text

Description automatically generated

Să revenim la metoda „run()”. Apoi apelăm metoda „sleep()“ timp de o secundă într-un bloc try-catch și verificăm dacă toate sarcinile au fost procesate chiar dacă simularea nu a ajuns la final. Dacă da, afișăm încă o dată situația actuală, astfel încât să se arate că toate cozile sunt goale, afișăm ora de vârf, timpul mediu de așteptare și timpul mediu de service și închidem fișierul de ieșire. Dacă toate listele nu sunt goale, dar timpul de simulare a expirat (după bucla „while”), afișăm și rezultatele și închidem fișierul.

Clasa „Controller” are ca atribute un simulator și o vedere de configurare. În constructorul său, inițializează vizualizarea de configurare și apelează ascultătorul butonului „RUN SIMULATION”. Când butonul este apăsat, numim metoda „startSimulator()”, care parsează toate șirurile de intrare în numere întregi într-un bloc try-catch pentru a ne asigura că utilizatorul a introdus numere valide. După aceea, inițializam simulatorul și începem firul său. În clasa principală, singurul lucru pe care trebuie să-l facem este să creăm un obiect „Controller”.

# Testarea

Trei seturi de parametri de intrare au fost selectate pentru a testa aplicația, așa cum a fost specificat în cerința proiectului. Pentru fiecare set, s-a generat câte un fișier de ieșire cu rezultatele simulării.

Text

Description automatically generated

# Concluzii

Cred că acest proiect reprezintă o implementare bună a conceptului de multi-threading, dar există câteva îmbunătățiri care ar putea fi aduse. Unele dintre acestea sunt o interfață grafică mai bună (un cadru derulabil, animații), eventual o clasă care generează clienți până când utilizatorul apasă un buton care o oprește și mai multe strategii prin care clienții sunt distribuiți la cozi.

# Bibliografie

* <https://www.javacodegeeks.com/2013/01/java-thread-pool-example-using-executors-and-threadpoolexecutor.html>
* <https://www.tutorialspoint.com/java/util/timer_schedule_period.htm>
* <https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html>