**DOCUMENTATIE TEMA NR.2**

QUEUES SIMULATOR

Student: Bumbu Iulia-Diana

An: II Semestrul: 2

Grupa: 30228

CUPRINS:

1. Obiectivul temei
2. Analiza problemei
3. Proiectare
4. Implementare
5. Rezultate
6. Concluzii
7. Bibliografie
8. Obiectivul temei:

1.1.Obiectiv principal:

Principalul scop al proiectului este de a realiza o aplicatie care sa gestioneze intr-un mod optim distributia unui numar dat de clienti la un numar dat de cozi, astfel incat fiecare client sa astepte la coada timp minim. Pentru fiecare client care trebuie distribuit se cunoaste timpul de sosire, timpul necesar pentru procesare si id-ul considerat unic. In functie de acestea proiectul realizat asigneaza respctivul client la cea mai optima coada pentru el.

1.2.Obiective secundare:

Pentru o dezvoltare optima a unei aplicatii de acest tip trebuie realizati anumiti pasi pentru indeplinirea obiectivului principal.

Principalele **obiective secundare** sunt:

* Realizarea USE CASE-urilor si a scenariilor:

O diagrama a cazurilor de utilizare ofera o descriere generala a modului in care va fi utilizat sistemul, furnizand o privire de ansamblu asupra functionalitatilor care doresc a fi oferite de sistem. (capitolul 2)

* Identificarea structurilor de date:

Structura de date este o metoda sistematica de stocare a informatiilor intr-un calculator astfel incat sa poata fi folosite in mod eficient. (capitolul 3)

* Impartirea claselor:

Pentru procesarea simultana a mai multor clienti, respectiv a mai multor cozi este imperativa folosirea threadurilor, asa ca se pot imparti clasele in functie de conceptele oferite de multithreading pentru o procesare cat mai eficace a clientilor (capitolul 3)

* Dezvoltarea algoritmilor:

Se va determina strategia necesara pentru a gasi coada cu cel mai scurt timp de asteptare. (capitolul 3)

* Implementarea solutiei:

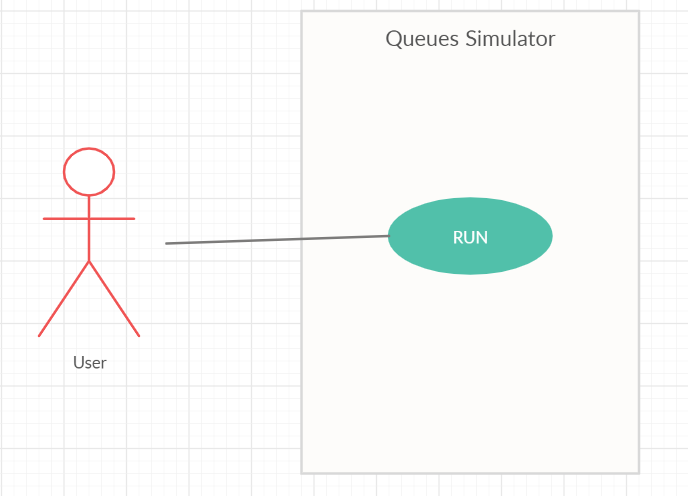
Este reprezentata de descrierea in cod a tuturor claselor anterior explicate, a metodelor necesare efectuarii distributiei precum si procesarea de fisiere pentru a testa functionalitatea programului creat. (capitolul 4)

* Gestionarea rezultatelor:

Se vor citi datele de intrare din fisierele in-test.txt, iar rezultatul generat se va scrie in out-test.txt. (capitolul 5)

2.Analiza problemei:

2.1.USE CASE Diagram:



2.2. Scenarii:

O utilizare corecta a programului presupune introducerea unui fisier in-test.txt cu continut valid pentru a determina conditiile initiale de desfasurare. Un input valid se refera la specificarea:

-pe prima linie: numarul de clienti(N)

-pe a doua linie: numarul de cozi(Q)

-pe a treia linie: timpul maxim de simulare

-pe a patra linie: timpul minim si maxim de sosire al clientilor separate printr-o virgula

-pe a cincea linie: timpul minim si maxim de prelucrare al clientilor separate printr-o virgula

Dupa procesarea inputului valid ca raspuns se vor scrie in fisierul out-test.txt toti pasii simularii pana in momentul in care este atins timpul maxim de simulare sau ultimul client din lista de asteptare a fost distribuit unei cozi. De asemenea, dupa simularea tuturor pasilor se scrie in fisier si o valoare reala corespunzatoare timpului mediu de asteptare. In fisierul de iesire formatul pentru fiecare timp este urmatorul:

Time x

Waiting clients: ( , , ) ( , , ) ( , , ) ...( N clients)

Queue1: ( , , ) (toti clientii distribuiti) || closed

Queue2: ( , , ) (toti clientii distribuiti) || closed

..............................................

Queue Q: ( , , ) (toti clientii distribuiti) || closed

Time x+1

.........................................................................

Average waiting time: value

1. Proiectare:

3.1.Structuri de date:

Deoarece multipli clienti trebuie procesati in mai multe metode adesea este folosita ca structura de date ArrayList deoarece faciliteaza lucrul cu vectori prin metodele preimplementate.

ArrayList<Client> list= **new** ArrayList<Client>();

Totusi, tinand cont de faptul ca ne folosim de threaduri in rezolvarea problemei este necesara si folosirea unor colectii “thread safe”, precum colectia concurenta BlockingQueue:

BlockingQueue<Client> clienti = **new** ArrayBlockingQueue<Client>(maxClientsPerServer);

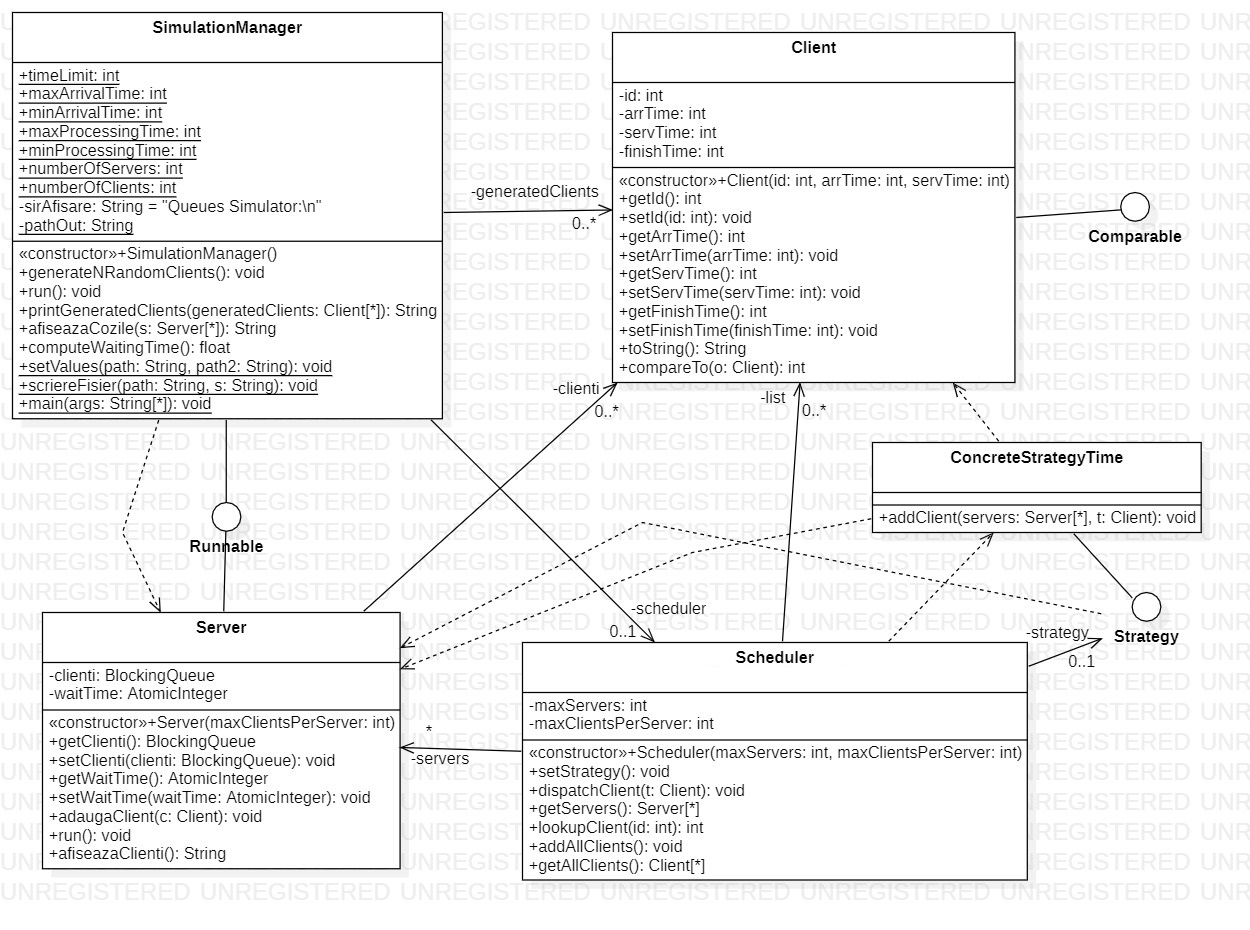
Un thread care incearca sa stearga elemente dintr-o BlockingQueue goala este blocat pana cand un alt thread insereaza elemente in ea. Daca un thread incearca sa adauge elemente in coada cand aceasta este pline operatia este blocata pana cand un alt thread sterge unul sau mai multe elemente.

De asemenea, deoarece mai multe threaduri procesaza acelasi sir de clienti din waiting list este necesara ca modificarea datelor clientilor sa se faca instant pentru a nu se introduce date eronate in calcul. Pachetul java.util.concurrent.atomic dispune de aceasta facilitate deoarce valorile se updateaza atomic:

**private** AtomicInteger waitTime;

3.2.Diagrama de clase

Diagrama de clase este folosita in modelarea orientata obiect pentru a descrie structura statica a sistemului si a modului in care este el structurat. De asemenea, aceasta ofera o notatie greafica pentru reprezentarea claselor (entitati care au caracteristici comune) si a relatiilor (legaturi dintre doua sau mai multe clase) dintre ele.



3.3.Diagrama de pachete:



UML furnizeaza mijloace de grupare a elementelor din cadrul diagramelor numite pachete. Pachetele constituie baza pentru controlul configuratiei si stocare, contribuind la reutilizarea codului.

3.4.Algoritmi:

Strategia corespunzatoare distribuirii unui client la coada cu timp de asteptare cel mai optim pentru el se face conform pseudocodului de mai jos:

**function** adauga (Client c) **is**

destinatie← ?

min←infinite

**for each** q - coada neverificata **do**

**if**  q.waitTime < min **then**

min←q.waitTime

destinatie←q

**add** (Client c, destinatie)

1. Implementare:

4.1.Descrierea claselor din pachetul Clienti:

* Clasa Client:

Are ca principala functionalitate stocarea datelor de identificare ale fiecarui client deoarece contine atributele:

**private** **int** id;

**private** **int** arrTime;

**private** **int** servTime;

**private** **int** finishTime;

Fiecare client are un id de identificare unic cuprins intre 0 si numarul de clienti care trebuie distribuiti la cozi. arrTime se refera la timpul de sosire la care clientul este gata sa fie distribuit. servTime implica cat timp dureaza ca clientul sa fie servit, adica cat timp este necesar pentru procesarea sa exclusiva. Pe langa cerintele din enunt s-a adaugat un atribut finishTime care retine cat timp a stat fiecare client in coada, tinand cont si de faptul ca inaintea lui puteau fi distibuiti deja alti clienti. Acesta faciliteaza ulterioara calculare a timpului mediu de asteptare.

Clasa contine doua metode

**public** String toString() -folosita pentru afisarea datelor fiecarui client

**public** **int** compareTo(Client o) -implementata pentru sortarea ulterioara a unui sir de clienti dupa timpul de sosire utilizand metoda:

void [java](eclipse-javadoc:%E2%98%82=TPPolinomCalc/C:%5C/Program%20Files%5C/Java%5C/jdk-13.0.1%5C/lib%5C/jrt-fs.jar%60java.base%3Cjava).[util](eclipse-javadoc:%E2%98%82=TPPolinomCalc/C:%5C/Program%20Files%5C/Java%5C/jdk-13.0.1%5C/lib%5C/jrt-fs.jar%60java.base%3Cjava.util).[ArrayList](eclipse-javadoc:%E2%98%82=TPPolinomCalc/C:%5C/Program%20Files%5C/Java%5C/jdk-13.0.1%5C/lib%5C/jrt-fs.jar%60java.base%3Cjava.util(ArrayList.class%E2%98%83ArrayList).sort([Comparator](eclipse-javadoc:%E2%98%82=TPPolinomCalc/C:%5C/Program%20Files%5C/Java%5C/jdk-13.0.1%5C/lib%5C/jrt-fs.jar%60java.base%3Cjava.util(ArrayList.class%E2%98%83ArrayList~sort~Ljava.util.Comparator%5C%3C-TE;%3E;%E2%98%82java.util.Comparator" \o "in java.util)<? super Client> c)

4.1.Descrierea claselor din pachetul Simulation:

* Clasa Server:

Are ca principala functionalitate modelarea conceptului de “coada”. Deoarece implementeaza interfata Runnable, aceasta clasa poate fi ulterior folosita pe post de thread. Aceasta clasa are ca atribute un sir de clienti precum si un AtomicInteger waitPeriod care prin care se identifica cat timp ar trebui sa astepte un nou client asignat la respectiva coada.

Constructorul clasei **public** Server(**int** maxClientsPerServer) initializeaza cele doua atribute:

**private** BlockingQueue<Client> clienti;

**private** AtomicInteger waitTime;

Clasa contine si metodele:

**public** **void** adaugaClient(Client c) –cu functionalitatea de a adauga un nou client in BlockingQueue-ul de clienti. De asemenea, in aceeasi metoda se seteaza si timpul de finish al clientului inserat ca fiind timpul de servire+ timpul de asteptare din acel moment. Dupa adaugare in coada de clienti se creste waitTime-ul cu o valoare egala cu timpul de servire al noului client inserat

**public** String afiseazaClienti()–utilizata pentru a afisa toti clientii din BlockingQueue de la un moment dat

**public** **void** run()-deoarece clasa impementeaza interfata Runnable aceasta metoda trebuie detaliata pentru a specifica ce trebuie sa faca Threadul dupa ce este pornit. Daca exista client in varful cozii aceasta este procesat astfel: se decrementeaza timpul de servire. In cazul in care timpul de servire al unui client a ajuns la zero inseamna ca acesta a fost complet procesat asa ca este scos din coada. Se decrementeaza waitPeriod cu o valoare egala cu service time al clientului din head-ul cozii, dupa care Threadul se pune pe sleep o secunda deoarece procesarea sa la momentul respectiv a fost complet facuta. In cazul in care nu avem niciun client in coada se opreste direct Threadul o secunda. Oprirea threadului este necesara pentru folosirea dinamica a cozilor.

* Clasa ConcreteStrategyTime:

Implementeaza interfata Strategy si ca urmare contine metoda:

**public** **void** addClient(ArrayList<Server> servers, Client t)

care determina in care dintre cozile existente un nou client ar trebui sa astepte cel mai putin timp conform algoritmului specificat in sectiunea 3.4

* Clasa Scheduler:

Este clasa responsabila cu managementul cozilor si cu distribuirea clientilor. O parte dintre atribute: **private** **int** maxServers;

**private** **int** maxClientsPerServer; sunt initializate in constructorul clasei:

**public** Scheduler(**int** maxServers, **int** maxClientsPerServer) –tot aici se creeaza si se deschid un sir de Threaduri corespunzatoare cozilor in care vor fi ulterior distribuiti clientii. Fiecarui Thread ii este asociat un Server din **private** ArrayList<Server> servers;

declarat ca atribut.

**public** **void** setStrategy()–are rolul de a creea o noua strategie specificata ca atribut. Aceasta metoda este utila pentru schimbarea usoara in cazul in care se implementeaza noi strategii

**public** **void** dispatchClient(Client t) –apeleaza metoda de addClient a strategiei create

De asemenea, clasa contine o noua lista de clienti in care se salveaza fiecare nou client adaugat in una dintre cele maxServers cozi create. Aceasta este folosita in ulterioara calculare a mediei de asteptare:

**private** ArrayList<Client> list= **new** ArrayList<Client>();

**public** **int** lookupClient(**int** id) –metoda care verifica daca exista deja in list un client cu id-ul trimis ca parametru pe a asigura unicitatea acestora

**public** **void** addAllClients()–adauga in sirul list fiecare client din cozile create atata timp cat el nu a fost deja introdus. Astfel, la finalul executiei in list vom avea toti clientii din initiala coada de asteptare, insa cu finish time calculat

* Clasa Simulation manager:

Este clasa responsabila cu generare clientilor, cu crearea simularii si de gestionarea rezultatelor.

Constructorul clasei **public** SimulationManager() realizeaza o noua instanta a planificatorului Scheduler, seteaza strategia si apeleaza metoda corespunzatoare generarii clientilor.

**public** **void** generateNRandomClients()-reprezinta metoda in care se adauga in sirul

**private** ArrayList<Client> generatedClients = **new** ArrayList<Client>();

cei *numberOfClients* clienti generati random cu ajutorul clasei ThreadLocalRandom. Se tine cont ca fiecare timp de sosire/ de prelucrare generat random trebuie sa fie intre limitele impuse de atributele specificate. De asemenea, pentru a se urmari mult mai usor evolutia clientilor acestia sunt sortati dupa timpul de sosire

**public** String printGeneratedClients(ArrayList<Client> generatedClients) –are rolul de a afisa toti clientii ramasi in lista de asteptare: generatedClients

**public** String afiseazaCozile(ArrayList<Server> s) –cu functie sugerata si de nume: afiseaza clientii existenti in fiecare coada la un moment dat

**public** **float** computeWaitingTime()–calculeaza suma tuturor timpilor de finish ale clientilor pentru a ajuta la calculul timpului mediu de asteptare

**public** **static** **void** setValues( String path, String path2) –aceasta metoda deschide fisierul trimis ca argument path si citeste si proceseaza din acesta valorile numerice pe care le asociaza atributelor declarate anterior: *timeLimit, maxArrivalTime, minArrivalTime, maxProcessingTime, minProcessingTime, numberOfServers, numberOfClients.*

Cel de al doilea parametru este retinut in atributul *pathOut* reprezentand calea fisierului out in care se va scrie rezultatul

**public** **static** **void** scriereFisier( String path,String s) –cu scop de a scrie in fisierul cu calea *pathOut* rezultatul simularii pastrat in atributul **private** String sirAfisare

**public** **void** run()–fiind dat faptul ca si aceasta clasa implementeaza interfata Runnable aceasta metoda este implementata. Cat timp timpul curent nu este mai mare decat cel impus de datele problemei sau lista cu clientii aflati in asteptare nu este inca goala se itereaza lista de clienti generatedClients si se aleg clientii care au timpul de sosire egal cu timpul curent pentru a fi trimisi la coada corespunzatoare lor. Deoarece au fost procesati se sterg din lista initiala. Se updateaza Stringul care trebuie scris in fisierul de iesire la fiecare timp si dupa un interval de o secunda se trece la timpul incrementat. Dupa ce toti clientii au fost procesati sau limita de timp a fost depasita se poate calcula timpul mediu de asteptare: daca toti clientii au fost procesati acesta este egal cu suma timpilor de finalizare divizata cu numarul de clienti. In caz contrar, din numarul de clienti se scad cei care nu au reusit sa fie procesati deoarece aveau arrival time mai mare decat limita de timp impusa. Dupa ce toate aceste date au fost calculate se apeleaza functia care scrie in fisierul de iesire rezultatul final

**public** **static** **void** main(String[] args) –metoda contine ca parametri calea fisierului de intrare, respectiv a fisierului de iesire. Dupa ce se apeleaza functia de initialiare a datelor setValues se creeaza o noua instanta a simularii, dupa care raportat la aceasta se creeaza si porneste un nou Thread responsabil cu creerea buclei de simulare

1. Rezultate:

Pentru realizarea verificarii rezultatelor programul este configurat sa ruleze din terminal si sa citeasca doi parametri: calea fisierului de intrare si calea fisierului de iesire. In acest sens s-a configurat un fisier .jar

.

1. Concluzii:

In concluzie, aceasta tema mi-a oferit oportunitatea de a ma familiariza cu threadurile deoarece pana in momentul de fata nu am avut ocazia sa lucrez cu ele. Astfel, am sedimenta o parte din conceptele de baza ale lucrului cu acestea si am aprofundat alte cunostinte in legatura cu POO . Proiectul poate avea dezvoltari ulterioare prin conceperea de noi strategii de distribuire a clientilor sau prin creerea unei interfete grafice care sa usureze setarea datelor initiale.

7. Bibliografie:

1.<http://coned.utcluj.ro/~salomie/PT_Lic/4_Lab/Assignment_2/Java_Concurrency.pdf>

2. <https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html>

3. <https://www.geeksforgeeks.org/multithreading-in-java/>