Tema 2

Queues simulator

Stefanovici Miruna Andreea

Grupa 30229

**Continut**

**1.***Obiectivul temei*

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

**3.***Proiectare (decizii de proiectare, diagrame UML, structuri de date, proiectare clase, interfete, relatii, packages, algoritmi, interfata utilizator)*

**4.***Implementare*

**5.***Rezultate si concluzii*

**6.***Bibliografie*

**-1- Obiectivul temei**

Obiectivul temei este de a transpune intr-un model programatic, modelul din viata reala de “coada”. Astfel, comparand simplu cu modelul real, deducem ca rolul unei cozi este de a face loc unui “client”, ce “asteapta” un timp pentru anumite servicii, dupa dobandirea acestora, fiind nevoit sa paraseasca respectiva coada, pentru a face loc unui alt client, procesul repetandu-se pana cand in coada nu mai exista niciun client.

Pentru ca scopul principal sa poata fi atins, trebuie urmarita o serie de pasi:

-analizarea potentialelor scenarii de folosire a aplicatiei de catre un utilizator, a interactiunii dintre user si program;

-alegerea modului de proiectare, a structurilor de date folosite in cadrul acesteia;

-alegerea modului de implementare a algoritmilor;

-descrierea claselor prin evidentierea rolului fiecarui camp si fiecarei metode;

-analizarea rezultatelor in urma implementarii, pe testarea unor exemple simple, moderate si complexe.

**-2-** **Analiza problemei, modelare, scenarii, cazuri de utilizare**

Scenariul de utilizare cerut este urmatorul:

* se introduce numarul de clienti ce vor astepta la cozi;
* se introduce numarul de cozi disponibile pentru clienti;
* se introduce timpul intregii simulari;
* se introduce intervalul din care vor fi extrasi timpii de sosire in simulare;
* se introduce intervalul din care vor fi extrasi timpii de asteptare;
* se va afisa la fiecare moment al simularii : numarul de clienti in asteptare si distributia clientilor in mod favorabil ( in functie de cel mai mic timp de asteptare) la cozile disponibile, care se inchid atunci cand nu au de “servit” niciun client;
* la final se afiseaza media timpilor de asteptare a tuturor clientilor procesati.

Datele se citesc din fisiere, iar rezultatele se scriu in alte fisiere

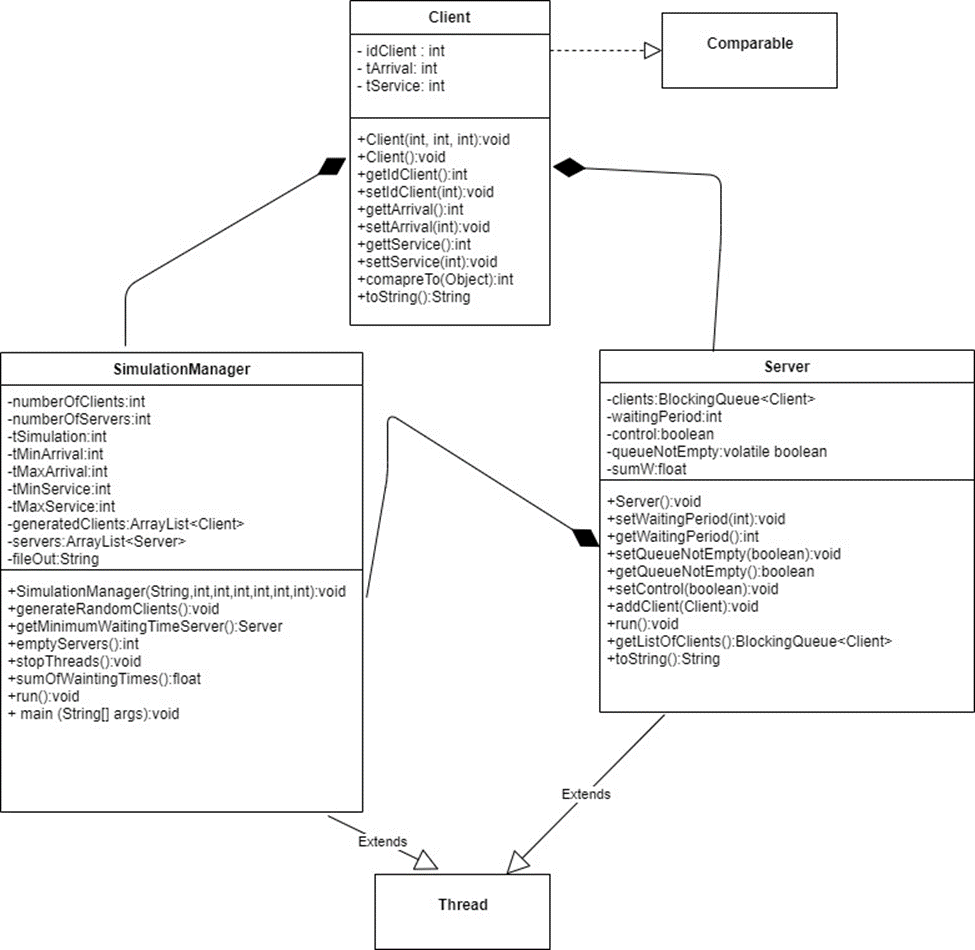
Aceasta simulare va putea fi vizualiata intr-un mod inteligibil, prin folosirea in background a conceptului de **Threading**.

**-3-Proiectare (decizii de proiectare, diagrame UML, structuri de date, proiectare clase, interfete, relatii, packages, algoritmi, interfata utilizator**

In ceea ce priveste proiectarea aplicatiei, simulatorul de cozi implementat are in spate principiul de “Fir de executie”. Aplicatiile ruleaza in mod normal in interiorul unui process al sistemului de operare, acest proces avand ca unitate de executie “un fir de executie”(thread). Fiecarui thread i se asociaza o secventa de instructiuni pe care le va executa. Principala diferenta intre thread si process : thread-ul se executa in spatiul de adresare al procesului din care face parte, proces care poate contine si alte thread-uri. Astfel, un anumit thread reuseste sa modifice valori care sunt vazute si de celelate fire. Aici apare problema sincronizarii. Firele respective au nevoie sa comunice intre ele, sa acceseze intr-un mod controlat datele utilizate in comun. Sincronizarea astfel vitala pentru intregul mecanism, asigura siguranta datelor, pentru ca acestea sa nu poata fi modificate intr-un mod ce poate distruge logica programului. Importanta thread-urilor apare in contextual in care o aplicatie doreste sa execute in paralel mai multe sarcini (multithreading). Acest principiu, de a realiza in paralel mai multe task-uri, poate fi folosit pentru sporirea eficientei in cadrul multiprogramarii.Exemplul cel mai bun si simplist in acest sens este chiar simularea unor cozi obisnuite la magazine. Clientii vin, isi expun cererile, primesc ceea ce isi doresc si parasesc cozile, dand ocazia noilor clienti sa faca acelasi lucru.Thread-urile ofera cumva ocazia unei abordari, de implementare a unei aplicatii, cat mai apropiata de activitatea umana.

Astfel, programul creat imita modelul real prin exitenta clasei **Client**, ce are ca si variabile instanta un intreg ce reprezinta un identificator unic, un intreg ce reprezinta momentul in care “clientul” ajunge in coada in timpul simularii si un alt intreg ce reprezinta timpul de prelucrare a “cerintelor”. Vizualizarea “scurgerii” acestui timp de asteptare este posibila tocmai datorita folosirii thread-urilor. De obiectele de tip “Client” se ocupa clasa **Server**, ce extinde clasa Thread si care va imita modelul real al celui care ofera serviciile. Astfel, clasa are ca variabila instanta un intreg ce reprezinta timpul de asteptare la coada respectiva pentru urmatorii clienti, aceasta incrementandu-se la adaugarea fiecarui client, cu timpul de asteptare al fiecarui client, urmand ca in timpul procesarii acestuia sa fie decrementat cu 1, cat timp durata de asteptare a clientului n-a ajuns inca la 0. Aceasta variabila este utila in contextual in care, imitand de asemenea si aici corectitudinea din viata reala(in unele cazuri), dorim sa plasam fiecare client urmator la coada la care are cel mai putin de asteptat, pentru obtinerea serviciilor. Clasa detine inca patru variabile instanta a caror existenta joaca un rol important in controlul thread-urilor in mod paralel. Pentru a asigura o buna functionarea a acestui paralelism, in mod normal, cand se lucreaza cu thread-uri, este recomandata utilizarea datelor de tip “Thread safe” ce ajuta la sincronizarea datelor, nepermitand accesul haotic la informatiile partajate in comun, programul reusind astfel sa se desfasoare conform logicii create. Asemenea tipuri de date ce ajuta la scrierea programelor concurente sunt de exemplu, volatile, AtomicInteger. De asemenea, se recomanda utilizarea colectiilor de tip “Thread safe”, precum BlockingQueue. Logica de procesare a clientilor fiind create duce la existenta clasei **SimulatorManager** care primeste datele necesarii simularii ( descries in capitolul 2) si de asemenea extinde clasa *Thread*, fiind firul principal de executie ce se poate spune ca are posibilitatea de a coordona desfasurarea in paralel a celorlalte fire (de tip Server), pornindu-le si inchizandu-le la finalizarea completa a task-urilor pana cand se ajunge la timpul maxim de simulare sau se termina de procesat pana la ultimul client.

**Diagrama UML**(*exista si ca poza separata si este adaugata impreuna cu toate celelalte fisiere*):



**-4-Implementare**

Urmeaza descrierea fiecarei clase.

*Clasa* ***Client***

Clasa client are ca variabile instanta *idClient*, ce reprezinta identificatorul fiecarui client, *tArrival*, ce reprezinta timpul la care clientul trebuie adaugat la co coada si *tService*, ce reprezinta timpul efectiv de asteptare a procesarii respectivului client.

Constructorul, setter-ele si getter-ele au rolurile de a seta campurilor unui anumit obiect, anumite valori sau de a accesa valorile acestor campuri.

Am adaugat, in mod necesar, metoda compareTo ( deoarece se implementeaza interfata Comparable) ce decide modul de sortare a clientilor in ordinea crescatoarea a timpilor de sosire la coada si functia toString() ce permite afisarea intre paranteze rotunde, a variabilelor instanta in ordinea in care au fost prezentate, acesta insiruire representant *clientul*.

*Clasa* ***Server***

Este firul de executie ce va asigura procesarea clientilor in mod paralel, in functie de distributia la o anumita coada. Am ales sa folosesc varianta in care definesc o clasa care mosteneste clasa predefinita **Thread**. Ca si variabile am lista de clienti, *clients*, ce alcatuiesc o coada, find de tipul BlockingQueue<Client> pentru a pastra principiul de “Thread Safety”. Am ales sa implementez coada folosind clasa LinkedBlockingQueue, ce implementeaza BlockingQueue, pentru a stoca elementele in ordinea FIFO ( first in, first out) .

De asemenea, ca variabile instanta mai folosesc si : *waitingPeriod*, ce reprezinta perioada de asteptare a clientilor , informatie pe care acestia o “cunosc” atunci cand vor sa intre intr-o coada; *control*, ce este de tip Boolean si stabileste sincronizarea cozilor, *queueNotEmpty* ce este tot de tip Boolean, dar si volatile (recomandat pentru principiul de “Thread Safety”) si *sumW*, o variabila de tip float care va calcula suma timpilor de asteptare pentru fiecare client, ca la sfarsit sa se poata realiza o medie a timpului de asteptare.

Constructorul, setter-ele si getter-ele au rolurile de a seta campurilor unui anumit obiect, anumite valori sau de a accesa valorile acestor campuri.

Clasa **Thread** este definite in pachetul java.Lang avand o serie de metode. Principala ei metoda este metoda **run(),** ce trebuie sa contina toate activitatile pe care firul ar trebui sa le execute. Astfel, metoda suprascrisa ***run()***, in clasa Server va realiza activitatea fiecarui fir de executie paralel ( fiecarui Server) cat timp booleana *control* este true. Verifica daca variabial instanta coada este goala, iar daca nu este goala, extrage ,print metoda **peek()** specifica respectivei colectii alese, primul element din coada si il proceseaza: cat timp acestuia nu i-a expirat timpul de asteptare, i se va decrementa si seta acest tService, variabila *sumW* se va incrementa cu lungimea cozii la fiecare moment(ca la sfarsitul procesarii tuturor clientilor sa fie egala cu suma timpilor de asteptare pentru fiecare client in parte), iar variabila *waitingPeriod* se va decrementa la fiecare moment al scurgerii timpului de asteptare, pentru a asigura mentinerea reguli de adaugare a urmatorului client la coada cu timpul minim de asteptare. Inaintea modificarii acestor date, se apeleaza metoda **sleep()** a clasei predefinite Thread , cu un argument care reprezinta timpul in milisecunde in care firul de executie asteapta(aici 1000). Aceaste metoda, de fapt, cere oprirea rularii firului de executie curent .Astfel , va fi afisat continutul cozilor la fiecare moment de procesare al clientilor aflati in cozi, in paralel.Dupa ce clientului i se termina timpul de asteptare al *procesarii* efective, este eliminat din coada. Daca in coada nu se mai afla niciun alt client, booleana *queueNotEmpty*, va fi setata pe false. Aceasta setare va ajuta la verificarea ulterioara a cozilor, in clasa *SimulatorManager*. Pe langa aceasta metoda, clasa **Server** mai detine si metoda de accesare a listei de client **, getListOfClients()**, precum si metoda **toString()** ce permite afisarea in formatul (id,tArrival,tService) a fiecarui client aflat in coada respective.

*Clasa* ***SimulationManager***

Este firul de executie principal, ce va avea datele necesare coordonarii firelor de executie de tip Server. Clasa are ca variabile instanta informatiile necesare enumerate in cap.2 si anume : *tSimulation*(timpul maxim de simulare), *tMinArrival, tMaxArrival* ( capetele intervalului din care vor fi generati random timpii de adaugare in coada), *tMinService, tMaxService* ( -//- timpii de asteptare), numberOfClients, numberOfServers si cele doua liste: cea de clienti *generatedClients* ce reprezinta lista de asteptare a clientilor de a fi distribuiti in cozi si lista *servers* ce reprezinta lista cozilor la care vor fi distribute clientii. Am folost ca stuctura de date pentru aceste liste ArrayList-ul pentru usurinta de a adauga sau a sterge elemente din listele respective. De asemenea, clasa mai are ca variabial instanta un string *fileOut* ce reprezinta path-ul fisierului in care vor fi scrise datele de iesire. In constructor are loc setarea datelor ce vor fi citite din fisier in metoda main a acestei clase ( cu informatii despre REGEX dobandite in urma efectuarii temei trecute, reusind sa extrag altfel in mod correct datele din fisier, convertite la Int). De asemenea, pe langa setarea datelor, in constructor este apelata metoda **generateRandomClients()** care imi va adauga in lista de clienti aflati in asteptare, client cu timpi de sosire si asteptare random, din intervalele stabilite la citirea din fisier. Lista de client va fi sortata in ordinea crescatoare a timpilor de sosire in coada. Un aspect important al constructorului este ca in interiorul lui, in functie de numarul de queues citit din fisier, se vor crea obiecte de tip Server, care la randul lor, fiind thread-uri, vor fi pornite. Prin apelul **start()** se cerea pornirea unui nou fir de executie. Astel aplicatia este formata din numbersOfServers+1 threaduri, acel +1 fiind clasa SimulationManager. Pe langa functia **run()** a thread-ului “cel mare”, am implementat alte functii ajutatoare organizarii codului : metoda **getMinimumWaitingTimeServer()** returneaza coada cu timpul de asteptare minim; metoda **emptyServers()** returneaza 1 daca lista de cozi contine numai cozi goale, asta insemnand ca toti clientii au fost procesati si se poate iesi din simulare, sau 0 daca aceasta mai contine cozi la care se afla client, fapt ce duce la continuarea simularii pana la procesarea complete a tuturor clientilor, procesare ce se incadreaza pana in timpul maxim al simularii (exista posibilitatea sa existe client care nu vor putea fi procesati complet sau deloc pe durata simularii); metoda **stopThreads()** care va realiza inchiderea tuturor thread-urilor de tip server la finalizarea simularii sau atunci cand au fost procesati toti clientii chiar daca timpul de simulare inca nu s-a scurs; metoda **sumOfWaintingTimes()** care face o suma a timpilor de asteptare din fiecare coada, pentru a se putea calcula timpul mediu de asteptare a unui client in coada, pana sa fie procesat, tinandu-se cont ca la timpul lui efectiv de asteptare (tService) se adauga si timpul in care asteapta ca cei din fata lui sa fie procesati ( pana ii vine randul ). In metoda **run()** se initializeaza timpul curent la 0 (precum si numarul de client procesati) si cat timp acesta este mai mic decat timpul maxim de simulare se efectueaza o serie de pasi, ce se repeta :

-Se parcurge lista de clienti aflati in asteptare ( am folosit interfata **Iterator** disponibila in Collection framework pentru a reusi sa accesez obiectele unul cate unul si in acelasi timp sa le pot si sterge din lista respectiva , fara a intampina erori). Astfel, **iterateWaitingClients** va tine lista de client, iar cat timp exista un client in aceasta lista, va verifica daca timpul lui de sosire se potriveste cu timpul current. Daca gaseste aceasta potrivire, clientul este adaugat in coada cu timpul de asteptare minim, se va seta variabila **queueNotEmpty** pe false pentru a indica faptul ca in coada exista cel putin un element, va creste numarul de clienti procesati, iar clientul va fi scos din lista de asteptare. Astfel, urmeaza afisarea situatiei in coada la fiecare currentTime ( atunci cand in coada nu va fi niciun client, aceasta se va considera “closed”). Daca in lista de asteptare nu mai sunt client, simularea va continua pana cand vor fi procesati toti clientii, adica pana cand functia emptyServers() va returna 1. Se incrementeaza timpul current si se invoca metoda **sleep()**, tot timp de 1000 de secunde, pentru sincronizarea cu afisarea continutului din Server la fiecare currentTime trecut. Metoda sleep() functioneaza in general in blocul de try-catch, deoarece, in caz ca firul de executie pus sa “doarma” este intrerupt, se va genera exceptia “InterruptedException”. La iesirea din while, se opresc toate firele de executie si este calculate timpul de asteptare minim ca raportul dintre sumOfWaintingTimes si numarul de client procesati, **numberVar**.De asemenea, in interiorul acestei metode au loc scrierile in fisier prin folosirea obeictului de tip *PrintWriter* , ce functioneaza ca o afisare efectiva, cu exceptia faptului ca respectivul continut este scris in fisierul dat. La sfarsitul scrierii, acesta trebuie inchis. In metoda main, va avea loc pornirea thread-ului cel mare.

**-5- Rezultate**

In urma implementarii, rezultatele afisate respecta cerinta, afisandu-se la fiecare moment de timp situatia fiecarei cozi.

**-6- Bibliografie**

1.https://www.stackoverflow.com

2.https://www.wikipedia.org

3.https://www.geeksforgeeks.com