

MINISTERUL EDUCAȚIEI NAȚIONALE



UNIVERSITATEA TEHNICĂ

DIN CLUJ-NAPOCA

FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DEPARTAMENTUL CALCULATOARE

QUEUES SIMULATOR

Documentatie Tema 2

Elev : Țoc Roxana-Ștefania

Grupa : 30224

Profesor îndrumător : Antal Marcel

Cuprins:

1. Obiectivul temei

1.1 Obiectivul principal

1.2 Obiectivul secundar

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

3. Proiectare

4. Implementare

5. Testare

6. Rezultate

7. Concluzii

8. Bibliografie

1. Obiectivul temei

1.1 Obiectivul principal

Obiectivul acestei teme este de a proiecta si implementa o aplicatie de simulare de cozi care vizeaza analiza sistemelor bazate pe cozi pentru determinarea si minimizarea timpului de asteptare al clientilor . Principalul scop al unei cozi este de a furniza un loc pentru ca un “client” sa astepte inainte de a primi un “serviciu”. O modalitate de a minimiza timpul de asteptare este de a adauga mai multe cozi in sistem, insa aceasta abordare creste costurile furnizorului de servicii. Cand se adauga o coada noua, clientii care asteapta vor fi repartizati uniform la toate cozile disponibile. Aplicatia ar trebui sa simuleze o serie de N clienti sositi pentru un serviciu, introducerea a Q cozi , clientii asteptand, fiind serviti si in final parasesc coada . Clientii sunt generati atunci cand simularea este pornita si sunt caracterizati de trei parametri : ID, timpul de sosire in coada si timpul de procesare . Programul urmareste timpul total petrecut de fiecare client in cozi si calculeaza media de asteptare .

Datele de intrare pentru acest proiect sunt citite dintr-un fisier text :

- Numarul de clienti (N)
- Numarul de cozi de asteptare (Q)
- Interval de simulare
- Timpul minim si maxim de sosire
- Timpul minim si maxim de procesare

1.2 Obiectivul secundar

- Dezvoltarea de use case-uri si scenarii : Realizarea de simulari de cozi . Se introduc datele de intrare dorite in fisierul de intrare , programul generand rezultatul .
- Alegerea structurilor de date : Folosirea ArrayList in loc de vector pentru usurinta de utilizare .
- Impartirea pe clase : Am ales clasa Client pentru a genera informatiile despre fiecare client (ID , timpSosire, timpServire) , clasa Procesare care repartizeaza clientii la cozi , clasa Coada care contine clientii si clasa Manager care cuprinde mai multe metode care vor fi prezentate mai jos .
- Dezvoltarea algoritmilor : Pentru acest proiect avem nevoie de cativa algoritmi din matematica si nu numai pe care sa ii transpunem eficient in lumea virtuala : generarea de numere random , sortari, servirea clientilor sub forma unei cozi etc. .
- Implementarea solutiei : Prezentarea claselor si metodelor realizate .

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

Analiza problemei

Pentru a implementa o metoda cat mai corecta a acestei aplicatii, trebuie indeplinite urmatoarele cerinte aflate in enuntul temei :

- Generarea random a clientilor
- Multithreading
- Aranjarea corecta a clientilor in listele de tip coada
- Evolutia cozilor in timp real realizata prin intarzierea thread-urilor cu cate o secunda la fiecare pas
- Repartizarea clientilor in cozi in functie de timpul de sosire a fiecaruia
- Progresul cozilor prin scaderea la fiecare pas a timpului de procesare a clientilor ajunsi in coada, dar totodata decrementarea si timpului total al cozii
- Eliminarea clientilor in momentul in care timpul de procesare a clientilor ajunge la zero
- Calcularea timpului mediu de asteptare a clientilor in timp ce asteapta la coada

Pentru introducerea clientilor in cozi trebuie urmarite cateva criterii :

- Fiecare client trebuie asezat doar intr-o singura coada
- Clientul va fi asezat in coada care are timpul de asteptare cel mai mic
- Daca exista mai multe cozi cu un timp minim de asteptare, atunci clientul va fi asezat la prima coada gasit

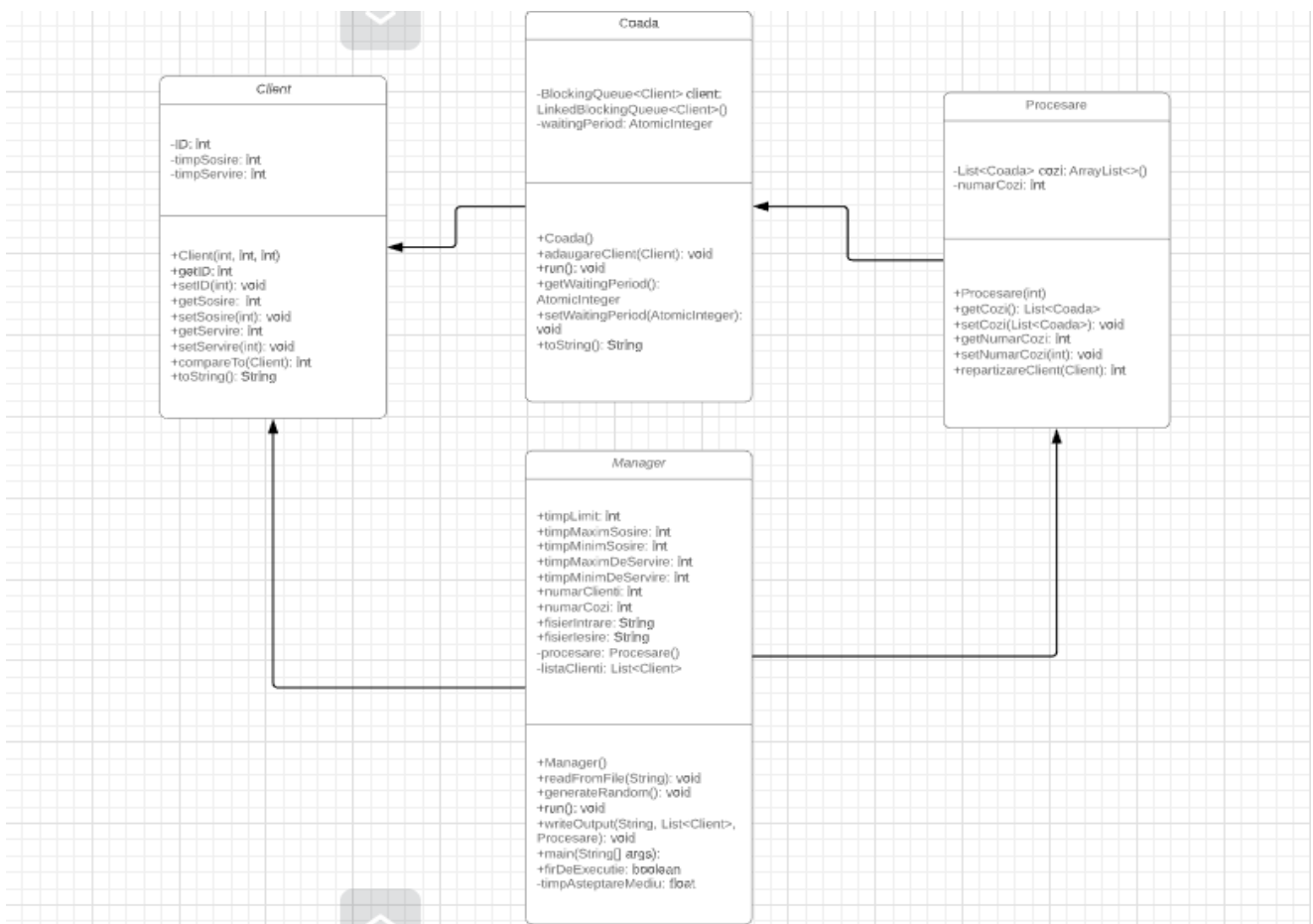
3. Proiectare

3.1 Structuri de date

Ca structura de date , am utilizat un ArrayList de client si cozi . Am lucrat cu aceasta structura , folosind functii definite in biblioteca java.util.ArrayList , cum ar fi add , remove , take , fiind mult mai usor de operat decat cu un vector .

3.2 Diagrama de clase

Unified Modeling Language (UML) este un limbaj standard pentru descrierea de modele si specificatii pentru software . Este folosita pentru reprezentarea vizuala a claselor si a interdependentelor , taxonomiei si a relatiilor de multiplicitate dintre ele . Diagramele de clasa sunt folosite si pentru reprezentarea concreta a unor instante de clasa , asadar obiecte , si a legaturilor dintre acestea .



4. Implementare

Proiectarea proiectului incepe prin definirea celor patru clase : Client , Coadă , Procesare pentru repartizarea clientilor si Manager care reprezinta clasa Main care genereaza date de iesire dupa datele citite din fisier .

Clasa **Client** are ca atribute ID-ul clientului, timpul de sosire si timpul de servire si contine metodele `getID()` , `setID()` , `getSosire()` , `setSosire()` , `getServire()` , `setServire()` pentru gettere si settere cu ajutorul carora atributele pot fi verificate si modificate in timpul procesarii informatiilor in celelalte clase , iar Constructorul `Client(int, int, int)` cu rolul de a initializa aceste atribute . Am implementat metoda `compareTo(Client)` care compara timpul de sosire a fiecarui client si returneaza 0 daca acesta este egal , 1 daca timpul primului client este mai mare decat celui de-al doilea si -1 altfel . In ultimul rand, contine metoda `toString()` care afiseaza caracteristicile fiecarui client .

Clasa **Coadă** are ca atribute : `BlockingQueue<Client>` `client` , o lista inlantuita de tip `queue` de clienti si care ne ajuta sa luam un client din coada prin functia `take()` si `AtomicInteger` `waitingPeriod` , cu ajutorul caruia o valoare int poate fi actualizata automat . Metodele `getWaitingPeriod()` si `setWaitingPeriod()` sunt pentru gettere si settere . Metoda `adaugareClient(Client)` realizeaza adaugarea clientului in coada setand si timpul de asteptare cu ajutorul timpului de procesare a clientului si totodata si timpul total de asteptare al cozii .

Aceasta clasa implementeaza `Runnable` si trebuie suprascrisa metoda `run()` . La inceput pornim o bucla care pare ca ruleaza la infinit, insa pentru oprirea ei am luat o variabila de tip `boolean` in clasa `Manager` care este initializata cu `true`. In momentul cand aceasta variabila devine `false` realizeaza oprirea `thread`-ului . Atunci cand trece un semnal de timp , timpul de procesare al clientilor este scazut cu o unitate , iar la fel si timpul de asteptare al cozii va scadea cu o unitate . Atunci cand timpul de procesare a unui client ajunge la zero, acesta este scos din coada cu ajutorul functiei `take()` . In final am suprascris metoda `toString()` pentru afisarea clientilor din fiecare coada sau pentru afisarea mesajului “closed” in momentul in care in coada nu exista nici un client .

Clasa **Procesare** are ca atribut o lista `List<Coadă>` `cozi` , care contine cozile si un alt atribut care inregistreaza numarul de cozi . Totodata , am realizat si metodele de `getter` si `setter` pentru aceste atribute: `getCozi()`, `setCozi(List<Coadă>)`, `getNumarCozi()`, `setNumarCozi(int)`. Constructorul `Procesare(int)` are rolul de a initializa aceste atribute si totodata se adauga in lista un numar de cozi si pentru fiecare coada se porneste un `thread` .

Ultima metoda implementata in aceasta clasa este `repartizareClient(Client)` care parcurge lista care cuprinde cozile si repartizeaza un client la coada cu timpul de asteptare minim . In cazul in care exista doua cozi cu acelasi timp de asteptare , clientul va fi plasat in cea care a fost gasita prima. Aceasta metoda returneaza un `int` care este timpul de asteptare total al cozii in care a fost adaugat clientul, insa dupa adaugarea acestuia. Pentru calcularea timpului mediu de asteptare vom avea nevoie de aceasta valoare.

Clasa **Manager** are ca atribute : `numarClienti` , `numarCozi`, `timpSimulare`, `timpMaximSosire`, `timpMinimSosire`, `timpMaximDeServire`, `timpMinimDeServire`, doua `Stringuri` `fisierIntrare` si `fisierIesire` si o lista care cuprinde clientii . De data aceasta constructorul `Manager()` este gol deoarece valorile vor fi citite din fisierul de intrare. Acest lucru se realizeaza in metoda `readFromFile(String)` care citeste de pe prima linie din fisier numarul de clienti, pe a doua numarul de cozi, apoi timpul de simulare, iar de pe liniile patru si cinci separate prin virgule, timpul minim de sosire si timpul maxim de sosire , respectiv timpul minim de procesare si timpul maxim de procesare .

Metoda `generateRandom()` genereaza pentru fiecare client in mod aleator timpul de sosire si procesare intre limitările citite din fisier, apoi clientii sunt introdusi in lista de clienti unde se seteaza ID-ul clientului, timpul de sosire si timpul de procesare . In final fiecare client este adaugat in “listaClienti”, dupa care se realizeaza sortarea acestora in functie de timpul de sosire in coada .

Aceasta clasa implementeaza `Runnable` este nevoie din nou de suprascrierea metodei `run()`. Am luat o variabila “currentTime” pe care am initializat-o cu 1 si pe care o comparăm cu timpul de simulare citit din fisierul de intrare . Aceasta variabila va fi incrementata la fiecare secunda , iar atunci cand momentul curent este mai mare decat timpul de simulare, executarea se opreste. La fiecare iteratie a `while`-ului se parcurge lista de clienti cautand clientii care au timpul de asteptare egal cu timpul curent al firului de executie . Atunci cand gaseste un astfel de client , acesta este plasat la una dintre cozi , dupa cum am explicat in clasa `Procesare` . Dupa actualizarea cozii, in variabila `timpMediuDeAsteptare` se va adauga timpul returnat de metoda `repartizareClient`. Intr-o lista de tip `Integer`

se va pastra indexul fiecarui client care a fost adaugat in coada . Dupa ce clientii care au timpul de sosire egal cu timpul firului de executie au fost adaugati in coada , acestia vor fi stersi din lista clientilor care se afla in asteptare prin indexul memorat anterior . Dupa stergere se afiseaza actualizarea cozii de asteptare si evolutia cozilor . Pentru a fi siguri ca thread-ul este sincron cu cel din clasa Coada , cu ajutorul metode sleep() vom opri programul timp de o secunda .

Dupa ce timpul de simulare ajunge la final , este afisat si timpul mediu de asteptare aflat prin impartirea variabilei timpAsteptareMediu calculate mai sus , la numar de clienti . Tot acum se realizeaza si oprirea firelor de executie prin setarea variabilei boolene firDeExecutie pe false .

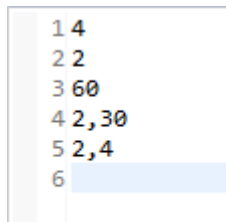
Metoda de scriere in fisier afiseaza lista clientilor ramasi in coada si care nu au fost asezati pana in momentul de fata in una dintre cozi . Apoi urmeaza cozile in ordinea generarii lor cu textul “closed” atunci cand in coada nu exista nici un client sau clientii aflati in coada cu timpii de procesare corespunzatori fiecarei generari .

In metoda main() este creat un obiect nou de tip Manager si ii sunt atribuite fisierele de intrare si iesire care sunt memorate in argumentele 0 si 1. In continuare , este creat si programul cozilor prin instantierea variabilei procesare si este apelata metoda care genereaza random clientii si ii pune in lista de clienti . In final , dupa initializarea acestor date , se creaza un fir de executie si este pornit in executie prin comanda start() .

5. Rezultate

Atat fisierul de intrare , cat si cel de iesire respecta o forma predefinita asa cum a fost prezentata mai sus . Voi atasa doua imagini care vor prezenta aceste detalii

Fisier de intrare :



```
1 4
2 2
3 60
4 2,30
5 2,4
6
```

Fisier de iesire

Time 0

Waiting clients: (4,4,2); (1,12,2); (2,13,2); (3,14,2);

Queue 1: closed

Queue 2: closed

Time 1

Waiting clients: (4,4,2); (1,12,2); (2,13,2); (3,14,2);

Queue 1: closed

Queue 2: closed

Time 2

Waiting clients: (4,4,2); (1,12,2); (2,13,2); (3,14,2);

Queue 1: closed

Queue 2: closed

Time 3

Waiting clients: (4,4,2); (1,12,2); (2,13,2); (3,14,2);

Queue 1: closed

Queue 2: closed

Time 4

Waiting clients: (1,12,2); (2,13,2); (3,14,2);

Queue 1: (4,4,2);

Queue 2: closed

Time 5

Waiting clients: (1,12,2); (2,13,2); (3,14,2);

Queue 1: (4,4,1);

Queue 2: closed

....

Avarange waiting time: 2.0

6. Concluzii

Concluzia principala pe care am dedus-o in urma realizarii acestei teme este faptul ca pentru o mai buna gestionare a memoriei calculatorului este indicat sa se foloseasca structuri de date de memorare in liste in loc de vectori. Astfel consumul memoriei este mult mai mic, desi timpul de acces este mai ridicat.

7. Bibliografie

- http://coned.utcluj.ro/~salomie/PT_Lic/4_Lab/Assignment_2/Assignment_2_rev.pdf
- http://coned.utcluj.ro/~salomie/PT_Lic/4_Lab/Assignment_2/Java_Concurrency.pdf
- <http://java2novice.com/java-collections-and-util/>

