QueuesSimulator

FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE

COORDONATOR PROIECT:DORIN MOLDOVAN

STUDENT: Rus Mihai-Tudorel

GRUPA : 30211

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

CUPRINS

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

I. Specificație

- Cerința

Proiectarea și implementarea unei aplicații de simulare care vizează analizasistemelor bazate pe cozi pentru determinarea și minimizarea timpului de așteptare al clienților.

Cozile sunt utilizate în mod obișnuit pentru modelarea domeniilor din lumea reală. Principalul obiectiv al unei cozi este să ofere un loc în care un „client" să aștepte înainte de a primi un „serviciu". Gestionarea bazată pe coada incearca să minimizeze timpul în care „clienții" lor așteaptă la coadă înainte de a fii serviti.

- Analiza problemei

Problema se împarte în mai mulți pași după cum ne și sugerează cerința. Un mod de a minimiza timpul de așteptare este să adăugați mai multe servere, adică mai multe cozi în sistem (fiecare coadă este considerată ca având un proces asociat), dar această abordare creștecosturile furnizorului de servicii. Când se adaugă un server nou, clienții în așteptare vor fi uniform distribuiti la toate cozile disponibile curente.

Aplicația ar trebui să simuleze (prin definirea unui timp de simulare $tsimulation$) o serie de N clienți, sosirea serviciului, intrarea în cozi Q, așteptarea, a fi servit și, în final, părăsirea cozi.Toti clienții sunt generați la începerea simulării și sunt caracterizați de trei parametri: nrClient(un număr între 1 și N), ajuns (timpul de simulare când sunt gata să meargă la coadă; adică ora la care clientul a terminat cumpărăturile) și servit(intervalul de timp sau durata necesară pentru a fii servit clientul de către casier; adică timpul de așteptare când clientul este în fața cozii).

```
public class Client {
    protected int nr;
    private int ajuns;
    private int servit;

    public Client(int nr, int ajuns, int servit) {
        this.nr = nr;
        this.ajuns = ajuns;
        this.servit = servit;
    }
```

II. Proiectarea și implementarea

Pentru inceput, pentru a genera un numar random de clienti am avut nevoie sa implementez o functie in interiorul MainClass, care se foloseste de array list pentru a creea un sir de clienti de la i=0 pana la i= nrClienti, utilizand java.util.ArrayList.

Pentru a putea citii din fisiere,respectiv In-Test-1.txt,In-Test-2.txt,In-Test-3.txt,m-am folosit de clasa Scanner, unde am utilizat un bloc de tip try,catch pentru a putea prinde o eventuala eroare la transmiterea caii fisierului input.Pentru a putea afisa intr-un fisier text, respectiv Out-Test-1.txt,Out-Test-2.txt,Out-Test-3.txt, am folosit obiect de tip Formatter,unde la fel am utilizat blocul try,catch dupa cum se poate observa in imaginea de mai jos:

```
public void openInputFile() {
    try {
        cititor = new Scanner(new File("In-Test-2.txt"));
    } catch (Exception e) {
        System.out.println("ERROR: Fisierul nu se poate deschide.");
    }
}

public void openOutputFile() {
    try {
        format = new Formatter("Out-Test-2.txt");
    } catch (Exception e) {
        System.out.println("ERROR: Fisierul nu se poate deschide.");
    }
}
```

Motivul pentru care am folosit obiecte de tip Scanner, a fost acela de a putea parcurge int cu int fisierul de intrare. Aplicația citește din In-Test.txt parametrii de simulare:

N = 4 clienți, Q = 2 cozi,

MAX= 60, o simulare de 60 de secunde

De asemenea, citește limitele pentru parametrii clientului,respectiv ora minimă și maximă de sosire 2, 30, adică clientii vor merge la cozi de la a doua 2 până la a 30 a.În plus, limitele timpului de serviciu sunt citite din linia 2 și 4, ceea ce înseamnă că un client are un timp minim pentru a aștepta în fața cozii 2 secunde și un timp maxim de 4 secunde.

Fisierul de iesire contine urmatoarele:numarul de clienti care au stat la cozi,numarul de cozi,timpul maxim de simulare,timpul minim de ajungere,cel maxim de ajungere,timpul minim de servire si timpul maxim de servire,precum si lista clientilor cu fiecare identificator in parte.

Indeosebi, clasa Input, contine nrClients,Q-numarul de cozi,timpul maxim de simulare,timpul de ajungere,timpul de servire,precum si un toString care genereaza toate operatiile facue pe fisierele de output,dar si gettere si settere pentru fiecare camp in parte.

Clasa Server incapsuleaza constructorul si metodele addClient,updateClient,si removeClient,in care actualizam de fiecare data timpii de ajungere si servire.

Clasa Server deține o listă de clienți pe care o prelucrează, timp în care un client potențial va trebui să aștepte coada și timpul total de așteptare a tuturor clienților au trecut pe server. Toate acestea sunt private, așa vor face să fie accesat folosind getters și setters.Metoda updateServer este utilizată pentru a urmări timpul de simulare și, de asemenea, timpul rămas până la terminarea clientilor la coadă.

Clasa RunSever suprascrie metoda Runnable din clasa Run, folosind override si mentionand comportamentul de sleep al metodei run. Această clasă conține serverul pe care îl rulează și se actualizează constant și două variabile booleane destinate manipularii fluxului programului, pauză și oprire (din moment ce „cozile ar trebui să se deschidă / să se închidă dinamic. Initial toate cozile sunt închise. Când clienții sunt distribuiți la cozi, aceștia devin creati, după cum este necesar. Când o coadă devine goala, se trece la alta,prin alocare de memorie si stergerea celei de dinainte.Metoda run () asigură că datele serverului sunt actualizate de fiecare dată și că firul serverului va întrerupe /se va opri în funcție de situația dată.

Metoda de rulare crește continuu timpul de simulare curent și verifică pentru a vedea care clienți sunt eligibili pentru a fi adăugati la unul dintre serverele noastre de coadă. De asemenea, metoda verifică în mod constant dacă au fost clienți adăugati la o coadă și, de asemenea, dacă le-am finalizat procesarea iar apoi am apelat pentru a opri toate thread-urile serverului și metoda sistemului de ieșire (0) este apelată pentru a ieși din thread-ul curent.

```
@Override
public void run() {
    while (quit==false) {
        while(standby==true)
            try {
                sleep(100);
            } catch (InterruptedException e) {
                System.out.println("ERROR: sleep");
            }
        }
        server.update();
    }
}
```

Clasa Simulator joacă poate cel mai important rol în programul nostru, deoarece gestionează execuția și direcționează datele către diferite obiecte în funcție de nevoile noastre. Clasa conține două liste de tablouri cu cele disponibile,serverele și clienții care urmează să fie prelucrați. Pentru ușurința punerii în aplicare, folosim și numărul de clienți pe care trebuie să-i procesăm încă de la început, deoarece dimensiunea listei va scădea în timp și inițial lungimea va fi pierdută. Se poate observa metoda allClients care este utilizată în ordine pentru a updata de fiecare data lista de clienti .Termenul de simulare este folosit atunci când noi avem mult mai mulți clienți decât serverele, iar procesarea tuturor acestora va dura mai mult timp.Astfel, simularea se va opri după atingerea termenului de simulare;

Asadar, in clasa Simulator am definit o lista de servere,o lista de clienti si timpii de simulare pentru fiecare coada in parte.Totodata, am calculat si media timpului de stat la coada pentru clienti si media timpului de servire a clientilor.

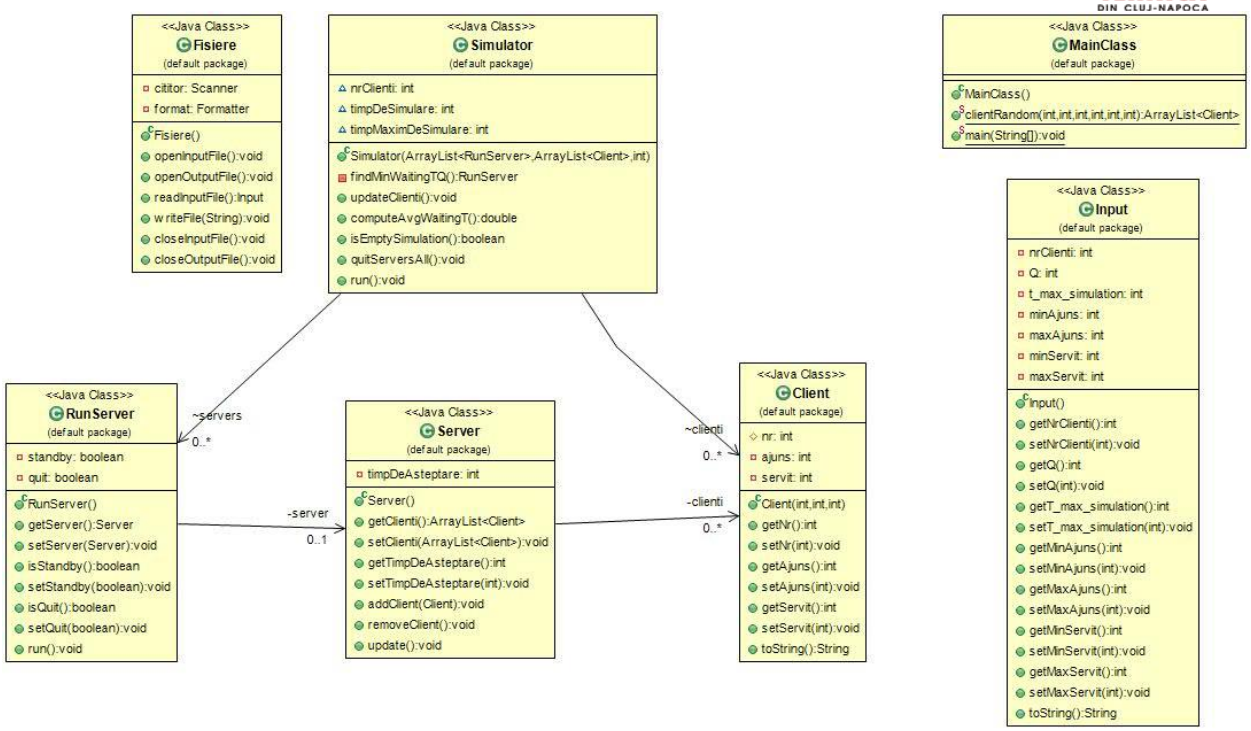
```
public double computeAvgWaitingT() {
    double timpAsteptare = 0;

    for (RunServer server : this.servers) {
        timpAsteptare = timpAsteptare+server.getServer().getTimpDeAsteptare();
    }

    timpAsteptare = timpAsteptare/nrClienti;
    return timpAsteptare;
}
```

In MainClass, am reusit sa implementez o lista de threaduri si o lista de servere,care m-au ajutat sa dau start la simularea propriu-zisa a programului, cu generarea random a clientilor si a timpilor de servire si sosire,precum si sortarea clientilor dupa timpul de sosire la coada,iar apoi am simulat fiecare coada in parte cu un thread separat pentru fiecae coada.La inceput am alocat memorie un sir de cozi,iar mai apoi m-am folosit de un thread pentru a procesa coada, iar la final am golit fiecare coada cu apelul clienti.remove(i);

III. Lista de clase

Pentru a respecta principiile fundamentale ale OOP, programul conține mai multe clase:

- Client
- Input
- Fisiere
- Server
- RunServer
- Simulator
- MainClass



IV. Justificarea soluției alese

Am optat pentru această soluție de implementare deoarece mi s-a părut o modalitate atât ușoară, cât și eficientă pentru implementarea aplicatiei. Folosirea stărilor intermediare prin care trece programul ne ajută atât la simplificarea soluției, cât și la implementarea eficientă a aplicatiei.

V. Utilizarea și rezultatele

Pentru a putea rula programul, am incercat utilizarea fisierului de tip jar, dar nu am reusit sa apelez din consola, asa ca am fost nevoit sa fac totul in interiorul clasei fisiere unde specific de fiecare data de unde sa preiau informatiile pentru input si unde sa le afisez pentru output:

```
public void openInputFile() {
    try {
        cititor = new Scanner(new File("In-Test-1.txt"));
    } catch (Exception e) {
        System.out.println("ERROR: Fisierul nu se poate deschide.");
    }
}

public void openOutputFile() {
    try {
        format = new Formatter("Out-Test-1.txt");
    } catch (Exception e) {
        System.out.println("ERROR: Fisierul nu se poate deschide.");
    }
}
```

VI. Posibilități de dezvoltare ulterioară

Consider ca, o dezvoltare ulterioara ar putea fii aceea de a adauga si o simulare in timp real a cozilor,nu in timp masurat in secunde ci, simularea cat mai realistica, cu aparitia clientilor la aceleasi momente de timp,cu intercalarea timpului,practic o aplicatie de simulare apropiata cu realitatea.

VII. Concluzii

Concluziile includ faptul că o planificare bine gândită a claselor ușurează lucrurile și economisește timp. Odată cu creșterea dificultății în proiecte, trebuie abordată problema cu o perspectivă inteligentă și nu doar să sari imediat la scrierea codului. Astfel, începând cu diagramele UML mi-am dat timp să reflectez pentru abordarea adecvată a implementării programului dorit.

Poate că cea mai interesantă tehnică pe care am învățat-o din acest proiect este utilizarea threadurilor și a multithreading-ului, care s-a dovedit a fi uimitor de util și puternic. Lucrul cu threaduri este cu siguranță ceva ce mi-aș fi dorit să știu mai devreme.

Îmbunătățirile care ar putea fi realizate în cadrul proiectului includ o ușoară modificare, astfel încât calculatorul acceptă intrări float, care ar putea fi ușor realizate prin modificarea matcherelor regEx din funcție care transformă intrarea utilizatorului șirului în obiectul respectiv. Într-o anumită măsură...

VIII. Bibliografie

http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

http://www.tutorialspoint.com/java/util/timer_schedule_period.html

http://www.javacodegeeks.com/2013/01/java-thread-pool-example-using-executors-and-threadpoolexecutor.html

http://coned.utcluj.ro/~salomie/PT_Lic/4_Lab/Assignment_2/Assignment_2_rev.pdf

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA