FUNDAMENTAL PROGRAMMING TECHNIQUE

ASSIGNMENT 2

QUEUES SIMULATOR

Diaconu Andrei
Grupa 30227

CUPRINS

1 Introducere

Threadul (fir de exectuie) este executat de sistemul de operare si reprezinta cea mai mica unitate de procesare . Acesta executa portiuni de cod distincte , in paralel si in interiorul aceluiasi proces. Totusi exista cazuri in care thread-urile intr-un scop comun comunica intre ele , in sensul ca unul este nevoit sa-l astepte pe celalalt , iar acest lucru este definit in Java ca sincronizarea threadurilor.

Obiectivul principal al temei este proiectarea si implementarea unei aplicatii de gestionare a unui simulator de cozi. Acest simulator de cozi ar putea beneficia de o interfata grafica (GUI) care ne arata in timp real gestionarea cozilor care practic pot reprezenta niste case de marcat dar si lista de clienti aflata in coada de asteptare .

Corelarea celor doua idei de thread si de simuluator este reprezentata de faptul ca fiecare coada (casa de marcat) este un fir de executie , gestionate la nivel de global de un alt fir de exectuie din care se apeleaza Frame-ul (daca este cazul) si face toate legaturile necesare . Totodata in obiectivul temei regasim , referitor la sincronizarea threadurilor , evitarea cu cat mai mult posibil a cuvantului synchronize. Asadar intru realizarea acestei conditii , trebuie sa folosim Colectii si tipuri de date Thread Safety de care vom vorbi in cele ce urmeaza.

Asadar pentru dezbaterea intregii simulari de cozi am creat o interfata graffica (Package Simulare -> Clasa View ..), interfata grafica este creata si gestionata de acel fir de executie despre care vorbeam si se regaseste la fel (in Package Simulare -> Clasa Simulation). Referitor la organizare , la elementele de care avem nevoie si anume de client am construit o clasa (Package Prelucrare-> Clasa Customer) , o coada (o

casa de marcat) (Package Prelucrare -> Clasa Queue) iar modul de atribuire al clientilor la cozi prin alta clasa (Package Prelucrare -> Scheduler).

2 Analiza problemei

Cand ma gandescs la analiza problemei , ma refer la ideea principala , schitata in mare , care functioneaza pe baza unor inputuri abstracte. Prin intermediulor lor gasim comportamentul problemei si cum ar trebui sa decurga rezovlarea.

Odata regasit firul principal al aplicatiei trebuie sa avem in vedere ca aplicatia ce o implementam va fi accesibila publicului larg de utilizatori (unii pot fi mai obisnuiti cu tehnologia decat altii). Asadar design-ul interfetei trebuie sa fie unul clar si comunicarea intre utilizator si interfata una simplista.

Pe baza acestui principiu am decis ca tot ce trebuie sa faca utilizatorul este de a introduce in gap-uri niste numere , sub forma de text. Fiecare gap avand in fata o descriere concisa.

In mare ideea este ca un client cand ajunge in fata cozii de asteptare doreste sa fie repartizat la o casa de marcat cu cel mai mic timp de asteptare. Pentru aceasta am creat simulatorul de cozi care:

- Permite unui utilizator sa introduca datele de intrare
- Calculeaza pentru fiecare casa de marcat timpul de asteptare
- La momentul oportun daca este o casa libera va fi trimis acolo, insa daca desi era timpul ca el sa mearga si toate casele sunt ocupate , acesta asteapta pana cand casa cu cel mai mic timp se elibereaza

- Utilizatorul vede desfasuarea in timp real , cu un current time afisat , care reprezinta momentele de timp in jurul careia se gestioneaza clientii.

Exista cateva scenarii de utilizare a acestui utilizabil, in functie de inputuruile utilizatorului . Cel mai propice caz ar fi ca datele sa fie introduse corect , clientii se pun la coada de asteptare , casele de marcat gestioneaza la momentul de timp in functie de cum sunt de libere abosult toti clientii , iar cand nu mai sunt clienti in asteptare si cei aflati in momentul respectiv la casa si au terminat timpul de porcesare acestea se inchid.

Insa, exista si cazul in care timpul de simulare introdus de utlizator este mai mic decat ar fi avut nevoie clientii sa finalizeze tot. Cand timpul de simulare se scurge clientii raman la casele de marcat fara sa li se mai scada timpul de procesare(ca si cand casele s-ar fi inchis) iar clientii din coada de aseptare nu vor mai ajunge niciodata la casa de marcat.

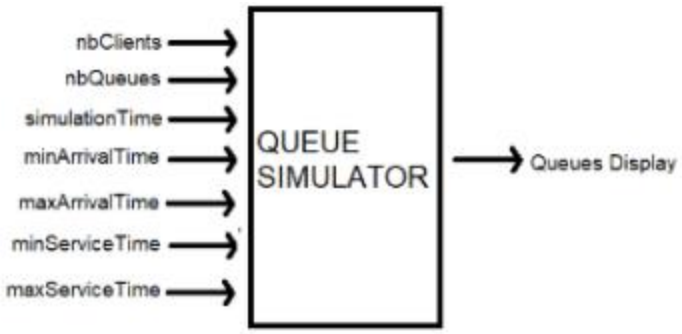
Tot la tratare de erori am putea trece si cazul in care utilizatorul cand introduce datele de intrare , la cele referitoare la intervalul de arriveTime si processTime minmul sa fie mai mare ca si maximul , in acel caz vom primi o eroare la generarea clientilor.

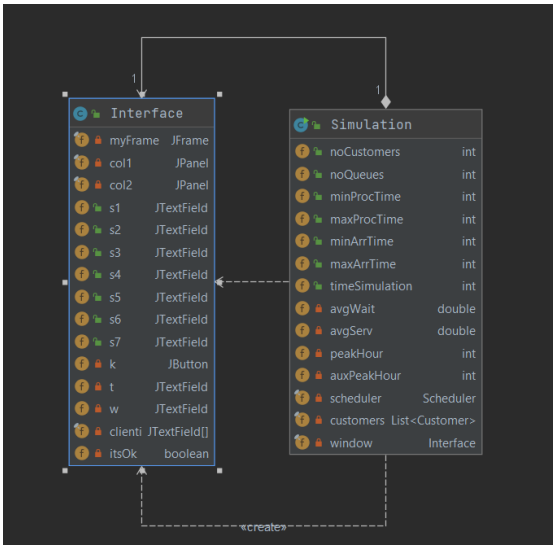
3 Proiectare si Implementare

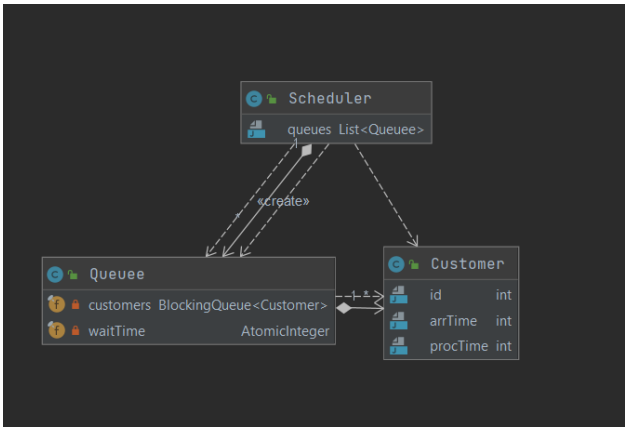
Proiectul contine 2 pachete cu nume conclusive. Aceste la randul lor contin cate 2 respectiv 3 clase. Toate clasele repsecta paradigmele OOP , conventiile Java si nu s-a depasit limita de linii referitor la clase si metode.

Pe parcursul discutarii fiecarei clase voi specifica ce tipuri de date am folosit si de ce. Pana atunci voi prezenta modul proiectarii si structura intreg programului printr-o diagrama UML.

Simulatorul este reprezentat de 7 intrari , care sunt asteptate sa fie introduse. In momentul introducerii lor , vor fi lansate q cozi, unde q reprezinta al doilea field introuds de utilizator . Toata aceasta executie se va opri cand N (primul field introuds de utilizator) s-au terminat de procesat , sau cand timpul simularii (al treilea field) a depasit timpul curent . Timpul curent incepe de la 0 si reprezinta o secunda din viata noastra. La fiecare secunda GUI-ul se updateaza si ne arata gestionare clientilor in timp real


nbClients
nbQueues
simulationTime
minArrivalTime
maxArrivalTime
minServiceTime
maxServiceTime
QUEUE SIMULATOR
Queues Display


Interface
myFrame JFrame
col1 JPanel
col2 JPanel
s1 JTextField
s2 JTextField
s3 JTextField
s4 JTextField
s5 JTextField
s6 JTextField
s7 JTextField
k JButton
t JTextField
w JTextField
clienti JTextField[]
itsOk boolean
Simulation
noCustomers int
noQueues int
minProcTime int
maxProcTime int
minArrTime int
maxArrTime int
timeSimulation int
avgWait double
avgServ double
peakHour int
auxPeakHour int
scheduler Scheduler
customers List<Customer>
window Interface
1
1
«create»


Scheduler
queues List<Queuee>
«create»
Queuee
customers BlockingQueue<Customer>
waitTime AtomicInteger
Customer
id int
arrTime int
procTime int

CLASA INTERFACE

La o primulare rulare a programului asa arata Interfata Simulatorului de cozi. M-am decis sa folosesc layout de GridLayout cu o linie si doua coloane, pe principiul ca prima coloana va determina inputurile si va comunica cu utilizatorul , iar cea de a doua coloana , cea din dreapta , reprezinta simularea in sine , care urmeaza a fi vizualizata

Referitor la partea de inputuri a acestei probleme , adica la JPanel din stanga pe care l-am numit col1, l-am impartit in 6 flowlayouturi . Aceste contin la randul lor cate cate un JPanel ce contine un JLabel care ne ofera informatii referitoare la ce urmeaza sa introduca utlizatorul si un JTextField de putine caractere deoarce sunt simple numere . Aceste Labe-luri sunt aliniate la stanga mereu fata de Textfield.

In partea inferioara a interfetei observam butonul verde tot sub forma de layout transmis(Layoutul 6) care declanseaza simularea intregii probleme . Cu ajutorul acestuia sunt transmise datele de la client la interfata , cu un ActionerListener apelat din Simulation . Tot aici in Interface am clasa itsOk care practic ne zice daca a fost apasat butonul OK sau nu .

Toate aceste campuri pot fi setate din Simulation , de aceea am facut clasele publice setT , setW,SetN,etc. Care ajuta mai mult la partea de output. In constructorul acestei clase pe langa intierea JFrame-ului si a structurii gridlayou exista alte 2 metode si anume input() si output()

Acestea determina de fapt ca fereastara sa arate asa . Ambele au o parte de CSS , in care seteaza bordurile flowlayoutului si culoarea de fundal. Outputul insa este afisat complet doar in momentul in care butonul OK a fost apasat, deoarce atunci incepe simularea prorpiu zisa

Partea de afisare a cozilor a fost mai complexa si am uitilizat un array de JPanel-uri bazat pe numarul de cozi primit ulterior inceperii simularii , mai apoi pe baza clientilor se afisa continului ei. Daca continutul era gol , trebuia sa fie afisat CLOSED.

CLASA SIMULATION

Aceasta clasa se comporta are intr-un fel sau altul cateva functii ce controler deoarce se ocupa de gestionarea textfiled-urilor dar si a singurului buton. De aici incepe executia totala a simularii.

In constructor se intiliazeaza la primul pas fereastra de GUI , iar apoi intra intr-un while infinit , care are scopul de a astepta citirea datelor . In momentul cand datele au fost introduse , utilizatorul trebuie sa apese butonul OK , acesta prin 2 metode din Interface reuseste sa scoata simularea din acel while infinit. In acest moment al problemei datele sunt citite si retinute in filed-urile acestei clase.Tot aici creez lista noua de clienti, si ba mai mult si ii generez.

Generarea clientilor este una din cerintele problemelor , sa fie realizata Random.De aceea am si folosit aceasta functie Random care pe

baza bound-urilor de minim si maxim introduse de utilizator va genera 2 numere random , unul va reprezenta arriveTime-ul , iar celalat processTime-ul. Odata generate, le pot adauga in lista nou creata. Dupa ce au fost creati noCustomers clineti , pasul urmator este letal , deoarce este foarte important ca acesti cilenti sa fie sortati in functie de timpul lor de arrive. Acum acestia se afla in coada de aseptare in oridine crescatore .

Pe baza cozilor am creat si Scheduler-ul , depsre care voi vorbi in cele ce urmeaza.

Aceasta clasa implementand metoda Runnable are o functie suprsa scrisa run(). Aici practic se gestioneaza toata simularea problemei. Initiez un currentTime , sa stim mereu in ce moment de timp ne aflam. Si atat timp cat noi ne aflam intr-un timp mai mic decat maximul admis (time simulation), simularea se executa continuu

Odata intrat inauntru acestui while , intalnim altul care verifica daca sunt clienti in lista de asteptare, daca timpul curent permite clientului aflat in fata la coada de asteptare sa fie repartizat la o casa de marcat si daca exista cozi libere , in caz afirmativ acesta este introdus si sters din lista clientilor de asteptare. Locul lui este preluat de urmatorul client cu cel mai mic timp de sosire , acestia fiind ordonati inca din constructor.

Analizand situatia mi-am dat seama ca omitem cazul in care un client aflat la o casa de marcat se afla in ultima secunda , primul client din lista de asteptare ar trebui sa I ia locul instant, nu pierzand o secunda. Privind astfel problema daca nu intra in while-ul de mai sus si intra in acest while care verifica daca mai sunt clienti , daca timpul curent permite clientului si daca cumva o casa de marcat mai are o secunda de asteptat , atunci clar a fost cauza cozilor pline si il introducem imediat dupa ce se scurge acel moment de timp. Inainte sa incrementam timpul curent , am hotarat sa verific daca nu cumva lista de

clineti e goala si cozile inchise , pentru ca aceasta situatie ar trebui sa ne scoata de tot din simulare.

De-a lungul acestui run() am mai implementat cele cerute si anume scrierea intr-un fisier si calcularea unor averageuri. Scriere in fisier am facut o folosind FileWriter si PrintWriter, deoarce trebuia sa folosesc modul append, fiind fisierul .txt updatat la fiecare pas al simularii. Insa daca eu vreau sa rulez inca odata , as dori sa se stearga simularea dinainte , in acest scop am deschis fisierul la inceputul metodei run() si am introdus un String gol .

Referitor la average-uri : avgWaitTime l-am gandit ca si timpul care il petrece fiecare client in fata cozii pana sa ajunga la o anumita casa de marcat . Daca nu a stat deloc in fata cozii va avea timpul de asteptare 0. In acest fel , dupa o analiza complexa am realizat ca problema timpul mediu de asteptare se reduce la gasirea timpului in care nu mai exista niciun client in asteptare. Acel prim moment de timp , imparit la numarul de clienti ne gereaza acest timp. Daca totusi timpul simularii a fost depasit , avem un caz mai special , acesta prevazand o impartire a intregii perioade de simulare la numarului de clienti satisfacuti.

AvgService time reprezinta timpul mediu de servire a tutoror clientilor serviti. Aici trebuie sa fim atenti la faptul ca doar la clientii serviti , in cazul terminarii timpului de simulare unii clienti nu au fost serviti asadar nu are rost sa I socotim la timpul mediu de procesare.

PeakHour , dupa explicatii nu reprezinta ora cu cei mai multi clienti, ci ora la care daca un client ar veni ar trbeui sa astepte cel mai mult . Pe baza acestui fapt am decis ca la fiecare moment de timp sa adun timpii de procesare ai fiecarui client aflat in fata casei de marcat . Dintre acestea alegem maximul si aceea este ora de varf. Toate aceste vor fi afisare la sfarsiutul .txt-ului.

```
Time:25
WAITING CLIENTS:
QUEUE 1: (1 25 4);
QUEUE 2: (2 25 2);

Time:26
WAITING CLIENTS:
QUEUE 1: (1 25 3);
QUEUE 2: (2 25 1);

Time:27
WAITING CLIENTS:
QUEUE 1: (1 25 2);
QUEUE 2: CLOSED

Time:28
WAITING CLIENTS:
QUEUE 1: (1 25 1);
QUEUE 2: CLOSED

Time:29
WAITING CLIENTS:
QUEUE 1: CLOSED
QUEUE 2: CLOSED
AVG WAIT 6.5
AVG SERV 2.5
PEAK HOUR 25
```

CLASA SCHEDULER

Ca field aceasta clasa contine doar lista de cozi(case de marcat). Desi contine doar 2 metode face legatura intre cele 2 elemente principale ale programului si anume Cusomer si Queuee.

Constructorul acestei clase instantiaza toate cozile in functie de numarul primit ca si argument din Simulation. Tot aici se si pornesc fierele de exectuie reprezentative fiecarei cozi.

Metoda attCostumer este o combinatie a adaugarii simple a unui Customer in lista-field din Simualtion dar si a metodeti addCustomer aflata in Clasa cu numele Queuee.Foloseste Attomic Integer depsre care vom discuta in cele ce urmeaza. Acest minim va reprezenta dupa parcurgerea tuturor cozilor , timpul de asteptare cel mai scurt al caselor de marcat. Unde se gaseste acest timp , la o noua parcurgere , se introudce noul client iar mai apoi se paraseste intreaga metoda.

```java
public Scheduler(int noQueues) {
    queues = new LinkedList<>();
    for (int i = 1; i <= noQueues; i++) {
        Queuee q = new Queuee();
        queues.add(q);
        Thread t = new Thread(q);
        t.start();
    }
}
```

```java
public void attCustomer(Customer customer){
    AtomicInteger min = new AtomicInteger( initialValue: 100);
    for(Queuee q : queues)
        if(q.getWaitTime().get()< min.get())
            min.set(q.getWaitTime().get());
    for(Queuee q : queues)
        if(q.getWaitTime().get() == min.get()) {
            q.addCustomer(customer);
            break;
        }
}
```

CLASA CUSTOMER

Aceasta Clasa desi preteaza ca cea mai simplista dintre ele prin 3 filed-uri si niste gettere , reprezinta partea de baza a afisarii atat in log.txt cat si pe interfata grafica .

Constructorul care este apelat in Scheduler ii ofera clientului un id (nu este voie sa lispeasca un id in sensul ca trebe sa fie un interval inchis de numere intregi de la 1 pana la N) , timpul de sosire pe baza cauria se sorteaza si se dispune coada de astepare si un timp de procesare care urmeaza a tine o casa de marcat ocupata .

Pe langa gettere , se observa suprascrierea metodei toString. Aceasta reprezinta aspectul grafic al fiecarui client , deoarce contine toate cele 3 campuri binedefinite , inconjurare de paranteze rotunde.Returneaza , normal , un String care urmeaza sa fie afisat ori in JTextFiled ori in log.txt , exact cum am mai spus.

```java
@Override
public String toString() { return " ("+id+" "+arrTime+" "+procTime+");"; }
```

CLASA QUEUEE

In aceasta clasa vom descire modul de compartare a unei case de marcat, locul unde clientii din coada de asteptare urmeaaza sa fie procesati. Prezinta 2 fileudri cu un tip de date mai diferit decat cele obisnuite si anume BlockingQueue si Atomic Integer

Respectand o cerinta principala a acestui simulator de cozi , sa evitam cu cat mai mult posibil folosirea cuvantului cheie Synchronize , am folosit tipuri de date ThreadSafety. Acestea nu pot fi apelate deodata din doua parti diferite , din doua fire de executie diferite.

Prin urmare in constructor le-am intializat, iar in addCustomer despre care vorbeam mai sus, pe langa faptul ca introduce un element in listsa mea inlantuita , prelucreaza waitTime-ul care este acum exact timpul de procesare al clientului venit in fata la casa de marcat.

IsBusy() este o metoda care este reprezntata de un simpla interogare referitoare la dimensiunea customerilor acelei case . Mai exact daca casa de marcat nu are niciun client va returna flase , deoarce casa nu este ocupata , in caz contrar true , si cel mai probabil nu va putea introudce momentan niciun client.

La fel ca si Simulation aceasta implementeaza Runnable asadar suprascrie metoda run(). O conditie principala ajunsi in acest punct este ca lista de clienti sa nu fie goala , daca este goala in run() nu trebuie sa se execute nimic , nu incarcam firul de exectuie cu instructiuni inutile. Totusi in cazul in care exista , deci este un client la casa de marcat , aici este locul in care acesta este extras cu functia peek() , care nici nu-l sterge spre deosebire de .take(). Iar Thread-ul este bagat in sleep timp de o secunda care va permite timpului curent din Simuluation sa decrementeze . De aici se face decrementarea timpului de procesare al unui client , si tot de aici este si sters cand timpul lui de procesare este 0. Mai concret spus pleaca de la casa de marcat cand timpul de asteptare(pt clientii de la coada timp de asteptare) s-a terminat.

```
@Override
public void run() {
    while(true) {
        while (!customers.isEmpty()) {
            Customer c = customers.peek();
            try {
                Thread.sleep( millis: 1000L);
                c.timesUp();
                waitTime.decrementAndGet();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            if(c.getProcTime() == 0)
                customers.remove(c);
        }
    }
}
```

CONCLUZII

In finalul acestei documentatii as vrea sa precizez cateva concluzii cu privire la intocmirea proiectarii acestui simulator de cozi. Consider importanta aceasta tema deoarce ne invata intr-un mod interactiv cum sa gestionam firele de executie , lucrul in paralel.

In ceea ce priveste performanta mea legat de aceasta tema, ma bucur ca am reusit sa o finalizez la timp cu toate cerintele. Insa , mi-a luat destul de mult timp sa inteleg principiul de functionare al threadurilor si cum ar trebui aceste sincronizate fara keyword. Totodata mi-a luat mult timp sa inteleg cum sa fac ca thread-ul meu sa astepte sa introduc datele de intrare iar mai apoi sa inceapa functionarea.

Ma bucur insa ca intr-un final am inteles logica acestor thread-uri si ma voi putea folosi in proiectele viitoare. Prezentarea suport mi-a fost de mare ajutor.

Referitor la alte implementari , da se poate imbunatati ca orice alt proiect. Ma gandesc la interfata care sa arate mult mai interactiv , un client sa reprezinte chiar un client in sine nu doar niste campuri introudse de utlizator, la fel si procesarea comenzii , nu doar o simpla decrementare, insa pentru un moment ma delcar multumit cu cele implementate.

BIBLIOGRAFIE

https://users.utcluj.ro/~igiosan/Resources/POO/Curs/POO11.pdf
5.04.2021 17:35

https://www.tutorialspoint.com/java/java_multithreading.htm

5.04.2021 21:30

https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html

7.04.2021 02:30

https://stackoverflow.com/questions/4818699/practical-uses-for-atomicinteger

7.04.2021 15:30

https://www.javatpoint.com/java-filewriter-class

7.04.2021 22:15