Documentatie

Queue simulator



Tehnici de programare

Tema2

Realizat de Balc Horia-Ovidiu

Grupa 30227

Cuprins

1 . Cerinte Functionale

2 . Obiectivul temei

3 . Analiza problemei

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

5 . Implementare

6 . Concluzie

7 . Bibliografie

# Cerinte Functionale

Cozile sunt utilizate în mod obișnuit pentru modelarea domeniilor din lumea reală. Principalul obiectiv al unei cozi este să furnizeze un loc pentru ca un client să aștepte înainte de a primi un serviciu.Intr-un management bazat pe coadă sistemul este interesat să minimizeze perioada de timp pe care clienții o asteapta la cozi înainte

ca ei sa fie serviti. O modalitate de a minimiza timpul de așteptare este adăugarea mai multor servere, adică mai multe cozi in simulator (fiecare coadă este considerată ca având un procesor asociat), dar la această abordare cresc costurile furnizorului de servicii. Când se adaugă un server nou, clienții care sunt în așteptare vor fi uniform distribuiti la toate cozile disponibile curente.

Proiectati si implementati un sistem de simulare a unui magazin cu ajutorul thread-urilor.

Programul:

- citirea din fisier a numarului de clienti, a numarului de server, timpul minim pentru a ajunge la coada, timpul maxim in care o casa este deschisa,timpul intre care casele primesc clienti noi si timpul intre care se scaneaza produsele

-scrierea in fisier a clientilor in asteptare,timpul current si oamenii care s-au pus in coada

-generarea random a n Task-uri

- utilizarea unui .jar file

- inchiderea si deschiderea cozilor dynamic

-calcularea timpului mediu

-testarea in cele 3 fisiere.txt

# Obiectivul Temei

2.1 Obiectivul principal

Obiectivul temei este crearea,in limbajul JAVA, a unui simulator de magazin cu ajutorul caruia se pun clientii la coada intr-un mod eficient din punct de vedere al timpului de asteptare a clientilor.Aceasta tema are practicabilitate in viata reala si are multe obiective importante care stau la bazaprogramarii in limbajul JAVA.

2.2 Obiectivul secundare

Obiectivele secundare sunt: - folosirea programarii obiectuale prin utilizarea incapsularii

- scrierea variabilelor in stilul Camel Case

-utilizarea metodelor(maxim 30 de linii) si claselor( maxim 300 de linii)

- citirea din fisier a numarului de clienti, a numarului de server, timpul minim pentru a ajunge la coada, timpul maxim in care o casa este deschisa,timpul intre care casele primesc clienti noi si timpul intre care se scaneaza produsele

-scrierea in fisier a clientilor in asteptare,timpul current si oamenii care s-au pus in coada

-generarea de numere random pentru datele numarului de client citite

- invatarea de a lucra cu multi-thread-uri implementate cu ajutorul interfetei Runnable si asigurarea sincronizarii thread-urilor

- utilizarea unui .jar file

- inchiderea si deschiderea cozilor dynamic

-calcularea timpului mediu

-testarea in cele 3 fisiere.txt

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

3.1Analiza problemei

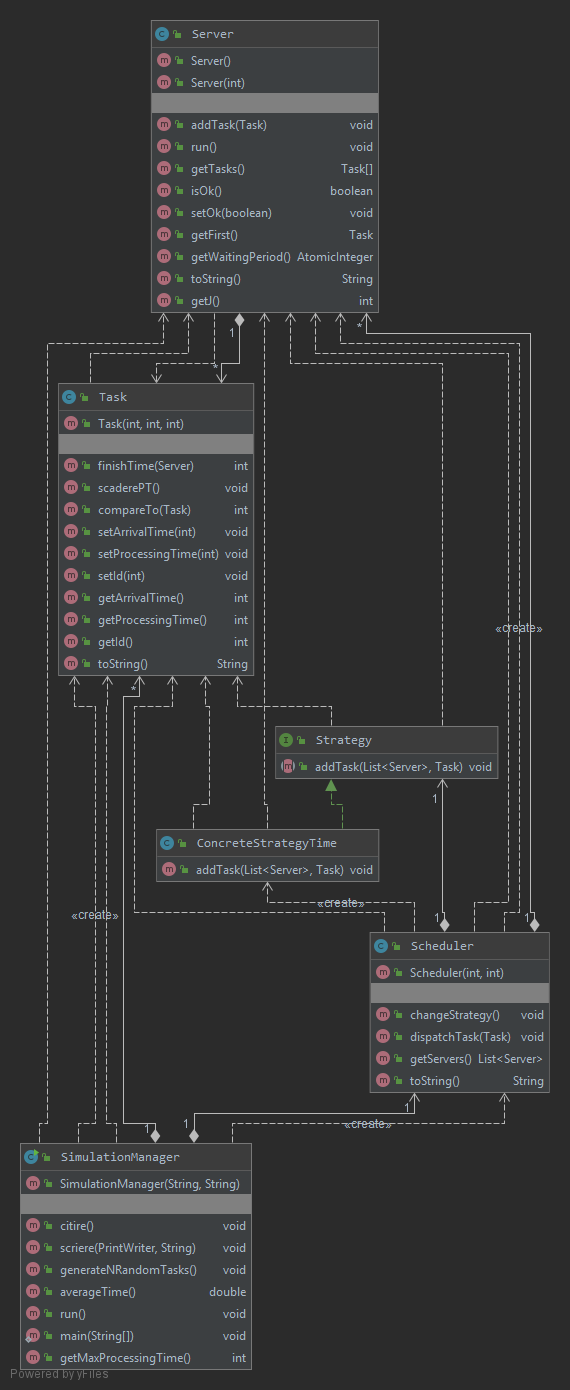
Pentru aceasta problema o utilizare mai eficienta este utilizarea multi-Threadurilor ,un thread pentru o singura coada . Avand o clasa Simulator care este un thread care este declarat si pornit in metoda statica main.Aceasta apeland clasa Scheduler care apeleaza si porneste threadul din clasa Server,care reprezinta o coada de la casa. Serverul, si SimulationManager utilizeaza cozi respective liste de task-uri, care reprezinta un client.

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

4.1Decizii de proiectare si Structuri de date

Am ales sa creez o claza principala (SimulationManager) in care simulam un model cu ajutorul unei liste de client(un client este reprezentat de clasa Task) care semnifica clientii care sunt in asteptare(nu s-au pus la coada) si a clasei Scheduler, care contine o lista de servere(un server reprezinta o casa de marcat dintr-un magazin). De asemenea , un server contine o coada de Task-uri care reprezinta oamenii care stau la coada.

4.2Diagrame UML



5.Implementare

Clasa Task

Aceasta clasa reprezinta un client care are un id, un timp de sosire la coada (arrivalTime) si un timp de procesare a produselor (processingTime). De asemenea, aici se implementeaza interfata default Comparable<Task> pentru a putea suprascrie metoda compareTo. Totodata, am utilizat getteri (getArrivalTime si getProcessingTime) pentru variabilele arrivalTime si ProessingTime. In aceasta clasa am folosit metodele:

-scaderePT pentru a scadea timpul de procesare cu o secunda

-toString, metoda default suprascrisa, cu ajutorul careia se returneaza un string intre paranteze ce reprezinta id-ul clientului (id), timpul de sosire la coada, daca nu este nimeni in fata la casa (arrivalTime), timpul de procesare a produselor la casa (processingTime)

Clasa Server

Aceasta clasa reprezinta o casa de la magazine care contine o coada de Task-uri, un waitingPeriod (o variabila AtomicInteger ce ne ajuta atat la functionarea mai exacta a threadului cat si in calcularea timpului mediu de asteptare), o variabila booleana ok care ne ajuta la oprirea thread-ului si o variabila de tip int, j, care ne ajuta la calcularea timpului mediu de asteptare. In aceasta clasa am folosit metodele:

-addTask pentru a pune un Task nou in coada, utilizand metoda predefinita add. De asemenea, se adauga la waitingPeriod valoarea timpului de procesare (processingTime),cu ajutorul metodei default addAndGet.

-folosim getteri (getWaitingPeriod pentru waitingPeriod si getJ pentru J) si setter (setOk pentru a seta ok cu booleanul dat de la parametru)

-toString, metoda default suprascrisa, in care se parcurge coada de Task-uri si se adauga in stringul ce trebuie returnat, tostringul de la fiecare Task (adica folosecte toStringul din clasa Task) Daca este nu este goala lista atunci se afiseaza mesajul”closed\n”.

-getFirst pentru a extrage primul element din coada, daca aceasta nu este goala (verificarea se face cu ajutorul metodei predefinite isEmpty) se va allege primul element din lista prin intermediul functiei default peek.

-run se foloseste datorita faptului ca aceasta clasa implementeaza interfata predefinita Runnable.Cat timp booleanul ok este true daca lista de Task-uri nu este goala (folosind metoda default isEmpty) luam primul element din coada (cu metoda peek), dupa care parcurgem cu un for unde punem threadul pe sleep timp de 1 secunda si decrementam waitingPeriod (cu metoda predefinita decrementAndGet). Dupa iesirea din for stergem Taskul din coada (cu ajutorul metodei remove). Tot ce este in while este prins de un block try catch pentru a nu exista intreruperi (InterruptedException).

Interfata Strategy

Aceasta este o interfata ce defineste metoda addTask.

Clasa ConcreteStrategyTime

Aceasta este o clasa ce implementeaza interfata Strategy si suprascrie metoda addTask.

Metoda acestei clase primeste ca parametrii o lista de server si un Task. Folosesc un obiect de tip server (m) pe care il initializez ca fiind primul server din lista. Cu ajutorul unui foreach parcurgem toate serverele din lista si cautam cea mai mica waitingPeriod existent in lista si m va lua valuarea serverului gasit, dupa care adaugam Task-ul din parametru in serverul m.

Clasa Scheduler

Aceasta clasa reprezinta organizatorul de cozi, contine o lista de servere (servers), doua variabile int (maxNoServers si maxTasksPerServer) si o variabila strategy de tip Strategy.In constructorul acestei clase initializam variabilele clasei si maxNoServers threaduri. Cu ajutorul unui for se va initialize cate un server si se va adauga in lista de servere si initializam si pornim thread-ul serverului respectiv. In acesta clasa am implementat metodele:

-dispatchTask apeleaza metoda addTasks din Strategy

-changeStrategy initializeaza variabila Strategy

-folosim getter-ul (getServers) pentru servers

--toString, metoda default suprascrisa, in care se parcurge lista de Servere si se adauga in stringul ce trebuie returnat, tostringul de la fiecare Server (adica folosecte toStringul din clasa Server)

Clasa SimulationManager

Aceasta clasa reprezinta simulatorul de magazin, si contine limita timpului (timeLimit) ,timpul maxim de procesare (maxProcessingTime), timpul minim de procesare (minProcessingTime), timpul maxim de sosire (maxArrivalTime), timpul minim de sosire (minArrivalTime), numarul de client (numberOfClients), numarul de servere (numberOfServers), o variabila de tip Scheduler si o lista de Task-uri ce reprezinta clientii care nu s-au pus la coada.

In constructor se genereaza n random Tasks, se initializeaza un obiect de tip Scheduler.

Metodele folosite in aceasta clasa sunt:

-citire initializam un obiect de tip file f cu numele fisierului dorit,dupa care intr-un bloc try catch initializam un nou scanner myReader in care putem prinde o exceptie de tipul

FileNotFoundException. Cat timp myReader mai are linii (hasnextLine) daca contorul nostru (pe care l-am initializat cu 0 inainte de structura repetitive while) este:

\* 1 se retine informatia in numberOfClients (cu ajutorul metodei predefinite Integer.parseInt),

\*2 se retine informatia in numberOfServers (cu ajutorul metodei predefinite Integer.parseInt),

\* 3 se retine informatia in timeLimit (cu ajutorul metodei predefinite Integer.parseInt),

\*4 se retine informatia in minArrivalTime si maxArrivalTime despartite de o virgule (cu ajutorul metodei predefinite Integer.parseInt si metodei split),

\* 5 se retine informatia in minProcessingTime si maxProcessingTime despartite de o virgule (cu ajutorul metodei predefinite Integer.parseInt si metodei split).

\*mai mare de 5 folosim break ca sa crestem eficienta din punct de vedere a timpului

Dupa iesirea din while inchidem myReader.

-scriere care primeste ca parametru un obiect de tip PrintWriter si un string s cu ajutorul carora se scrie in fisier stringul s

-generatedNRandomTasks foloseste functia de citire, dupa care se initializeaza lista generatedTasks ca un ArrayList, folosim inca un Array List ca o copie si intr-un for generam, cu ajutorul unui obiect de tip Random si metodei predefinite nextInt, numerele random intre minArrivalTime si maxArrivalTime, si respective dintre minProcessingTime si maxProcessingTime.Dupa ce creem Task-ul cu valorile calculate (id este iteratorul din for I incrementat cu 1 ) acesta se adauga in ArrayList-ul copie. In afara structurii repetitive sefolosecte metoda default Collections.sort care sorteaza crescator din punct de vedere a timpului de sosire (datorita faptului ca am suprascris metoda compareTo).La finalui metodei generatedTasks ia valoarea ArrayList-ului copie.

-averageTime care returneaza un double i care este initializat cu 0 si cu ajutorul unui for, i va fi suma tuturor timpilor de procesare din lista genersatedTasks.

- run se foloseste datorita faptului ca aceasta clasa implementeaza interfata predefinita Runnable. Se initializeaza un double i cu valoarea returnata de metoda averageTime. Intr-un bloc try catch sunt puse majoritatea lucrurilor importante. Initializam un obiect de tip PrintWriter pe care il vom folosi pentru scrierea in fisier, care este stringul dat in parametrul initializarii. Cat timp currentTime, care este o variabila de tip int initializata cu 0 la inceputul metodei, este mai mic decat timeLimit folosim un iterator cu ajutorul caruia parcurgem generatedTasks si daca arrivalTime-ul este egat cu currentTime il adaugam in coada cu metoda dispatch apelata de obiectil scheduler, dupa care stergem elementul din lista generatedTasks si iesim din aceasta structura repetitive (parcurgerea). Scriem cu ajutorul unui for toti clientii care se afla in asteptare (nebagati in coada), iar dupa iesirea din for se va afisa in fisier timpul current, dupa care scheduler.toString, adica cozile de la case. Cu ajutorul unui for, daca generatedTasks nu este goala, vom parcurge toate serverele si le vom verifica daca waitingTime-ul este 0 (cu ajutorul getter-ului), in caz afirmativ setam booleanul ok din clasa Server cu 0, iar daca nu lui i i se va adauga getJ-getWaitingPeriod.intValue pentru a se calcula averageTime-ul si variabila k va lua valoarea 1, nu 0 cum era la initializare. Daca dupa iesirea din for k este 0 atunci se face break si se iese si instructiunea if (cea mare cu lista). Dupa iesirea din if intr-un bloc try catch, pentru anu arunca o exceptie de tipul IntrerruptedException, threadului ii aplicam sleep timp de 1 secunda. Dupa acest bloc scriem in fisier AverageTime-ul care este reprezentat de impartirea lui i la numarul de clienti, dupa care se inchide fisierul cu metoda default close. La finalul metodei prindem posibilele exceptii care se pot arunca, adica FileNotFoundException si respectiv UnsupportedEncodingException.

-main este metoda static in care se instantiaza un obiect de tip SimulationManager, si unul de tip Thread, iar pe acesta din urma il porneste cu ajutorul metodei start.

Teste de implementare

In fisierul out se vor afisa clientii in asteptare, dupa care timpul curent dupa care cozile, daca sunt goale se va afisa closed.

De asemenea am facut fisierul .jar

6. Concluzii

6.1Ce s-a invatat

Mi-am imbunatatit cunostiintele in limbajul JAVA, mai ales in ceea ce consta folosirea thread-urilor si a citirii si scrierii in fisier.De asemenea am folosit functii random ceea ce nu utilizasem des in JAVA

6.1Dezvoltari ulterioare

Imbunatatirea unor metode din clase, din punct de vedere al eficientei in functie de memorie sit imp de executare.

7.Bibliografie

-Am cautat pe <https://stackoverflow.com/> si pe <https://www.geeksforgeeks.org/> pentru a cauta citirea si scrierea din fisier precum si unele informatii despre threaduri, unde am avut cate o nelamurire sau o problema.

-De asemenea scheletul proiectului meu este inspirit din pdf-ul di Assigment2, linku il gasiti mai jos:

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