

Tehnici de Programare

Tema 2 – Queue Simulator

Profesor curs: Conf. Tudor Cioara Sudent: Varga Alexandra-Ioana

Profesor laborator: Dorin Moldovan Grupa: 30223

Cuprins:

1.Cerinte functionale

2.Obiectivul temei

3.Analiza problemei

4.Proiectare

5.Implementare

6.Concluzii

7.Bibliografie

1. **Cerinte functionale**

Proiectati si implementati o aplicatie de simulare, cu scopul de a analiza sistemele bazate pe cozi pentru determinarea si minimizarea timpului de asteptare al clientilor.

Aceasta aplicatie genereaza aleator niste clienti la inceputul simularii, iar in functie de momentul in care clientul ajunge pentru a primi un “serviciu”, este adaugat la coada la care timpul de asteptare este minim. De asemenea,aplicatia urmareste timpul total petrecut de fiecare client la cozi si calculeaza timpul mediu de asteptare.

1. **Obiectivul temei**

2.1.Obiectivul principal

Obiectivul principal al acestei teme este crearea unei aplicatii care determina si minimizeaza timpului de asteptare al clientilor.

2.2.Obiective secundare

* Dezvoltarea de use case-uri si scenario

“Use case”-urile reprezinta un set de scenarii legate de cum sistemul este folosit si interactiunea acestuia cu un rol(in cazul nostrum, un utilizator al aplicatiei).

* Structuri de date

Alegerea structurii de date pentru a duce la capat obiectivul principal

* Impartire pe clase

Se va prezenta modul in care am impartit clasele si motivul alegerii

* Diagrame UML
* Dezvoltarea algoritmilor

1. **Analiza problemei**

3.1.Use case-uri

Simulatorul de cozi, consta in simularea a unei serii de N clienti care ajung pentru un anumit “serviciu” la Q cozi, sunt serviti si in final parasesc cozile. Toti clientii sunt generati aleator cand simularea a inceput. Clientii sunt caracterizati de 3 parametrii: id-ul (un numar de la 1 la N), t\_arrival (timpul de simulare in care clientii sunt gata de a se aseza in fata cozilor) si t\_service (durata in care clientul trebuie sa astepte pentru a fi servit de catre casier).

Scopul acestei aplicatii este de a distribui clientii in mod optim, in functie de timpul de asteptare al fiecarei cozi.

3.2.Scenarii

Principalele scenarii de success:

1. Aplicatia genereaza corect N client.
2. Aplicatia creeaza corect cele Q cozi.
3. Simularea a inceput.
4. Clientii sunt asezati la cozi in functie de timpul minim de asteptare.
5. Simularea s-a incheiat cu success.

Secvente alternative:

1. Clientii nu sunt distribuiti corespunzator la cozi.

**4.Proiectare**

4.1.Structuri de date

Prima data, am construit diagrama de clase si relatiile dintre clase. Acesta este un prim pas, conform caruia ne vom ghida in continuare. Diagrama de clase contine 4 clase principale: Server, Task, Scheduler si SimulationManager (care tine locul clasei MainClass). De asemenea, diagrama contine interfata Strategy, care trebuie implementate conform specificatiilor temei. Din aceste clase au fost extrase use-case-urile determinate anterior.

4.3.Diagrama de clase

O imagine care conține text

Descriere generată automat

4.4.Dezvoltarea algoritmilor

Pentru a citi datele de intrare dintr-un fisier, am folosit un obiect file, prin intermediul caruia, cu un scanner, am parcurs fisierul, astfel reusind sa extrag datele. Dupa ce am extras datele, am generat clientii, iar apoi simularea a inceput.

Pentru aceasta tema am folosit thread-uri. O data ce simularea a inceput, acestea si-au inceput ciclul de viata. Clasa Server implementeaza interfata Runnable, suprascriind metoda “run()”, deoarece cozile sunt niste obiecte care au viata atata timp cat au clienti.

In timpul simularii, pentru fiecare timp de simulare, s-a afisat timpul, listele de asteptare a clientilor, si cozile care fie sunt inchise, “CLOSED”, fie au client care asteapta pentru a fi serviti.

Clientii sunt adaugati la cozi, doar atunci cand t\_arrive, adica timpul in care sunt gata de a fi asezati la coada, este egal cu timpul de simulare. Daca exista clienti care indeplinesc aceasta conditie, sunt asezati la cozi, este setat noul timp de asteptare al cozii in care este adaugat, si este eliminat din lista de clienti care asteapta pentru a fi pusi la o coada. Dupa fiecare astfel de verificare, se afiseaza in fisierul de iesire, datele de iesire actualizate.

4.5.Interfate utilizate

La acest proiect am folosit interfata Comparable pentru a ordona lista de client dupa timpul in care a ajuns fiecare client, usurandu-mi astfel munca.

De asemenea, interfata pe care se bazeaza acesta tema, este Runnable, cu ajutorul careia thread-urile prind viata. Dupa ce thread-ul prinde viata, thread-ul este “runnable”. Un thread care este in aceasta stare, se considera ca isi executa propriile servicii.

**5. Implementare**

a. Clasa Server – tine locul casierului (intr-o asociatie cu lumea reala ) si are atributele waitingPeriod, care reprezinta suma timpilor de asteptare a tuturor clientilor (Task-urilor) care se afla in coada si in dependenta de care se va face selectia serverului in cazul utilizarii strategiei ConcreteStrategyTime, ID-ul serverului este folosit cu scop de afisare a numarului serverului (cozii). De asemenea, serverul mai contine si o lista de task-uri ce urmeaza a fi indelpinite, stocate intr-o structura de date numita BlockingQueue <Tasks>. Am preferat aceasta structura de data simplului ArrayList pentru a evita problemele cu firele de lucru. Toate atributele au gradul de vizibilitate private. Am folosit fenomenul de incapsulare pentru a limita accesul obiectelor de alta clasa la atributele clasei Server, Astefel, am implementat settere si gettere pentru fiecare atribut. De asemenea, Serverul poate fi initializat din constructorul Server(int c, int g). Metoda run este cea care asigura posibilitatea simularii in timp real a cozii. In cadrul acestei metode, se ia primul element din coada , si pune threadul serverului pe wait un interval de timp egal cu timpul de procesare al Task-ului. Clasa Server implementeaza interfata Runnable si implementeaza metoda run, pentru a face posibila implementarea Threaduri-lor.

b. Clasa Task – are drept atribute ID, arrivalTime si processingTime. Aceasta clasa poate fi asociata cu clientii, daca e sa facem o analogie cu lume reala. Id-ul clientului este folosit pentru a identifica fiecare client, arrivalTime-ul reprezinta timpul cand clientul ajunge a fi primul element din coada, adica poate fi servit de Server si procTime-ul indica timpul necesar Task-ului pentru a fi indeplinit. De fiecare data cand Task-ul ajunge sa fie servit, thread-ul corespunzator serverului isi opreste executia pentru “processingTime” secunde, timp in care acest atribut al Task-ului devine 0 si Task-ul se considera indeplinit.Au fost implementate metodele principale: gettere, settere si generateOneTask, care genereaza Task-uri random in functie de datele primite din fisier. Timpul arrive si timpul de procesare sunt cuprinse in intervalul stabilit si se genereaza random.

c. Clasa SimulationManager – este responsabila pentru gestionarea tuturor proceselor din program()supermarket. Am incercat sa implementez aceasta clasa cat mai user-firendly, adica sa afisez in consola cat mai clar coada, clientii si sa implementez o simulare real-time. De asemenea, am dat nume cat mai sugestive variabilelor , pentru ca user-ul sa stie instinctiv ce reprezinta fiecare.

Aceasta clasa declara si initializeaza toate elementele principale a programului: Sheduler-ul, SelecyionPolicy-ul si task-urile generate. Aceasta casa implementeaza interfta Runnable si suprascrie metoa run(), in care am initializat thread-ul de baza, care tine evidenta timpului. Aici au loc si afisarile.

Aceasta clasa este responsabila pentru impartirea Task-urilor spre servere si este un simulator a “organizatorului” real al unui procesor care randuieste task-urile catre executie.. Tot in interiorul clasei sunt implementate metodele changeStrategy, care selecteaza metoda de selectare a serverului (in dependenta de lungimea cozii sau de timpul de asteptare), dispatchTask - care imparte task-urile catre servere, si getter pentru lista de servere. Toate atributele sunt, la fel, Private, pentru a respecta principiul de incapsulare.

e. Interfata Strategy

Aceasta interfata are rolul de a concepe o strategie conform careia se vor distrbui clientii in cozi. Aceasta interfata va fi implementate de clasele ConcreteStrategyQueue si ConcreteStrategyTime, car vor trebuie sa implementeze metoda addTask() declarata in interfata;

public interface Strategy {

void addTask(List<Server> servers , Task t);

}

f. Clasele ConcreteStrategyTime si ConcreteStrategyQueue

Aceste clase implementeaza interfata Strategy si implementeaza metoda addTask. In functie de clasa, aceasta metoda este implementate diferit. In priml caz, Un client va fi adaugat la serverul cu cel mai scurt timp de asteptare. In al doilea caz, Task-ul va fi preluat de server-ul cu cele mai putine task-uri.

**6. Concluzii**

Toate obiectivele proiectului au fost atinse. Aceasta tema mi-a consolidat cunostintele de programare orientata pe obiect, intrucat implementarea temei a necesitat structurarea claselor si crearea obiectelor : Scheduler, SimulationManager. De asemenea, am invatat sa implementez strategiile, ceea ce este o buna tehnica de programare si o voi implementa pe viitor. Am invatat sa folosesc firele de executie si sa simulez cu ajutorul lor o aplicatie in timp real. Ca posibilitati de dezvoltare ulterioara, ar putea fi implementata o inerfata care sa semene cat de mult cu coada unui supermarket. Ca functionalitate, s-ar putea adauga alte strategii de impartire a clientilor .

**7.Bibliografie**

1. „Java\_Concurency.pdf”
2. <https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html>
3. <https://www.tutorialspoint.com/java/util/timer_schedule_period.htm>
4. <https://www.javacodegeeks.com/2013/01/java-thread-pool-example-using-executors-and-threadpoolexecutor.html>

a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a

a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a

a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a

a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a

a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a

a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a

a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a

a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a

a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a