DOCUMENTATIE TEMA 2

SIMULARE COZI

Nume: Balint
Prenume: Simina Monica
Grupa: 30226
Universitatea Tehnica din Cluj-Napoca,
Facultatea de Automatica si Calculatoare.

Cuprins:

1. Obiectivul temei:

Obiectivul principal al temei este acela de a proiecta si a implementa o aplicatie care are ca si scop realizarea unei simulari care sa analizeze pozitionarea anumitor clienti in mai multe cozi in functie de timpul de asteptare al acestor.

Cozile sunt des inatlnite in viata reala, iar scopul principal al unei cozi este de a oferi un loc mai multor clienti pentru a astepta dupa un anumit serviciu.

Aplicatia pe care dorim sa o implementam ar trebui sa simuleze procesul repartizarii a N client in mai multe cozi, sa calculeze timpul optim de asteptare a acestora, servirea clientilor si eventual momentul in care acestia parasesc coada.

2. Analiza problemei, asumptii, modelare, scenarii, cazuri de utilizare, erori:

Pentru a vorbi de implementarea cerintei de mai sus, este necesar in pimul rand analiza acesteia. Se poate spune ca aceasta problema este de complexitate medie, deoarece se propune realizarea unor operatii relativ simple asupra unor cozi . Aceasta problema presupune in primul rand intelegerea foarte bine a conceptului de Thread, dar si a sincronizarii acestuia. Conceptul de thread (fir de execuție) defineste cea mai mica unitate de procesare ce poate fi programata spre executie de catre sistemul de operare. Este folosit în programare pentru a eficientiza executia programelor, executand portiuni distincte de cod în paralel în interiorul aceluiași proces. Câteodata însă, aceste portiuni de cod care constituie corpul threadurilor , nu sunt complet independente și în anumite momente ale executiei, se poate intampla ca un thread să trebuiasca sa astepte executia unor instructiuni din alt thread, pentru a putea continua executia propriilor instructiuni . Aceasta tehnica prin care un thread asteapta executia altor threaduri inainte de a continua propria executie, se numeste sincronizarea threadurilor.

Threadurile sunt diferite fata de clasicele procese gestionte de sistemele de operare ce suporta multitasking, in principal prin faptul ca, spre deosebire de procese, toate threadurile asociate unui proces folosesc acelasi spatiu de adresare . Procesele sunt in general independente , in timp ce mai multe threaduri pot fi asociate unui unic proces. Procesele stocheaza un numar semnificativ de informatii de stare, in timp ce threadurile dintr-un proces impart aceeasi stare, memorie sau alte resurse. Procesele pot interactiona numai prin mecanisme de comunicare interproces speciale oferite de sistemul de operare (semnale, semafoare, cozi de mesaje și altele asemenea) . Cum impart acelasi spatiu de adresare, threadurile pot comunica prin modificarea unor variabile asociate procesului si se pot sincroniza prin mecanismele proprii. In general este mult mai simplu și rapid schimbul de informatii intre threaduri decat intre procese.

Atat firele de executie, cat și procesele au stari ce pot fi sincronizate pentru a evita problemele ce pot aparea datorita faptului ca impart diverse resurse . In general , fiecare fir de executie are o sarcina specifica și este programat astfel incât sa optimizeze utilizarea procesorului.

Cerinte functionale:

- generarea random a clientiilor;
- introducearea in coada a clientului cu timpul de sosire egal cu timpul current din coada, avand timpul de asteptare cel mai mic + cresterea timpului de asteptare pentru clientii din coada respective;
- dupa terminarea timpului de astepatare / servire a primului client din coada, trebuie realizata scoaterea acestuia din coada respectiva si scaderea timpului de asteptare a cozii din care facea parte.

Scenarii, asumptii si cazuri de utilizare:

Utilizatorul are la dispozitie 3 fisiere text in care se afla 3 teste pentru verificarea simularii sistemului, dar acesta poate introduce manual si alte date. Afisarea se realizeaza tot intr un fisier text, fiecare simulare avand un fisier separat.

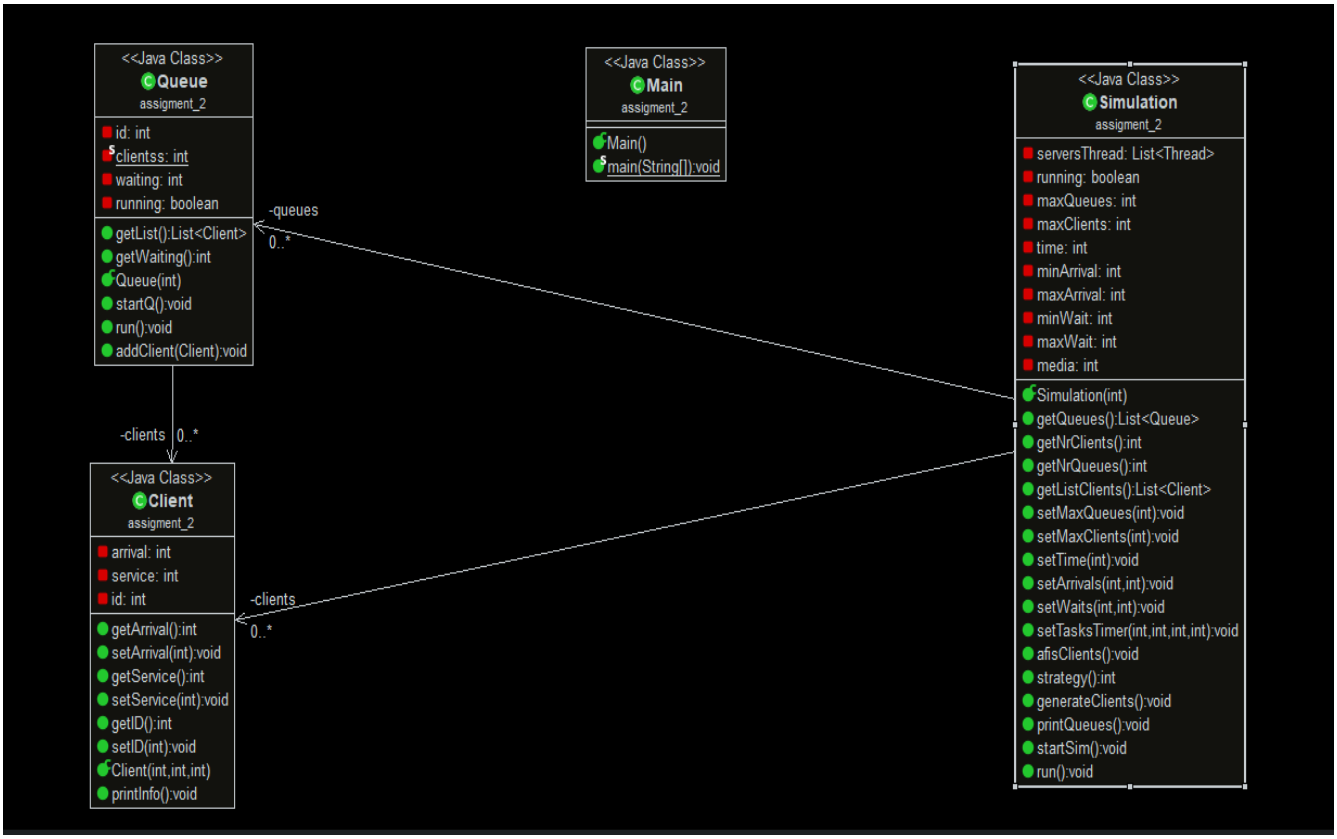
Acest proiect este unul foarte practic, cu un nume sugestiv care indica cu desavarsire scopul acestuia. Desi implementarea aleasa este una foarte simplista, consider ca aplicatia proiectata poate fi de folos in multe domenii, la o scara mai mica.

3. Proiectare:

Decizii de proiectare:

- in dezvoltarea aplicatiei care are ca si scop realizarea unei simulari care sa analizeze pozitionarea anumitor clienti in mai multe cozi in functie de timpul de asteptare al acestora, am decis utlizarea unui singur pachet cu mai multe clase, deoarece in lipsa unei interfete grafice nu se poate realiza partitionarea Model – View – Controller, iar clasele utilizate in acest proiect se leaga foarte bine intre ele.
- deoarece o coada din viata reala e formata dintr-unul sau mai multi clienti, dar si existenta propiu-zisa a unei cozi, am decis utilizarea unei clase Client si a unei clase Queue in care vor fi stocati clientii intr-un ArrayList de client.
- clasa care realizeaza simularea este clasa Simulation, in care se identifica un vector de tip Queue, unde vor fi adaugati clientii.

Diagrama UML:



Structuri de date:

- in dezvoltarea aplicatiei am folosit structure de date de tip ArrayList pentru stocarea in coada a clientilor si a umarului total de cozi, deorece operatiile fundamentale de modificare a structurii sunt usor de utilizat;

Proiectare clase:

- aplicatie este alcatuita din 4 clase:

1. Clsasa Client => are in component sa un constructor in care sunt date in ordine, id-ul clientului, arrival time-ul si service-ul acestuia, de asemenea aceasta clasa contine si gettere si sttere.
2. Clasa Queue => aceasta mosteneste clasa Thread; aici apere o lista de client care reprezinta clientii aflati in coada; Contine atat metoda suprascrisa run(), din clasa Thread, unde se relizeaza servirea primului client cat si metoda startQ() care porneste thread-ul(+gettere si settere).

3. Clasa Simulation => mosteneste clasa Thread. Are ca atribut un vector de coada, care reprezinta cozile deschise pentru simulare si atributele pentru setarea simularii: minimul si maximul timpului de servire, minimul si maximul timpului de sosire, timpul in care se va realiza simularea si numarul clientilor. Contine metoda generateClients() in care se genereaza clientii random si sortarea acestora in functie de timpul de sosire. Suprascrie metoda run() in care se realizeaza adaugarea fiecarui client la momentul de timp egal cu timpul de sosire a fiecarui client in coada cu cel mai mic timp de asteptare, indexul cozii fiind returnat de metoda min(). Metoda startSim realizeaza pornirea simularii si metoda printQueues care printeaza cozile.
4. Clasa Main=> aici se realizeaza citirea din fisier si se apeleaza metoda de scriere in fisier; se face apel de startSim() s porneste thread-ul.

Algoritmi:

-pentru a adauga un client in coada: se gaseste coada cu timpul de asteptare cel mai mic; la adaugarea clientului se creste timpul de asteptare pentru respective coada;

-la servirea unui client din coada, la momentul de servire al clientului, se scade din timpul de asteptare al cozii timpul de servire al respectivului client;

-generarea random a clientilor se face folosind clasa Random, care genereaza un intreg random intre limitele standard pentru timpul de servire si pentru timpul de sosire intre valorile standard: timpul de servire intre 1 si 10, timpul de sosire intre 1 si 20, iar timpul de simulare este 60 secunde, adica 6000 pentru thread, iar numarul de client este 15. Sau se face generarea valorii random intre valorile introduce de utilizator;

-pentru fiecare Thread de Coada, pornirea thread-ului duce la luarea valorii timpului de servire al clientului din capul cozii (primul client din coada) si se apeleaza metoda sleep din clasa Thread, care “adoarme” thread-ul pentru timpul de servire al respectivului client *1000, cee ace inseamna ca thread-ul o sa astepte cat timp dureaza servirea clientului, apoi se realizeaza stergerea clientului din coada si servirea urmatorului client;

-pentru Thread-ul de Simulare: se realizeaza generarea random a clientilor si introducearea acestora pe rand, in functie de timpul de sosire si de timpul current, in coada cu timpul de asteptare cel mai scurt. Metoda startSim si stopSim realizeaza pornirea thread-urilor de cozi, respective oprirea acestora.

-timpul mediu de asteptare se realizeaza impartind totalul timpului de asteptare al tuturor cozilor adunat la fiecare timp curent al simularii si se imparte la numarul total al clientilor cu care se realizeaza simularea.

4. Implementare:

Clasa Client:

-aceasta clasa contine atributele specific unui client: timpul de sosire, timpul de servire si indexul clientului si metode de get pentru;

Clasa Coada:

-aceasta clasa are mai multe atribute, dintre acestea cele mai importante fiind ArrayList-ul de clienti in care se

salveaza clinetii din coada si metode de pornire a thread-ului, suprascrierea metodei run().

Clasa Simulare:

-aceasta clasa realizeaza simularea propriu-zisa a cozilor: aici se genereaza random clientii, avand valorile specifice clientului, apoi se pun intr-un ArrayList de client si sorteaza crescator in functie de timpul de sosire.

Clasa Main:

-are doar metoda main in care se relizeaza citirea din fisier si aici incepe rularea unui thread.

5. Rezultate:

In urma rularii programului implementat s-au generat cu success informatile cautate, am atasat aici un exemplu al unei rulari:

```
The clients participating in the simulation are:
(1, 24, 4) (2, 28, 2) (3, 5, 3) (4, 18, 4)

Time: 0
  Queue no. 1:  is closed!
  Queue no. 2:  is closed!

Time: 1
  Queue no. 1:  is closed!
  Queue no. 2:  is closed!

Time: 2
  Queue no. 1:  is closed!
  Queue no. 2:  is closed!

Time: 3
  Queue no. 1:  is closed!
  Queue no. 2:  is closed!

Time: 4
  Queue no. 1:  is closed!
  Queue no. 2:  is closed!

Time: 5
---->(3, 5, 3) added to the queue no. 2
  Queue no. 1:  is closed!
  Queue no. 2: (3, 5, 3)

Time: 6
  Queue no. 1:  is closed!
  Queue no. 2: (3, 5, 3)

Time: 7
  Queue no. 1:  is closed!
  Queue no. 2: (3, 5, 3)

Time: 8
  Queue no. 1:  is closed!
  Queue no. 2: (3, 5, 3)

---->(3, 5, 3) has left the shop!

Time: 9
  Queue no. 1:  is closed!
  Queue no. 2:  is closed!

Time: 10
  Queue no. 1:  is closed!
  Queue no. 2:  is closed!

Time: 11
  Queue no. 1:  is closed!
  Queue no. 2:  is closed!

Time: 12
  Queue no. 1:  is closed!
  Queue no. 2:  is closed!

Time: 13
  Queue no. 1:  is closed!
  Queue no. 2:  is closed!

Time: 14
  Queue no. 1:  is closed!
  Queue no. 2:  is closed!

Time: 53
  Queue no. 1:  is closed!
  Queue no. 2:  is closed!

Time: 54
  Queue no. 1:  is closed!
  Queue no. 2:  is closed!

Time: 55
  Queue no. 1:  is closed!
  Queue no. 2:  is closed!

Time: 56
  Queue no. 1:  is closed!
  Queue no. 2:  is closed!

Time: 57
  Queue no. 1:  is closed!
  Queue no. 2:  is closed!

Time: 58
  Queue no. 1:  is closed!
  Queue no. 2:  is closed!

Time: 59
  Queue no. 1:  is closed!
  Queue no. 2:  is closed!

Time: 60
  Queue no. 1:  is closed!
  Queue no. 2:  is closed!

Average waiting time: 3 seconds!
```

6. Concluzii:

--aplicatia este una simplista, folositoare in simularea asezarii clientilor la cozi in functie de timpul de asteptare;

Ce s-a invatat:

-aprofundarea cunostintelor legate de ArrayList, Iteratori si bulca For-Each;

-aprofundarea cunostintelor legate de crearea unei interfete utilizator;

-aprofundarea cunostintelor legate de folosirea Thread-urilor si interfetei Comparable;

-am invatat diferite metode utile (ex.: String.substring()).

Dezvoltari ulterioare:

-selectarea de catre utilizator in functie de cum doreste sa se efectueze aseazarea clientilor la cozi: ex.: in functie de lungimea cozilor;

-imbunatatirea interfetei utilizator.

7. Bibliografie:

https://stackoverflow.com/

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

https://www.geeksforgeeks.org/

https://www.wikipedia.org/