**Facultatea de Automatica si Calculatoare**

**Departamentul de Calculatoare si Tehnologia Informatiei**

**QUEUES SIMULATOR**

**Student: Soponari Beatrice-Valentina**

**Grupa: 30221**

**An: 2020-2011**

Cuprins:

1. Obiectivul temei

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

3. Proiectare

4. Implementare

5. Rezultate

6. Concluzii

7.Bibliografie

1.Obiectiv

Scopul acestei teme de laborator a fost implementarea unei aplicatii de simulare a cozilor de clienti, cunoscand timpul de sosire al acestora si durata de procesare a serviciului, realizand doua metode de a prelua clientii, plasarea unui client la coada cu timpul minim de asteptare, sau la coada cu cei mai putini clienti.

Pentru a putea lucra cu aplicatia, era nevoie si de o interfata grafica prin care utilizatorul sa poata introduce anumite date necesare pentru derularea simularii, cum ar fi timpul simularii, numarul de clienti, numarul de cozi la care acestia vor fi distribuiti si doua intervale de timp pentru a se putea genera momentul sosirii si timpul necesar prelucrarii nevoilor clientului. Deoarece datele erau relativ multe pentru a fi atat introduse, cat si variate, am creat metode de generare aleatoare a acestora pentru a lucra mai usor si a nu necesita foarte mult timp pentru introducerea acestora.

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

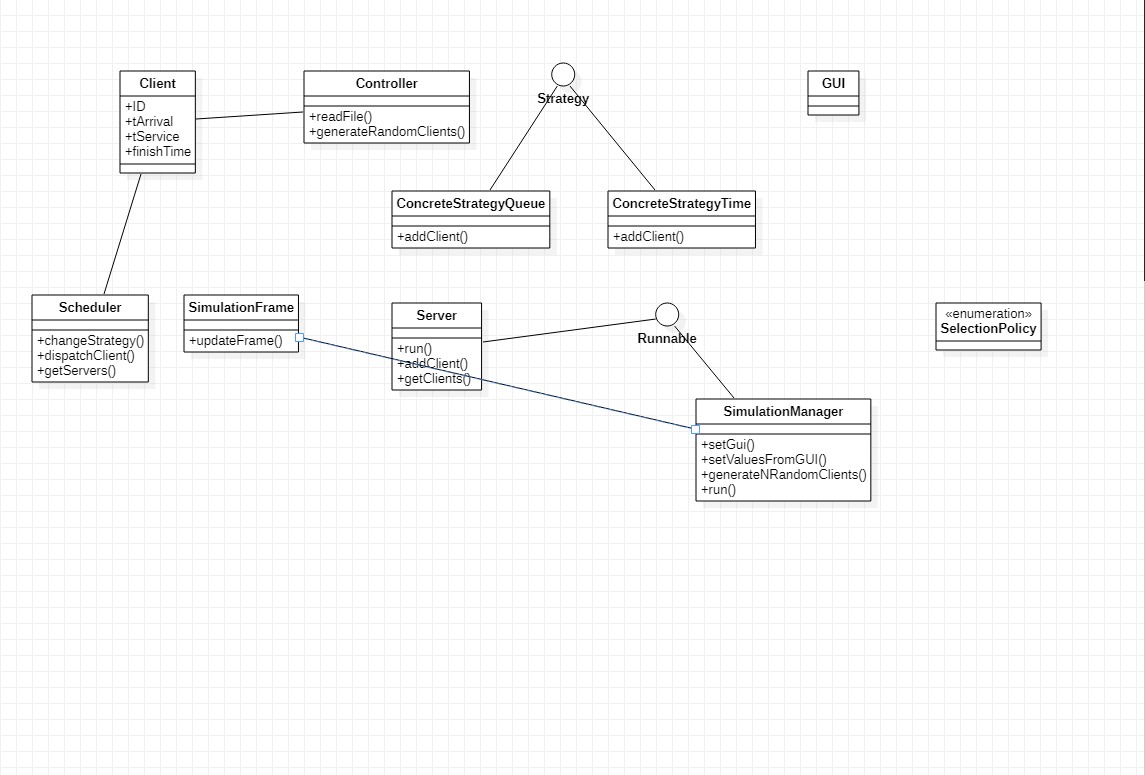
Problema propusa de aceasta tema este aceea de a rezolva problema cozilor sau ghiseelor aglomerate prin a oferi metoda cea mai eficienta de a dispune clientii la cozi, astfel incat timpul de asteptare pentru acestia sa fie minim, sau, o alta abordare ar fi plasarea clientilor la coada cu un numar minim de clienti, dar nu cu cel mai mic timp de astepare, strategie care, din punctul meu de vedere nu prezinta metoda optima, care sa rezolve intr-adevar problema ridicata de aceasta tema de laborator.

Folosim cozile pentru a rezolva probleme din lumea reala. Cu o politica de inserare FIF0, cozile au fost modul potrivit de a implementa aceasta cerinta, o coada avand corespondentul in lumea reala chiar cu o coada de clienti la magazin, o lista de persoane etc.Obiectivul principal pentru care folosim cozi este de a plasa un client la locul cel mai potrivit pentru a astepta inainte de a primi un serviciu. Problema propusa a fost alegerea cozii potrivite pentru fiecare client la care acesta se poate aseza pana la momentul sau de a primi un serviciu.

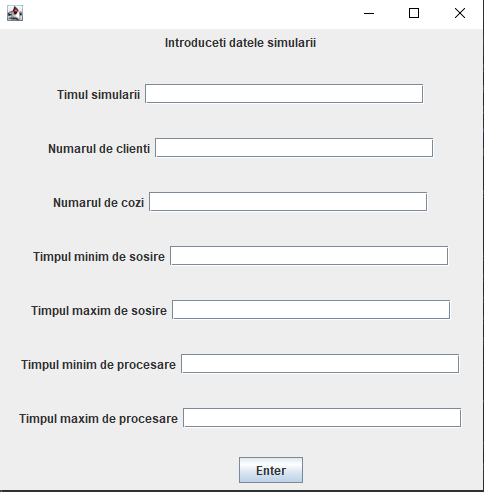
Scenariile de utilizare ale aceste aplicatii sunt modelate de numarul de clienti, timpul simularii, numarul de cozi si timpul de sosire si de procesare a nevoii clientului. Dupa procesarea acestor date se genereaza un scenariu aleator, care creeaza numarul de clienti specificat si le atribuie un ID si un timp de sosire si de procesare pentru a putea privi mai apoi simularea . Plasam fiecare client la coada care are timpul de asteptare cel mai mic, nu la coada cu cei mai putini clienti, deoarece am considerat ca aceasta abordare este mai apropiata de lumea reala. Ca si date de intrare si posibile scenarii am introdus datele oferite in lucrarea de laborator pentru testare si am atasat fisierele .txt generate, pe care le-am verificat si consider ca sunt corecte si ca datele rezultate corespund cu asteptarile din realitate. Pentru a prelua datele de intrare se poate folosi fie interfata grafica pentru a introduce datele de la tastatura, fie un simplu fisier, deoarece am implementat ambele metode, dar deocamdata datele se introduc prin interfata grafica pentru a demonstra functionalitatea acesteia.

3. Proiectare

Diagrama UML a fost creata inaintea inceperii proiectului propriu-zis, pe baza indrumatorului oferit in lucrarea de laborator si bazat pe cum am considerat necesare anumite metode, legaturi si atribute. Pentru Thread-uri am folosit metoda de implementare a interfetei Runnable, pentru a ma putea folosi mai apoi de obiectele din clasa respectiva pentru a porni thread-urile, care au fost sincronizate folosindu-ma de structuri de date ca si BlockingQueue si AtomicInteger. De asemenea am mai creat interfata Strategy pentru a putea crea doua politici de adaugare a unui client intr-o lista, prima si cea utilizata in simularea mea fiind ConcreteSTrategyTime care adauga un client in coada cu timpul de asteptare cel mai scurt, iar cea de a doua, pe care am implementat-o dar nu am folosit-o mai departe in rezolvarea temei, este politica dupa care se adauga un client in coada cu cei mai putini clienti, indiferent de timpul de asteptare estimat al fiecarei cozi.



Am rezolvat cerinta acestei aplicatii folosindu-ma de clasele Client, Controller, ConcreteSTrategyTime, ConcreteSTrategyQueue, Server, Scheduler, SimulationFrame, SimulationManager si GUI, de interfetele Strategy si Runnable si de pachetele java.util.\*, java.awt.\*, java.swing.\*, java.util.ArrayList, java.util.Random, oferind astfel o interfata user friendly si o aplicatie usor de folosit. Am avut nevoie de clasa GUI pentru a implementa interfata grafica care preia datele de la utilizator, iar clasa SimulationFrame prezinta tot o interfata grafica care afiseaza simularea in timp real, afisand pe rand fiecare client care apare intr-o coada si il sterge mai apoi cand iese. Clasa Client prezinta obiectul principal al aplicatiei noastre, adica clientul si clasa SimulationManager se ocupa de simularea intregului scenariu.



4. Implementare

Clasa Client detine atributele ID, tArrival, tService si finishTime care sunt suficiente pentru a defini fiecare client in aplicatia noastra. tArrival specifica timpul la care ajunge clientul la o potentiala coada, tService specifica cat dureaza oferirea unui serviciu clientului nostru si finishTime este timpul estimat la care clientul ar trebui sa plece de la coada. De asemenea clasa Client implementeaza Comparable<> pentru a putea compara mai apoi in aplicatie potentialii clienti in functie de tArrival si a ii putea astfel plasa in ordinea corespunzatoare la o lista de asteptare (coada). Pentru un client care ajunge la momentul t, est eluat in considerare momentul la care ajunge primul in coada, iar abia apoi se decrementeaza tService corespunzator cu fiecare secunda care trece din timpul curent.

Clasele ConcreteStrategyTime si ConcreteStrategyQueue implementeaza interfata Strategy, care contine metoda addClient. Diferenta intre aceste doua clase este ca desi amandoua suprascriu metoda addClient() politica dupa care se introduc clientii intr-o coada este diferita. Pentru prima fiecare client este pus la coada cu timpul de asteptare minim, respectiv pentru cea de a doua clasa se adauga clientul in coada care contine cei mai putini clienti. Am implementat ambele metode desi pentru simulare am decis sa folosesc doar metoda care aseaza clientii la coada cu timpul de asteptare mai scurt. Pentru implementarea simularii noastre am ales sa folosesc politica de adaugare implementata in metoda ConcreteSTrategyTime deoarece aceasta s-a specificat la laborat si am considerat-o ca fiind metoda optima, dar pentru o mai mare varietate a alegerii am implementat si metoda care adauga un client in functie de lungimea cozii.

Enumeratia SelectionPolicy contine cele doua politici de introducere a clientilor in cozi, si anume SHORTEST\_QUEUE, care presupune plasarea clientului la coada cu cele mai putine persoane, respectiv SHORTEST\_TIME, care presupune alegerea cozii cu cel mai scurt interval de timp de asteptare si este folosita pentru a specifica mai departe politica de implementare in clasa Scheduler, folosindu-se interfata Strategy.

Clasa Server implementeaza interfata Runnable si are ca si atribute o lista de clienti clientTask si timpul de astepare care trebuie sa treaca pentru a fi preluat un nou client waitingPeriod. Metoda addClient() adauga un client nou in lista clientTask si creste

waitingPeriod cu tService al respectivului client care intra in coada.

De asemenea campul waitingPeriod este decrementat de fiecare data cand un client paraseste coada. Metoda getClients() returneaza un array cu clientii din lista clientTasks. Metoda run() plaseaza clientii in coada potrivita si updateaza frame-ul prin care se realizeaza afisarea simularii. Clasa contine structuri de date de tip BlockingQueue si AtomicInteger pentru a ajuta la sincronizarea thread-urilor pornite mai apoi de obiecte din aceasta clasa, lucru care era necesar pentru a ne putea descurca cu un numar mai mare de cozi de clienti deodata.

Clasa SimulationFrame este clasa care realizeaza interfata grafica de unde utilizatorul poate vizualiza rezultatul simularii. Metoda updateFrame() este apelata din SimulationManager si aceasta este folosita pentru a afisa doar clientii care sunt la momentul respectiv de timp in coada si de a ii scoate pe cei care au finishTime-ul egal cu momentul de timp la care ne aflam. Se afiseaza clientii ca un triplet de forma (ID, arrivalTime, serviceTime), unde ID-ul este generat automat fiecarui client prin incrementarea cu 1, arrivalTime este timul la care clientul ajuge la simulare si serviceTime este durata necesara pentru realizarea cerintelor clientului.

Clasa GUI implementeaza interfata grafica care interactioneaza direct cu utilizatorul, in care acesta introduce datele necesare pentru inceperea simularii, cum ar fi timpul in care aceasta se deruleaza, numarul de clienti, numarul de cozi, timpul de sosire minim si maxim al unui client si timpul de procesare minim si maxim al acestuia. Aceasta se foloseste de obiecte din pachetu java.swing.\*, cum ar fi JFrame, JPanel, JLabel, JTextField si JButton a caror scop este preluarea datelor de la utilizator si intrebuintarea lor atributelor din clasele noastre, prind text field-uri. In momentul in care buntonul Enter este apasat incepe executia programului nostru prin care se deschide mai apoi un al doilea frame care prezinta simularea propriu-zisa.

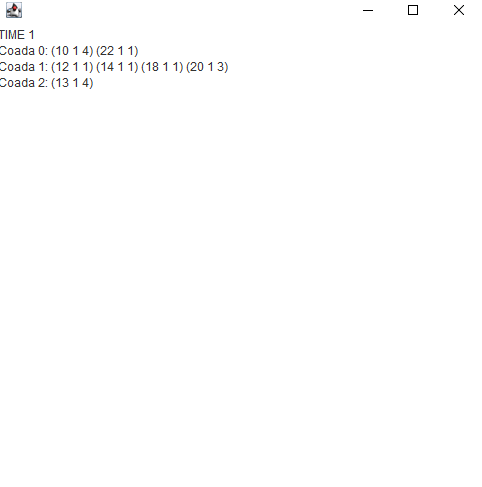
Clasa Scheduler contine ca atribute o lista de servere, o strategie de urmat (SHORTEST\_QUEUE sau SHORTEST\_TIME), un numar maxim de servere si un numar maxim de task-uri per server. Constructorul Scheduler() creeza numarul de servere necesare, metoda changeStrategy(SelectionPolicy policy) modifica strategia in functie de politica de selectie introdusa si metoda dispatchClient(Client c) adauga un client intr-un server specificat in functie de strategia abordata, folosindu-se de metoda addClient() implementata in clasa Server.

Clasa SimulationManager contine campurile carora li se atribuie datele preluate din interfata grafica, oferite de utilizator, politica dupa care functioneaza aplicatia, pe care am setat-o ca fiind SHORTEST\_TIME, un obiect Scheduler si o lista de clienti generati aleator in concordanta cu datele conferite de utilizator. Metoda setValuesFromGUI() atribuie valorile din campurile completate de utilizator atributelor clasei. Metoda generateNRandomClients() genereaza aleator cati clienti sunt specificati la inceput de utilizator, cu timpul de sosire incadrat in intervalul specificat si timpul de procesare la fel, apoi sorteaza lista de clienti dupa timpul de sosire al fiecaruia, pentru a ii putea plasa la o coada corespunzatoare momentului la care ne aflam. Metoda run() adauga pe rand fiecare client la coada potrivita, in functie de strategia aleasa, in momentul in care timpul este acelasi cu timpul de sosire al clientului si updateaza fereastra care ii prezinta utilizatorului parcursul simularii folosind un obiect din clasa SimulationFrame si metoda updateFrame() care primeste ca parametrii timpul curent, lista de clienti care trebuie afisata, obiectul scheduler prin care se iau clientii care sunt la rand, al caror timp de procesare nu este zero si care ajung la un moment de timp t egal sau mai mic decat momentul actual la care ne aflam si numarul de cozi disponibile, fiecare cu timpul de asteptare spefic calculat in functie de clientii care se afla in coada la momentul respectiv.

Clasa Controller contine si ea campurile cerute utilizatorului, doar ca aceasta preia datele dintr-un fisier prin intermediul metodei readFile(), iar metoda generateRandomClients() genereaza de asemenea o lista de clienti cu date aleatoare. Metoda readFile() nu este folosita la final in implementarea proiectului deoarece am preferat sa folosesc implementarea care preia valorile din interfata grafica destinata utilizatorului.

5. Rezultate

Din implementarea acestei aplicatii a rezultat o simulare a cozilor de asteptare cu clienti, care optimizeaza timpul de asteptare al clientilor, adaugandu-l pe fiecare client la coada in care va avea cel mai scurt timp de asteptare pana la preluarea sa si oferind o imagine de ansamblu in timp real a unei simulari care prezinta o modalitate eficienta de optimizare a timpului de asteptare al clientilor. Am introdus ca input datele specificate in suportul de prezentare al temei si am atasat fisierele in folderul cu proiectul pentru a putea urmari corectitudinea datelor oferite. Pentru fiecare client din coada, se afiseaza timpul la care a ajuns si timpul de procesare corespunzator in functie de momentul curent si de momentul in care ajunge primul in coada.



6.Concluzii

Aceasta tema a pus accentul pe folosirea thread-urilor, pe lucrul cu cozi, care sunt foarte folositoare pentru modelarea aspectelor din realitate, pe sincronizarea anumitor aspecte din problema si pe implementarea unui interfete Java. Am invatat lucruri noi despre sincronizarea thread-urilor folosindu-ne de structuri de date precum BlockingQueue si AtomicInteger, lucru cu care la inceput nu eram familiarizata si am inteles importanta lor si modalitatea corecta de folosire.

7.Bibliografie

* <http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html>
* <https://www.w3schools.com/java/java_threads.asp>
* <https://www.w3schools.com/java/java_files_read.asp>
* <https://www.w3schools.com/java/java_enums.asp>
* Notitele de la cursuri si de la laboratoarele de la disciplina Tehnici de programare