**DOCUMENTAȚIE**

**QUEUES SIMULATOR**

**TEMA 2**

NUME STUDENT: **ADELIN MOLDOVAN**

GRUPA:**302210**

**CUPRINS**

1. Obiectivul temei
2. Analiza problemei, modelare, scenarii, cazuri de utilizare
3. Proiectare
4. Imprelementare
5. Rezultate
6. Concluzii
7. Bibliografie

**1.OBIECTIVUL TEMEI**

Cerinta temei este de a proiecta si implementa un simulator de cozi. Aplicatia va simula o serie de N clineti care ajung sa fie pusi in Q cozi. Clientii intra intr-una dintre cele Q cozi si vor astepta sa fie serviti, iar la final acestia vor fi scosi din coada, pentru a putea intra urmatorul client. Fiecare client este reprezentat de un id, care este un numar de la 1 la N, timpul de sosire(arrivalTime), care reprezinta timpul din momentul simularii la care clientul va fi pus in coada si timpul de servire(serviceTime) care reprezinta intervalul de timp sau durata necesara in care se face servirea clientului. Altfel spus timpul de servire este timpul de asteptare al clinetului respective din momentul in care este pus in capul cozii. Clinetii sunt generati aleator si acestia vor fi adaugati in cozi in fucntie de timpul lor de sosire la timpul curent din executie. Aplicatia in sine va urmari parcursul clientilor in cozi, si aceasta este implementata intrucat sa aleaga cea mai buna metoda de a adauga clientii in cozi din punct de vedere al timpului de servire a clientilor care deja au fost introdusi in coada si totodata din punctul de vedere al lungimii conzilor. Rezulatele testelor sunt afisate intr-fisier “simulationOutput” si in interfata grafica. Tot in fisier se afiseaza si o medie a timpului de asteptare si a timpului de servire la momentul in care cozile sunt cele mai aglomerate.

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

Cozile sunt ulilizate peste tot si in lumea reala, fiind un lucru chiar foarte important in viata de zi cu zi pentru a pastra ordinea. Principalul obiectiv al cozii este acela de a oferi unui client specific un loc de asteptare pentru a putea fi servit cat mai repede. Pentru a gestiona cat mai bine o coada ne intereseaza sa minimizam timpul de asteptare al fiecarui client si sa-l facem cat mai eficient. Astfel, daca o coada este goala, clientul va fi pus in aceasta. Cu cat avem mai multe cozi cu atat timpul de servire al fiecarui client va fi mai bun, dar exista un mare dezavantaj in cazul acesta, si anume consumul in exces de memorie. Pentru a rezolva aceasta problema am creat Q cozi. Fiecare coada are atribuit un thread care sa asigure sicronizarea si executarea mai multor evenimente in acelasi timp. De asemenea si client au fost generate aleatory, astfel am generat N clienti. Acesti clienti vor fi pusi in cozi prin clasa *Scheduler*, in care am planificat acest aspect, care alege cea mai buna coada din punctul de vedere al lungimii conzilor si al timpului de servire. Pentru alegerea cozii speficice si potrivite fiecarui client, am creat doua strategii concrete, una in functie de lungime si cealalta in functie de timp.

**3. Proiectare**

In aceasta aplicatie m-am folosit de structure de date precum: *List* si *ArrayList* pentru a stoca liste de cozi( care sunt considerate Servere) si *List* pentru a avea o lista de clienti generate random. Totodata am folosit si asemenea *BlockingQueue*<> pentru cozile de client(Aceasa coada accepta operatii care asteapta ca si coada sa devina “non-empty” la prelucrarea si eliminarea unui element si asteapta ca spatial din coada sa devine disponibil pentru a putea adauga un element). Am creat 3 pachete specific pentru a reprezanta modelul: *MODEL->VIEW->CONTROLLER*. Pachetul *controller*, are clasa *SimulationController*, pachetul *model*, un care care am implementat clasele: *Client, ConcreteStrategyQueue, ConcreteStrategyTime, FilesUtil, Scheduler, SelectionPolicy, Server, SimulationManager, Strategy,* care contin tot ce tine de program in sine si de implementare cozilor si de asemenea si simularea acestori. Si ultului pachet *view*, care contine clasa *SimulationView*, unde este conceput design-ul aplicatie.

**4. Implementare**

In proiect sunt in total 11 clase. Fiecare clasa are rolul ei si o denumire specifica pentru o intelegere mai buna a poiectului in sine. Acestea sunt:

Clasa *“Client”* reprezinta un client in sine. Aceasta are ca si variabile instanta, variabila *id* care este de tipul int, *arrivalTime*(timpul de sosire) care este de asemenea de tipul int, *serviceTime*(timpul de servire), tot de tipul int. In aceasta aceasta am generat prin intermediul Intellij gettere si settere pentru variabilele instate si o metoda compareTo pentru compararea clientilor si de asemenea si metoda *toString* care are rolul de a afisa clinetii mai frumos pentru o mai buna intelegere in cadrul simularii: Acesti client sunt generati random si vor fi introdusi.

Clasa *“ConcreteStrategyQueue”* reprezinta stategia alegerii cozii in functie de lungimea cozii, adica in aceasta clasa se verifica cati client se afla intr o coada. Aceasta clasa implementeaza interfata Strategy. Astfel aceasta implementeaza codul pentru adaugarea unui client “*adaugaClient”*.

Clasa “*ConcreteStrategyTime”* implementeaza codul pentru stategia alegerii cozii in fucntie de timpul de asteptare al clientilor dintr-o coada specifica. Astfel in aceasta clasa se face suma pe fiecare coada in parte a timpilor si pe urma alegand-o pe cea mai mica, astfel urmatorul client ce va trebui adaugat intr-o coada va astepta cat mai putin pentru a fi pus in capul cozii intrucat sa fie servit. De asemenea si in aceasta clasa este implementata interfata Strategy, astfel si aici este implementat codul pentru adaugarea unui client “*adaugaClient”.*

Clasa”FilesUtil” reprezinta clasa pentru a scrie intr-un fisier denumit “*simulationOutput.txt”* date simularii aplicatiei in timp real. Pentru a putea implementa aceasta metoda am folosit metode si obiecte de tipul BufferWriter pe care le-am folosit incluzand bilbiotecile :

import java.io.BufferedWriter;  
import java.io.FileWriter;

Pentru a scrie in fisier am creat un obiect de tip “*BufferWriter”* , am creat un string care sa fie de forma celuia pe care dorim sa il afisam. Adica acesta contine timpul simularii, clinetii care asteapta sa fie introdusi intr o coada, adica care au *arrivalTime*<*timpCurent*  sim ai apoi pe randuri noi fiecare coada cu clientii din acesta la timpul curent al simalarii. Acestui string i-am facut append in fisier folosind apelul motodei implementate in java, care a fost totodata sugerat de intellij. Astfel in fisier nu se vor suprascrie datele de mai sus ci se vor adauga la finalul fisierului, aceasta avand toate datele simularii.

Clasa *“Scheduler”* este clasa unde am creat metodele unde se apeleaza alegerea pentu fiecare client a strategiei si a cozii in care urmeaza sa fie introdus. Aceasta clasa are ca si instante: o lista de servere(queues), numarul maxim de servere, numarul maxim de clienti dintr-un server si strategia aleasa in functie de care se face alegerea cozii in car va fi pus urmatorul client, adica timpul lui de sosire sa fie egal cu timpul de simulare curent.

Clasa sau altfel spui enumeratia *“SelectionPolicy”* contine strategiile implementate mai sus, si anume:

public enum SelectionPolicy {  
 *SHORTEST\_QUEUE*, *SHORTEST\_TIME*}

Clasa *“Server”*  implementeaza interfata runnable intrucat pentru fiecare server am creat un thread. Adica pentru fiecare coada. Aceasta clasa are ca si instante: o coada de client, o perioada de asteptare, timpul petrecut in coada si timpul de servire. In aceasta clasa am implementat metoda *“adaugaClient”*  prin care adaugam clintii in coada si calculam timpul de asteptare pentru coada respective.

Clasa *“SimulationManager”* este clasa in care am implementat metoda pentru a genera random clientii si pentru a ii sorta pe acestea in fucntie de timpul lor de sosire in coada. Totodata in aceasta clasa avem un constructor *SimulatonManager*() care ia din interfata creata datele necesare simularii si le proceseaza astfel incat sa creeze o lista de clienti apeland metoda *generateRandomClients,* si cozile apelanda constructorul Scheduler:

Scheduler(int nrMaxServere, int MaxClientiPerServer)

Si mai apoi alege strategia care va fi folosita pentru adaugarea in cozi a clientilor. Tot in clasa SimulationManager contine si urmatoarele metode: “*calcTimpMediuAsteptare”, “calcTimpMediuServire“(*care calculeaza timpul de asteptare, respective timpul de servire dupa cum le spune numele*), “oraVarf”(*calculeaza momentul cand a fost cea mai mare aglomeratie in cozi), si metoda *“updateFrame*”(updateaza permanent interfata grafica si afiseaza modificarile care se face in cozi la fiecare moment de timp al simularii). Aceasta clasa implementeaza interfata *Runnable,* astfel ca in interiorul ei am creat si metoda run(). In aceasta am implementat codul pentru simularea cozilor in sine, adica pentru a introduce clientii in cozi atunci cand timpul de sosire este egal cu timpul curent de simulare si de scoatere din coada atunci cand a trecut timpul de procesare al clinetului din capul cozii. Tot in acealsa clasa am scris codul pentru actualizarea listei de clienti care asteapta sa fie introdusi in cozi, astfel daca un client a intrat intr-o coada va fi scos din lista de asteptare.

Clasa *“SimulationView”* este clasa unde am creat design-ul interfetei, la aceasta am adaugat butonul de simulare care va incepe simularea in momentul apasarii. Tot aici avem panoul in care se vor afisa rezultatele simularii, si 7 casute de text in care introduce datele: numarul de clineti(N), numarul de cozi pe care le avem la dispozitie(Q), timpul limita, adica timpul maxim de simulare, timpul minim de sosire al clientilor care urmeaza sa fie generate, timpul minim de servire sau de procesare al clientilor care urmeaza sa fie generate random. Timpul maxim de servire sau de procesare al clientilor care urmeaza sa fie generate. Astfel interfata grafica arata astfel:

Graphical user interface, application, table

Description automatically generated

Clasa “*SimulationController”* este clasa in care am implementat o subclasa SimuateListener care implementeaza interfata ActionListener si care creaza ActionListener pentru butonul de simulare din interfata grafica. Aici am creat o variabila *startSimulation* care va fi egala cu ‘0’ daca butonul de simulare din interfata nu a fost apasat sau altfel poate lua valoarea ‘1’ daca apasam butonul de Start simulation de la mouse. Acest buton are rorlul de a verifica daca a inceput sau nu simularea ca sa stim cand putem lua din castuele de test din interfata datele pe care le-am introdus de la tastatura. Daca nu am folosi acest buton ar aparea niste erori pe parcursul rularii programului sau anumite exceptii datorita faptului ca nu avem de unde extrage datele pentru a crea cozile si pentru a genera clientii si deci acestea din urma nu se vor crea si vor ramane nule sau neinitializate putand fi posibile tosui afisari incomplete, fara sens, sau care sa aiba anumite lipsuri. Asadar este foarte important sa avem in vedere daca simularea este sau nu pornita, pentru a putea extrage corect datele din casutele completate de noi din interfata grafica.

**5.REZULTE**

Testele pe care le am incercat sunt cele primate la laborator unde sunt definite si cerintele necesare si minime cat si baremul de notare si cateva mici indicatii legat de proiectarea aplicatiei. Astfel cele 3 teste pe care le am incercat sunt:

N=4

Q=2

tLimit=60sec

tMinArrivalTime=2

tMaxArrivalTime=30

tMinProcessingTime=2

tMaxProcessingTime=4

N=50

Q=5

tLimit=60sec

tMinArrivalTime=2

tMaxArrivalTime=40

tMinProcessingTime=1

tMaxProcessingTime=7

N=1000

Q=20

tLimit=200sec

tMinArrivalTime=10

tMaxArrivalTime=100

tMinProcessingTime=3

tMaxProcessingTime=9

**6.CONCLUZIE**

Realizarea acestei aplicatii a avut in principal ca scop familiarizarea cu conceptul de Threads, dar si de gestionarea lor intr-un mod corect. Acestea sunt deosebit de importante in realizarea unei aplicatii ce necesita multitasking. Aceasta mica simulare este deosebit de interesanta si cuprinde un set mare de instructiuni si de metode ce ajuta la intelegerea in intregime a ce inseamna o simulare a lumii reale. Acest lucru inseamna o simulare a lumii in care traim, a ceea ce ne inconjoara. In jurul nostru nu exista doar actiuni ce sa se intample liniar, una dupa alta, ci mai ales fenomene ce se petrec simultan intr-un mod atat de armonios si care sa fie independente una de cealalta, dar care totusi sa functioneze perfect. Lumea noastra este deosebit de bine realizata, totul se intampla cu un scop, iar din dorinta de a simula acest lucru, programatorii au droit realizarea in paralel a mai multor lucruri deodata. Acest lucru a dus la deschiderea unui nou concept, iar calculatoarele modern functioneaza pe aceasta baza, a lumii noastre. Aceasta aplicatie simpla poate fi dezvoltata intr-o aplicatie ce poate eficientiza timpul de asteptare al unor client la o casa, problema extrem de intalnita in viata noastra aglomerata de zi cu zi. Astfel, am implementat un simulator de cozi pentru clienti generate random. Iar pentru aplicatia realizata am creat si o interfata grafica pritenoasa pentru orice utilizator si cu date care sa il faca pe utilizator sa inteleaga cat de cat ce face programul cat si afisarea din interiorul interfetei este destul de “straight forward” fiind foarte usor de urmarit si de inteles pasii si cum lucreaza aplicatia in spate pentru a putea pune cei N clienti generati random in cele Q liste.

**7.BIBLIOGRAFIE**

* Notite, sfaturi si prezentari din laboratoarele aferente temei;
* <https://dsrl.eu/courses/pt/materials/A2_Support_Presentation.pdf> (prezentarea facuta speciala pentru aceasta tema)
* <https://www.geeksforgeeks.org/>
* <https://www.youtube.com/channel/UCs6nmQViDpUw0nuIx9c_WvA>
* <https://www.jetbrains.com/help/idea/gui-designer-basics.html>
* <https://docs.oracle.com/javase/8/docs/api/java/io/BufferedReader.html>

Diagrama UML:

SimulationController

SimulationManager

SimulationView

SelectionPolicy

FilesUtil

Server

Client

Scheduler

Strategy

ConcreteStrategyQueue

ConcreteStrategyTime