Universitatea Technica Cluj-Napoca

Facultatea de Automatica si Calculatoare

Departamentul de Calculatoare

Semestrul 2

**Tehnici de Programare**

**Tema2-ProcesareCozi -**

Nume:Danciu Maria Alexandra

Grupa:30222

**Cuprins**

1. **Obiectivul temei**
2. **Analiza problemei,modelare,scenarii,cazuri de utilizare**
3. **Proiectare**
4. **Implementare si testare**
5. **Rezultate**
6. **Concluzii , dezvoltari ulterioare**
7. **Bibbliografie**

**1.Obiectivul Temei**

**Cerinta Problemei**

Creati si implementati o aplicatie care simuleaza si analizeaza un sistem de cozi , pentru a determina si minimiza timpul de asteptare a clientiilor.

Asadar am ales ca si cozile sa fie un fir de executie (thread-uri).

Datele ce trebuiesc introduse pentru a putea porni simularea sunt:

* Intervalul minim si maxim de sosire intre clienti
* Intervalul minim si maxim de servire pentru fiecare client
* Numarul de cozi
* Timpul de simulare
* In plus eu am ales sa introduc numarul de clienti ce se vor aseza la cozi

Ca si date de iesire avem:

* Durata medie de asteptare
* Durata medie de serviciu
* Durata medie in care cozile sunt goale
* Ora de varf
* Evolutia Cozilor

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

Principala cerință a acestui proiect o reprezintă funcționarea corectă asimulatorului, dispunerea corectă a clientilor la cozi și afișarea acestora în formatul așteptat .Prin funcționare corectă înțelegem generarea de rezultate corecte, dar și prevederea situatiilor in care simularea se poate termina mai repede, sau a cazurilor in care un client nu mai poate sa intre in coada.Cerințe:

- citirea datelor din fisier;

- generarea aleatoare a celor N cozi pe baza datelor citite din fisier;

-multithreading-un singur thread la o coada;

-inchiderea si deschidere dinamica a cozilor in functie de continutul acestora;

- calcularea timpului mediu de asteptare la coada;

-fisier .jar pentru a se rula din linia de comanda;

-realizarea de teste pentru verificarea corecta a simularii;

   Obiectivele secundare, mai exact pașii care trebuie urmați pentru îndeplinirea obiectivului principal, în vederea realizării aplicației finale, sunt următorii:

* Citirea și validarea datelor citite din fișier, transmiterea datelor citite din fișier la simulator, urmând că apoi să se genereze în mod aleator timpii minimi și maximi de procesare a datelor și cele n cozi, apoi crearea unui thread asupra clasei care realizează simularea, pentru a realiza simularea efectivă a cozilor.
* Crearea celor n cozi si a cate unui thread pentru fiecare coada in parte.
* Generarea random a datelor legate de clientii care vor participa la cozi si transmiterea acestora catre cozi
* Stabilirea functionalitatii unei cozi
* Stabilirea functionalitatii metodei de dispunere a clientilor la coada dupa una dintre cele doua metode: Clientul va fi trimis la coada cu timpul de asteptare minim sau Clientul va fi trimis la coada care in acel moment va avea cel mai mic numar de clienti
* Crearea de functii care vor deschide si afisa datele corespunzatoare in fisier
* Stabilirea modului in care se va realiza simularea
* Calcularea timpului mediu de asteptare la coada

**Fundamente Teoretice**

Ce este o coada ?

Cozile sunt frecvent intalnite atât în lumea reală cat și în modele. Obiectivul principal al unei cozi este de a oferi un loc pentru un "client" să aștepte înainte de a primi un "serviciu". Astfel o coada funtioneaza dupa principiul FIFO ( first in , first out ).

Managementul sistemelor bazate pe cozi este intresat în minimizarea timpului de asteptare pentru "clienții" săi care sunt în așteptare în cozi. O modalitate de a minimiza timpul de așteptare este de a adăuga mai multe servere, adică mai multe cozi in sistem (fiecare coadă este considerată ca având un procesor asociat), dar acesta abordare crește costurile furnizorului. Atunci când un nou server se adaugă clienții ce sunt in așteptare vor fi distribuiti în mod egal intre toate cozile de așteptare curente disponibile.

Mai departe sa vedem ce este un thread : Conceptul de thread (fir de execuție) definește cea mai mică unitate de procesare ce poate fi programată spre execuție de către sistemul de operare. Este folosit în programare pentru a eficientiza execuția programelor, executând porțiuni distincte de cod în paralel în interiorul aceluiași proces. Cateodata insa, aceste portiuni de cod care constituie corpul threadurilor, nu sunt complet independente și în anumite momente ale executiei, se poate intampla ca un thread sa trebuiasca sa astepte executia unor instructiuni din alt thread, pentru a putea continua executia propriilor instrucțiuni. Aceasta tehnica prin care un thread asteapta executia altor threaduri înainte de a continua propria executie, se numeste sincronizarea threadurilor.

Finalmente aplicatia trebuie sa simuleze un numar de clienti care sunt asezati in cozi in funtie de timpul lor de sosire la “casa”.Astfel fiecare client asteapta un anumit timp inainte de a fi servit daca acesta nu este singur in coada.Clientii creati prezinta date random de la un client la celalalt in functie de de datele de intrare(Minim , Maxim).

**Modelare**

Solutia aleasa pentru a crea aceasta aplicatie este urmatoarea:Am un numar minim de clase si anume. O clasa *Client*  care contine *ID*-ul fiecarui client si un timp de sosire si timpul de servire . Cum aplicatia trebuie sa fie cat mai aproape de realitate , datele pentru clienti sunt generate random in functie de datele de intrare .

Acestui clienti trebuie pusi in cozi , astfel am o clasa *Queue* in care avem o lista de clienti , asupra carei aplicam doar operatii specifice FIFO.

Avem clasa *Simulator* care este tot un thread si se refera la asa-zisele “case de marcat” .

**Cazuri de Utilizare**

Pentru a simula aceasta aplicatie , se vor extrage datele din 3 fisiere txt :in-test-1,in-test-2,in-test-3,pentru a realiza 3 teste diferite.Rezultate sunt stocate tot in 3 fisiere txt denumite out-test-1,out-test-2,out-test-3.

**3.Proiectare**

**Diagrama cu Cazurile de Utilizare**

O astfel de diagrama ne arata ce poate sa faca utilizatorul cu aceasta aplicatie.In cazul acestui proiect utilzatorul ofera datele necesare pentru ca aplicatia sa poata rula.Dupa citirea datelor din fisier evolutia cozilor este afisata fisierul corespunzator de iesire.

Dupa cum se observa se vor extrage timpul Minim respectiv Maxim de Sosire . Acesta se refera la ce diferenta de timp vin clientii si se aseaza in coada . Mai departe Timpul Minim si Maxim de Servire , se refera la un interval concret si din care se aloca o valoare random clientului pentur servire , este bine sa fie un interval destun de mic pentru ca , clientii sa isi termine servirea.Numarul de cozi se refera simplu la numarul de cozi accesibile.Numar de clienti , se refera la cati clienti v-or merge in magazin pe perioada simularii . Timpul de simulare este timpul de simulare al aplicatiei.

Întrucât aplicația realizeaza o simulare, întâlnim clasa Simulator care are implementate metodele necesare simularii, cât și metode care operează asupra cozilor pentru oprirea simularii mai devreme.Aceasta clasă poate fi considerată clasa de baza a aplicației, împreună cu clasa Main, care incepe citirea din fisier si crearea threadu-ului de simulare dupa citirea datelor.

[Concurența](https://ro.wikipedia.org/wiki/Concuren%C8%9B%C4%83_(informatic%C4%83)) este abilitatea de a rula mai multe părți ale unui program sau mai multe programe în paralel. Cuvantul cheie synchronized asigura

* că numai un singur fir poate executa un bloc de cod, în același timp;
* că fiecare thread de înregistrare a unei bloc sincronizat de cod vede efectele tuturor modificărilor anterioare, care au fost pazite de aceeași blocare.

Sincronizarea este necesara pentru accesul reciproc exclusiv la blocuri de comunicare și de încredere între fire.

Pentru a asigura aceasta concurenta, pe langa structurile de date adecvate s-au folosit si cuvantul cheie synchronized asupra anumitor metode.

**Diagrama UML**

Aplicatia prezinta urmatoarele clase : Client , Queue ,Simulator si PdfWriter.Unde Simulator se refera defapt la “casele de marcat” si desigur la interfata aplicatiei.In plus fata de acestea , am clasa Tests folosita la testele Junit.

Astfel in cadrul diagramei UML avem urmatoarele realtii intre clase . Intre clasele *Client* si *Queue* avem o realatie de agregare , clasa *Client* fiind unitatea de baza a clasei *Coada* , iar aceasta fiind formata dintr-un ArrayList de clienti.Intre clasele Queue *si Simulator* exista tot o relatie de agregare , clasa  *Simulator* avand un vector de cozi .

**Diagrame de Secventa**

Diagramele de secventa mai sunt cunoscute ca si **MSC** (Message Sequence Charts). Acestea prezinta temporal interactiunile intre obiecte.

Rolul acestor diagrame este de a modela scenariile posibile , pentru a descoperi intefetele necesare fiecarui obiect si pentru a valida fiecare interfata cu adevarat utilizata.

**Structuri de Date**

In acest proiect am folosit mai multe tipuri de date , pornind de la cele primitive int,double,etc. pana la liste , ArrayList<> .

Ca si lista am folosito pe urmatoarea *ArrayList <Client> coada = new ArrayList <Client>();* in clasa *Coada* si aceasta se refera la coziile propriu zise.

Mai avem o lista de accelasi tip in clasa *Controller* in care stocam clientii pe care ii cream random pana cand le vine timpul sa se puna la coada.

**Proiectare Clase**

Clasa Client

Clasa *Client* este clasa care se refera la datele clientului si anume prezinta urmatoarele atribute: *ind ID* – fiecare client prezinta un ID pentru diferentiere- , *int timpSosire* -timpul de sosire al fiecarui client si adica cand se pune la coada- si ultimul atribut *int timpDeServire* -fiecare client prezinta un timp de servire personal-.

Ca si metode avem *Getter-ele* si *Setter-ele* specifice fiecarui atribut.

Clasa Queue

Clasa *Queue* este clasa care se refera la cozi propriu zise si anume o lista de clienti , ce functioneaza ca si FIFO.Clasa prezinta ca si atribut *ArrayList <Client> coada.*

Aceasta clasa extinde clasa *Thread* si astfel este un fir de executie care va rula concurenct cu celelalte cozi-threaduri-.

Ca si metode avem meotda *addClient(Client client)* care adauga un client la coada , desigur doar la sfarsitul ei.

Metoda *getSize()* , ce returneaza dimensiunea cozii.

Metoda *ArrayList <Client> geQueue()* ce returneaza coada propriu zisa.

Finalmente avem metoda *run()* , metoda specifica *Thread-urilor* , ce porneste threadul.Aceasta metoda astepta pentru un timp real -*sleep(1000)*- daca nu este nimeni in coada iar daca coada nu este goala va astepta pentru un timp reprezentat de *serviceTime* a clientului \* *1000* . In plus se calculeaza timpul de asteptare din coada.Dupa *sleep()* clientul pleaca si se afiseaza un asemenea mesaj iar dupa aceea se face *coada.remove(0)* .

Clasa Simulator

Clasa *Simulator* este clasa care se refera la asa-zisele “case de marcat” si prezinta urmatoarele atribute: *int timpServireMinim* si *timpServireMaxim* -date necesare pentru crearea random a timpilor pentru clienti- ; *int timpSosireMinim* si *timpSosireMaxim* -date necesare pentru crearea random a timpilor pentru clienti - ; *int timpDeServireMediu* -ce ne va oferi timpul mediu de servire- ; *int nrCozi* -cate cozi v-om avea la “casele de marcat”- ; *int timpReal* -timpul real din simulare si adica la ce secunda suntem din simulare incepand de la secunda 1 < timpSimulare-; *int timpSimulare* -timpul de Simulare al aplicatiei- ; *int oraDeVarf* -v om stoca ora de varf a aplicatiei- ; *Coada[] cozi* -se refera la cozile pe care le v om avea - ; *Random random* -folosit pentru a genera date random- ; *int timpCoadaGoala* -v om stoca timpul mediu pentru care cozile sunt goale- ; *String text* -folosit pentru Logger-.

Aceasta clasa extinde clasa *Thread* si astfel este un fir de executie care va rula concurenct cu celelalte threaduri.

Ca si metode pe langa *Gettere* si *Settere* , avem metode precum : *cresteTimpDeAsteptare(int timp)*  ce creste timpul de asteptare cu acel parametru.

Metoda *createClients()* creaza clienti cu date random si ii adauga intr-o lista temporara din care se vor extrage clientii cand trebuie sa se pune in cozi.Tot in acceasi metoda are loc start-ul fiecarui cozi , si anume *cozi[i].start* .

Metoda *checkIfEmpty()* verifica daca vreuna dintre cozi este goala , iar daca este are loc cresterea unui contror ca va fi impartit la final la numarul de cozi iar valorea stocata in *timpCoadaGoala*.

Metoda *getTimpServireMediu()* aflta timpul de servire mediu din cozi si il stocheaza in atributul *timpDeServireMediu.*

Metoda *showQueues()* afiseaza evolutia coziilor , ce clienti sunt in cozi in fiecare secunda a simularii .

Metoda *run()* porneste threadul si practic aplicatia.Initial creaza toti clientii , ii stocheaza in lista , si incepe un for de la 1 care merge pana la timpulDeSimulare .Se verifica daca timpul de sosire al vreunui client este egal cu cel Real si daca da , atunci acel client va fi adaugat la coada minima . Mai departe se verifica daca coziile sunt goale , ora de varf si desigur datele necesare afisate in *Logger* , plus printarea evolutiei coziilor .Dupa ce timpul de simulare este gata se afiseaza in *Logger* timpul mediu de asteptare , de servire , ora de varf si timpul mediu in care cozile au fost goale.

In clasa PdfWriter folosesc metoda main pentru modelarea datelor.

Ex:

BufferedWriter out = **new** BufferedWriter(**new** FileWriter("out-test-1"));

String line1=Files.*readAllLines*(Paths.*get*("in-test-1.txt")).get(0);//nr clienti

String line2=Files.*readAllLines*(Paths.*get*("in-test-1.txt")).get(1);//nr cozi

String line3=Files.*readAllLines*(Paths.*get*("in-test-1.txt")).get(2);//simulare

String line4=Files.*readAllLines*(Paths.*get*("in-test-1.txt")).get(3);//arrival

String line5=Files.*readAllLines*(Paths.*get*("in-test-1.txt")).get(4);//service

String[] words4 = line4.split("\\W+");

String[] words5 = line5.split("\\W+");

Integer clients=Integer.*parseInt*(line1);

Integer queues=Integer.*parseInt*(line2);

Integer time=Integer.*parseInt*(line3);

Integer minArrival=Integer.*parseInt*(words4[0]);

Integer maxArrival=Integer.*parseInt*(words4[1]);

Integer minService=Integer.*parseInt*(words5[0]);

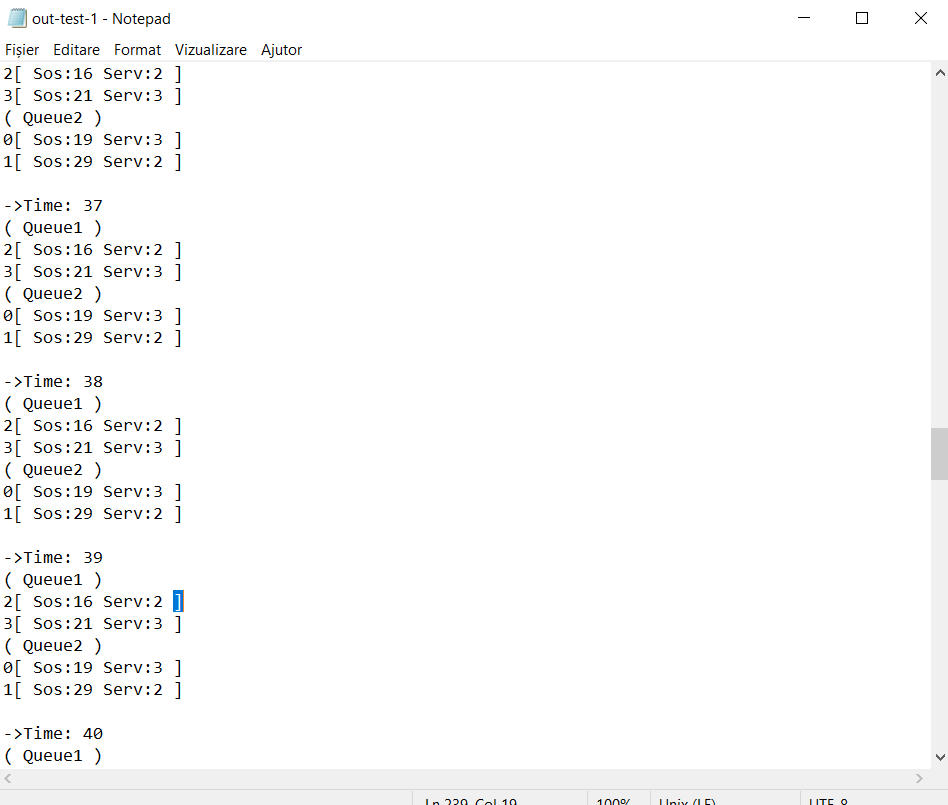
Integer maxService=Integer.*parseInt*(words5[1]);

Simulator ctrl = **new** Simulator(clients,queues,time,minArrival,maxArrival,minService,maxService,out);

ctrl.start();

Utilizez un bufferd writer pentru afisarea datelor si scrierea acetora in txt file.Din fisierul txt in-test-1 citesc fiecare linie si o salvez intr-un string,ulterior transfor string-ul in Int pentru a transmite datele spre simulare.Linia 4 si 5 din fiecare fisier contine 2 informatii-respectiv minArrivalTime,Max arrivalTIME si minServiceTime ,maxService time,separate prin virgula.Realizez o splitare a liniei daca intalneste orice altceva inafara de cuvant.Ulterior aplic start pentru pornirea aplicatiei.

**5.Rezultate**

Dupa rularea testelor am obtinut rezultatele in 3 fisiere txt.

Celelate 2 fisiere sunt asemenea dar au alte date de intrare si respective alte date de iesire.

**6.Concluzii**

In concluzie , au fost indeplinite obiectivele cerute in realizarea proiectului , chiar mai mult , prin printarea evloutiei cozilor.

Din punct de vedere al dezvoltarii ulterioare , am putea adauga o baza de date pentru clienti si un numar mai mare de cozi , etc.

In plus se poate adauga o clasa Angajati care sa prezinte atribute precum vechime , experienta , grad de obososeala , etc . Astfel in functie de experienta pe care o are sa se miste mai repde si cu cat lucreaza mai mult sa creasca gradul de oboseala si sa se miste mai incet.

**7.Bibliografie**

-Documentatia Java:

<https://docs.oracle.com/javase/7/docs/api/>

-Pentru Threaduri:

<https://www.youtube.com/watch?v=KUdro0G1BV4&list=PLBB24CFB073F1048E&index=5>

-Pentru Probleme

<http://stackoverflow.com/>