**Facultate: Automatică și Calculatoare**

**Specializare: Calculatoare și Tehnologia Informației**

**Disciplina: Tehnici de Programare**

**Tema: Simulator de Cozi**

Student : Herman Felician-Nicu

Seria: B

Grupa: 30229

**Cuprins**

1.Obiectivul temei

1.1 Obiectivul Principal

1.2 Obiective Secundare

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

3. Proiectare

3.1 Structuri de Date

3.2 Proiectare Clase

3.3 Diagrama UML

4. Implementare

4.1 Clasa Task

4.2 Clasa Server

4.3 Clasa Scheduler

4.4 Clasa SimulationManager

4.5 Clasa ConcreteStrategyTime

4.6 Clasa Main

4.7 Interfata Strategy

5. Rezultate

6. Concluzii

7. Bibliografie

**1.Obiectivul temei**

**1.1 Obiectivul principal:**

Obiectivul principal al temei este reprezentat de proiectarea unui simulator de cozi menit sa repartizeze un anumit numar de clienti sau taskuri(n) intr-un anumit numar de cozi(q) intr-un anumit timp de procesare total(finalTime), stabilit intial, folosind fisierele de intrare. Reprezentarea clientilor (taskurilor) o sa fie afisata in fisierele de iesire(acest lucru poate fi modificat ulterior in functie de necesitati).

**1.2 Obiective secundare**:

|  |  |  |
| --- | --- | --- |
| Obiectiv Secundar | Descriere | Capitol |
| Scenarii | In computere, un scenariu este o naratiune a interactiunilor dintre roluri ( actori in UML ) si sistem. In cazul nostru generarea fisierului de iesire pe baza fisierului de intrare. | 2 |
| Alegerea structurilor de date | Structurile de date folosite pentru a duce la capat obiectivul principal | 3 |
| Impartirea pe clase | Am ales o impartire pe clase astfel incat cozile sa ruleze independent de programul principal, folosind threaduri. | 3 |
| Dezvoltarea algoritmilor | Vor fi descrise structurile de date necesare pentru atingerea obiectivului principal, schema UML ( diagrama de clase ) precum si algoritmii folositi pentru realizarea actiunilor de repartizare si procesare a clientilor(Taskurilor). | 4 |
| Implementarea solutiei | Vor fi descrise fiecare clasa cu campurile si metodele importante . | 4 |
| Testare | Vor fi descrise cateva scenarii de testare a generarii random a celor N Taskuri si repartizarea respectiv procesarea acestora de catre cele Q cozi. | 5 |

**2.Analiza problemei**

Având în vedere contextul social creat de pandemia de COVID-19, această aplicație are capacitatea de a facilita procesul de servire a clienților în cadrul societăților comerciale, astfel evitându-se aglomerarea.

Aplicatia o sa functioneze astfel, in fisierele de intrare o sa fie introdusi pe rand intregi reprezentand:

-pe prima linie N, numarul de clienti

-pe a doua linie intregul Q, numarul de cozi

-pe a treia linie timpul maxim de simulare

-pe a patra linie, doi intregi, timpul minim la care clientul ajunge si timpul maxim la care un client poate ajunge

-pe ultima linie, doi intregi, timpul minim de procesare si timpul maxim de procesare

Exemplu:

4 –numarul de Task-uri(N)

2 –numarul de cozi(Q)

60 –timpul maxim(timeLimit)

2 30 –minArrivalTime maxArrivalTime

2 4-minProcessingTime maxProcessingTime

Dupa ce datele sunt procesate, o sa fie generat un fisier de iesire in care o sa fie afisata programarea fiecarui Task in anumite cozi in funntie de currentTime si de arrivalTime al fiecarui Task. Daca un Task o sa aiba timpul in care ajunge(ArrivalTime) mai mare decat timeLimit, acesta nu o sa fie procesat si o sa ramana in waitingQueue.

Am decis ca timpul de procesare de la etapa la etapa sa fie un intreg, adica o secunda pentru a putea fi mai usor de urmarit evolutia cozilor si a listei de asteptarea.

**3.Proiectarea**

**3.1 Structuri de date**

In ceea ce priveste structurile de date, am incercat sa le folosesc pe cele care s-ar potrivi cel mai bine in functie de contextul utilizarii lor.

Astfel, fiecare coada este stocata intr-un asa-zis Server care sunt instantiate in clasa Scheduler sub forma unui ArrayList pentru a fi folosite in SimulationManager sub functionarea threadului principal.

Pentru sirurile de taskuri(din fiecare coada), am ales sa folosesc BlockingQueue deoarece este o structura folosita in programele cu threaduri. Avantajul acestei structuri este ca sincronizeaza automat accesul threadurilor asupra continutului, fara a mai fi nevoie synchronized. De asemenea, daca queue-ul este gol, iar threadul doreste sa ia element din ea, nu se va returna Null, ci threadul va fi pus in asteptare pana cand este disponibil un element.

Taskurile(clientii din lista de asteptare) o sa fie generate random pe baza datelor din fisierul de intrare si o sa fie memorate intr-un ArrayList de unde o sa fie repartizate in cozi pe baza arrivalTime-ului in momentul in care acesta este egal cu timpulCurent al Threadului principal. Coada in care o sa fie repartizat este aleasa pe baza strategiei implementata de Clasa ConcreteStrategyTime.

**3.2 Proiectare Clase**

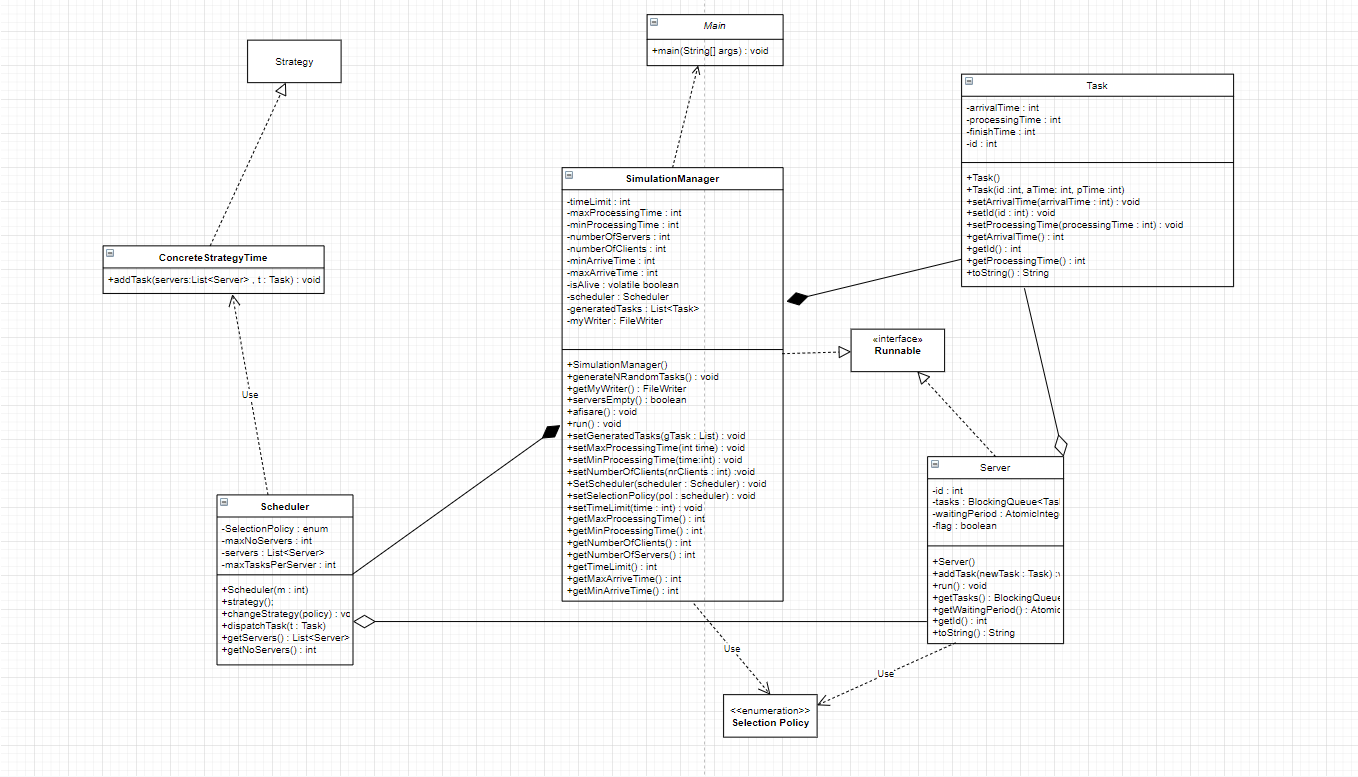
Tema contine 6 clase si o interfata cu care o sa lucram pentru a rezolva cerinta, mai exact pentru a ajunge la un rezultat in functie de cererile utilizatorilor. In plus mai exista o clasa ConcreteStrategyQueue care implementeaza deasemenea interfata Strategy insa in care nu am implementat vreo functie, acest lucruu fiind o extindere ulterioara a aplicatiei.

Clasa cu care am inceput este clasa **Task** care imi reprezinta un anume client avand un id unic, un timp la care ajunge si e pregatit sa fie procesat(arrivalTime) si un timp de procesare(processingTime). Mai apoi, clasa **Server** reprezinta una dintre cozile in care o sa fie procesate Taskurile, fiecare server, sau coada avand un waitingTime, un id si un BlockingQueue de Taskuri, adica o lista de Taskuri pe care o sa le proceseze.Clasa **Scheduler** instatiaza numarul de cozi si strategia dupa care taskurile o sa fie repartizate in cozi.

Clasa care se ocupa de firul principal al programului este clasa **SimulationManager** care pe o durata fixa ruleaza si repartizeaza Taskurile aflate in lista de asteptare in cozile goale sau cozile care au un waitingTime minim, stabilit de strategia **ConcreteStrategyTime** care implementeaza interfata **Strategy**.

In final, clasa **Main** creeaza drumul spre fisierele de intrare(le creeaza implicit) si iesire(am generat un string in care am pus calea spre fisierele de iesire din folderul proiectului), deasemenea instantiaza un nou SimulationManager, creeaza un Thread pe baza simulationManagerului creat si il porneste.

**3.3 Diagrama UML**

****

**4. Implementare**

**4.1 Clasa Task**

In aceasta clasa pur si simplu creez cate un Task nou folosind constructorul. Fiecare Task o sa aiba un id diferit, iar ArrivalTime si processingTime in interiorul intervalului citit din fisier.

**4.2 Clasa Server**

Fiind o clasa care lucreaza cu Threaduri, aceasta implementeaza interfata Runnable, avand ca variabile de clasa lista de Taskuri si waitingPeriod reprezentand perioada de asteptare a fiecarei cozi. Mai exact, clasa Server reprezinta o coada, deci o sa avem instantiate Q servere(q fiind citit din fisierul de intrare). Aceasta clasa este capabila de a realiza adaugarea unui nou task in lista proprie de taskuri si deasemenea procesarea taskurilor aflate deja in coada. Procesarea reprezinta mai exact punerea pe „sleep” a threadului(o secunda, timp ales de noi), timp in care taskului o sa ii fie decrementat timpul de procesare cu 1 si deasemenea waitingPeriod-ul o sa scada deoarece un task a fost procesat pentru o secunda.

**4.3 Clasa Scheduler**

Putem asimila aceasta clasa cu un planificator deoarece pe baza strategiei implementata in clasa ConcreteStrategyTime se adauga cate un Task in Servere(adica in cozi). Functia dispatchTask are rolul de a adauga in servere taskul primit ca si parametru, aceasta functie fiind folosita in SimulationManager in momentul gasirii unui Task care are timpul de start egal cu timpul curent.

**4.4 Clasa SimulationManager**

Clasa de baza in acest program,extinde clasa Thread deci este implementata o metoda run() pe baza careia funcioneaza Threadul care controleaza currentTimeul, dupa care sunt sincronizate celelalte Threaduri ale serverelor. In aceasta clasa sunt declarate toate datele necesare unei bune functionarii a proiectului, datele citite din fisier pe baza carora sunt generate cele N randomTasks. Deasemenea aici este implementata si functia propriu-zisa de generare a celor N randomTasks, in fcuntia generateNRandomTasks() pe care am implementat-o generand de N ori cate un id unic si deasemenea generand random cate un intreg in fiecare interval precizat in fisierul de intrare. Mai apoi, am implementat functia de afisare in fisier si functia de citire din fisier(in constructor).

Fiind o clasa care foloseste un Thread, implementeaza Runnabla prin functia run(), functie care proceseaza Taskurile astfel:

-variabila currentTime reprezinta timpul curent, dupa care o sa fie selectate Taskurile iar acel „flag” isAlive o sa fie folosit pentru a opri Threadul, o sa vedem in continuare cazurile in care acesta trebuie sa fie oprit.

-in primul rand, dupa ce Threadul este pe sleep, pentru o secunda, verificam daca timpul curent conincide cu timpul vreunui Task din lista noastra generata random.

-daca se gaseste un Task care corespune, acesta este adaugat in lista de cozi pe baza Scheduler-ului care foloseste strategia „Time”.

-dupa ce s-a adaugat in coada, este eliminat din lista de Taskuri, urmand sa fie procesat de o anumita coada.

-in momentul in care lista de Taskuri a fost parcursa, se afiseaza Timpul, Waiting List(adica taskurile generate random) si cozile .

-dupa afisare, verificam conditiile pentru a stii daca trebuie sa oprim Threadul, sau inca mai avem de lucru cu el

-conditiile ca Threadul sa isi fi terminat treaba sunt: lista de Taskuri sa fie goala(adica Taskurile random sa fie fiecare in coada), cozile sa nu mai aiba Taskuri de procesat sau timpul curent sa fii depasit timeLimit.

-in momentul in care Threadul se inchide, programul isi termina treaba si se opreste.

**4.5 Clasa ConcreteStrategyTime**

Implementand interfata Strategy, aceasta clasa verifica care dintre Servere are timpul de asteptare cel mai scazut, fiind potrivit pentru a „primi” un nou Task pentru procesare, in cazul in care sunt mai multe cozi cu acelasi waitingPeriod, o sa fie selectata coada cu index cel mai mic.

**4.6 Clasa Main**

Clasa „de baza” unde am deschis fisierele de intrare si am creeat calea spre cele de iesire. Deasemenea am instantiat SimulationManager pentru fiecare fisier de intrare, afisand rezultatul in fisierele de iesire, bineinteles odata cu pornirea Threadului principal. Fiecare fisier de intrare este creeat aici, pentru fiecare dintre cele 3 teste iar calea catre cele 3 fisiere de iesire este transmisa ca parametru clasei SimulationManager.

**4.7 Interfata Strategy**

Aceasta interfata contine doar o metoda, „addTask” care primeste lista de Servere(cozi) si un anumit Task. Clasa ConcreteStrategyTime implementeaza aceasta interfata care ajuta la repartizarea taskurilor in serverul potrivit.

**5.Rezultate**

Programul genereaza repartizarea in fisierele de iesire Test\_Out, Test\_Out2 si Test\_Out3 pentru fiecare fisier de intrare. Pentru testare suplimentara se poate modifica continutul oricarui fisier de intrarea iar rezultatele o sa fie afisate in fisierele de iesire. Deasemenea am creea si .jar pentru a putea fi apelat programul din CMD.

**6.Concluzii**

Sincer, nu credeam ca o sa duc la capat vreodata aceasta tema fiind prima data cand am lucrat cu Threaduri insaaa cu putin ajutor am reusit sa o termin si mai mult sa functioneze chiar bine. Pot spune ca a durat mai mult pana am inteles functionalitatea si legaturile dintre clase decat implementarea codului. A fost foarte de ajutor acel pdf, deoarece ne-a ghidat si nu ne-a lasat prea mult loc de interpretare. Mai mult, am invatat cum sa citesc si sa scriu in fisier in Intellij, nemaifacand asta pana acum decat in C/C++. Am invatat destul de multe din aceasta tema si chiar sunt mandru de ce am realizat. Mai exact pot spune ca acum am inteles si m-as descurca cu sincronizarea si lucrul cu Threaduri, acestea fiind foarte prezente in viata de zi cu zi si in aplicatiile pe care noi le folosim.Ba mai mult, acestea pot fi sincronizate astfel incat sa creeze foarte multe aplicatii extrem de utile. Fiind presat de timp, nu am apucat sa calculez avem averageTimeWaiting, insa cred ca esenta am reusit sa o rezolv si sa implementez ce era mai greu.

Mai mult, mi s-a parut un proiect interesant deoarece are o aplicabilitate in viata de zi cu zi, avand la baza un principiu dupa care functioneaza foarte multe supermarketuri sau fast-fooduri in momentul de fata. In plus, situatia in care ne aflam acum a ridicat gradul de interes pentru mine realizand ca aplicatiile de genul, bineinteles mai dezvoltate pot ajuta omenirea si ajuta la o mult mai buna organizare.

Ca dezvoltare ulterioara m-am gandit ca aceasta aplicatie ar putea fi imbunatatita adaugandu-i o interfata grafica care sa permita utilizatorului sa introduca mult mai usor datele pe care noi le citim din fisier si deasemenea sa fie creata o noua strategie pe baza careia sa fie selectati clientii.

Ba mai mult, unele Taskuri ar putea avea un camp suplimentar(adica unii clienti sa fie privilegiati) deci fiind repartizati si procesati primii.

**7. Bibliografie**

<file:///C:/Users/My-Pc/Downloads/Java_Concurrency.pdf>

<https://www.geeksforgeeks.org/different-ways-reading-text-file-java/>

<https://www.w3schools.com/java/java_files_create.asp>