QUEUES SIMULATOR

**Student : Vesa Bianca**

**Grupa : 30227**

1. **Obiectivul temei**

Tema acestui proiect este implementarea unei aplicatii care sa simuleze un sistem de gestionare a unor cozi, cu scopul de a minimiza timpul de asteptare a fiecarui client. Pentru indeplinirea acestei cerinte am urmat modelul pus la dispozitie in suportul lucrarii de laborator, si am impartit obiectivul principal in mai multe obiective secundare :

* Identificarea claselor de implementat, care sunt subiectii acestui scenariu : Server si Client
* Separarea fizica, in clase diferite, a cerintelor de creare si distribuire a obiectelor necesare (Scheduler), precum si implementare a logicii gestionarii operatiilor de efectuat (ConcreteStrategyTime)
* Simularea efectiva a cerintei => clasa ClientGenerator

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

Cazul de utilizare al acestei aplicatii este simularea (intr-un timp *tSimulation*) a *N* clienti, sositi pentru serviciul oferit, intrarea acestora in cele *Q* cozi, asteptarea, servirea si, in final, parasirea cozilor. Toti clientii sunt generati la inceperea simularii si sunt caracterizati de anumiti parametri : *ID* (un numar intre 1 si N), *tArrival* (timpul de simulare cand sunt pregatiti sa mearga la coada dupa terminarea cumparaturilor) , *tService* (intervalul de timp sau durata necesara pentru servirea clientului de catre casier, adica timpul de asteptare cand clientul de afla in fata cozii) si *waitingTime* (timpul pe care il asteapta fiecare client din momentul in care este asezat la coada, pana cand ajunge in fata pentru a fi servit). Aplicatia urmareste acest ultim parametru, calculand media timpului de asteptare petrecut la coada. Fiecare client este adaugat la coada cu un timp minim de asteptare.

Pentru utilizarea acestui program se cere sa se citeasca din fisier datele de intrare : numarul de clienti, numarul de cozi, intervalul de simulare, intervalul de sosire la coada (minim si maxim), si intervalul de servire a fiecarui client (minim si maxim).

Programul pune la dispozitie urmatorul scenariu de utilizare :

* Utilizatorul executa fisierul .jar, oferind ca argumente fisierul de intrare, din care se citesc datele si fisierul de iesire, in care vor fi disponibile rezultatele finale
* Firul de executie principal al programului executa operatiile pe cozi si clienti intr-un timp *tSimulation* , citit din fisier
* Dupa efectuarea operatiilor , in fisierul de iesire dat ca argument , sunt scrise rezultatele simularii

1. **Proiectare (decizii de proiectare , diagrame UML , structuri de date , proiectare clase , interfete , relatii , packages , algoritmi , interfata utilizator)**

Elementele de baza ale acestei aplicatii sunt clientii si cozile , sau casierii. Astfel, o decizie de implementare luata este de lucra cu cu doua clase separate : Client ( modeleaza obiecte ce au ca parametri ID-ul unic al fiecarui client, timpul in care ajunge la coada, timpul in care este servit la coada si timpul asteptat pana cand este servit ) si Server (primeste ca parametri numarul maxim de clienti pe care trebuie sa ii serveasca, si numarul de ordine al cozii).

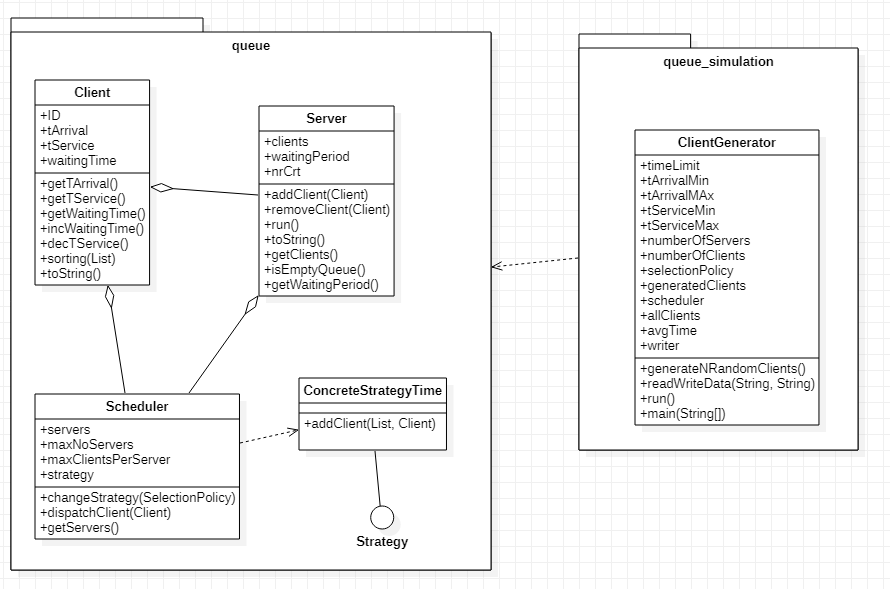
Clasa care se ocupa cu crearea cozilor , adaugarea de client si eliminarea lor este Scheduler, avand rolul de a implementa logica simularii, si fiind caracterizat de un *waitingPeriod ,* care indica timpul necesar de asteptare in coada curenta. Aceasta clasa pune in aplicare si strategia de distribuire a clientilor in cozi , utilizand clasa ConcreteStrategyTime. Cea din urma modeleaza alegerea cozii in care clientul sa fie introdus : se cauta in lista de cozi cea cu timpul de asteptare cel mai mic. De asemenea , extinde interfata Strategy, care lasa programatorului posibilitatea sa extinda acest proiect dupa alta logica decat cea a timpului minim de asteptare, fara a fi nevoie de modificarea claselor deja existente.

Clasa ClientGenerator are scopul de realiza, cu ajutorul unui obiect a clasei Thread, simularea propriu-zisa.

Toate clasele sunt impartite in doua pachete : queue si queue\_simulation, care le separa in functie de rolul fiecareia in proiectul de ansamblu.

In ceea ce priveste structurile de date utilizate , principalele sunt BlockingQueue si AtomicInteger , care asigura caracteristica de Thread Safety. Am folosit si instante ale colectiei List , pentru a manui cu usurinta un numar mare si variabil de clienti si casieri. Fiind o structura dinamica, putem sa adaugam, sa eliminam obiecte din lista, precum si sa o parcurgem eficient , cu ajutorul metodelor pe care le pune la dispozitie.

In cele din urma , voi atasa diagrama UML aferenta proiectului :



1. **Implementare**

* Clasa Client

Campurile unui client sunt incapsulate si au vedere private , fiind disponibile prin intermediul metodelor getter. Parametri primiti la crearea obiectului Client sunt ID (numar unic de identificare) , tArrival (momentul in care clientul a ajuns la coada) , tService (timpul necesar pentru procesarea unui client) si waitingTime (timpul asteptat de fiecare client pana la momentul in care este servit) , care se initializeaza la 0.

Metodele puse la dispozitie in aceasta clasa , exceptand gettere si settere , sunt : incWaitingTime() , care incrementeaza timpul de asteptare pentu clientii care au pe altcineva in fata la coada ; sorting() , care ajuta la sortarea clientilor in functie de tArrival si toString() , care returneaza obiectul Client scris sub forma *(ID, tArrival, tService).*

* Clasa Server

Un server este caracterizat de un ArrayBlockingQueue de clienti , numarul de ordine si un AtomicInteger care pastreaza perioada de asteptare la coada curenta.

In interiorul acestei clase gasim metodele :

* addClient : adauga in array-ul de clienti un nou client primit ca parametru si creste perioada de asteptare , cu tService , a acestui client
* removeClient : sterge din array un client primit ca parametru
* getClients : returneaza ca sir de clienti, ArrayBlockingQueue-ul
* isEmptyQueue : ne informeaza daca in coada curenta se afla sau nu clienti
* toString : afiseaza coada curenta , urmata de cuvantul ‘closed’ in cazul in care nu are client
* run ( clasa Server implementeaza metoda Runnable , avand un fir de executie care are rolul de a permite modificarea mai multor serveri in mod concurent) : threadul este deschis atat timp cat coada de client nu este goala , ia urmatorul client disponibil , asteapta un timp egal cu tService secunde , iar apoi decrementeaza timpul de asteptare
* Clasa Scheduler

Campurile clasei Scheduler sunt : o lista de obiecte ale clasei Server , numarul maxim de cozi , numarul maxim de clienti per coada , precum si o variabila Strategy , care ne informeaza cu privire la strategia adoptata pentru distributia clientilor in cozi.

In constructorul Scheduler se creeaza maxNoServers cozi, care mai apoi se adauga in campul servers, fiecare avand capacitatea maxClientsPerServer. Dupa crearea fiecare cozi, i se atribuie un Thread, care mai apoi este pornit.

Celelalte metode descrise sunt : changeStrategy , care initializeaza campul strategy cu o instanta a clasei ConcreteStrategyTime, si dispatchClient , care apeleaza metoda addClient a clasei anterior mentionate , cu scopul de a pune in aplicare strategia de distributie a clientilor.

* Clasa ConcreteStrategyTime

In interiorul acestei clase , suprascriem metoda addClient a interfetei Strategy. Aceasta interfata ne spune ca avem posibilitatea sa alegem logica dorita pentru a imparti fiecare client in cozile puse la dispozitie. Pentru aceasta cerinta am ales strategia de minimizare a timpului petrecut la coada , de unde si numele ConcreteStrategyTime.

Metoda addClient primeste ca parametri lista de serveri existenti si clientul ce trebuie asezat la coada. Iterand prin aceasta lista , alegem server-ul care are waitingPeriod minim , urmand ca la finalul buclei sa inseram noul client la coada corespunzatoare.

* Clasa ClientGenerator

Variabilele instanta ale ClientGenerator sunt citite dintr-un fisier de intrare , primit ca argument la executia metodei main.

Acestea sunt: timeLimit (timpul necesar pentru simulare) , tArrivalMin si tArrivalMax (dau itervalul de valori in care se incadreaza timpul de sosire al fiecarui client) , tServiceMin si tServiceMax (intervalul de timp pentru tService) , numberOfServers (numarul de cozi) , numberOfClients (numarul de clienti de generat).

Pe langa cele mentionate mai sus , unui ClientGenerator ii corespunde un Scheduler , un Enum SelectionPolicy , ce contine macro-uri care indica politica dupa care clientii vor fi impartiti , si o variabila de tip float , cu care calculam timpul mediu de asteptare la coada.

Metodele descrise :

* generateNRandomClients : creeaza , cu ajutorul unei instante a clasei Random numberOfClients clienti , cu tArrival si tService ales la intamplare din intervalul dat de tArrivalMin/Max si tServiceMin/Max ; ii stocheaza apoi in variabila generatedClients , pe care ii sorteaza dupa tArrival
* readWriteData : se ocupa cu citirea datelor din fisierul de intrare si crearea fisierului de iesire
* run ( clasa ClientGenerator contine firul de lucru principal al intregii aplicatii ) :
* creeaza o variabila currentTime , care sa tina evidenta timpului de simulare , adica de executie a Thread-ului
* itereaza lista generatedClients , iar clientul curent este trimis scheduler-ului pentru a fi repartizat unei cozi la momentul in care timpul sau de sosire este egal cu currentTime
* clientii care sunt introdusi in cozi sunt eliminati din lista
* in fisierul de iesire se printeaza treptat cozile , la fiecare moment de timp , precum si clientii care inca nu au fost distribuiti la o coada
* firul de lucru principal este oprit pentru o secunda la fiecare incrementare a currentTime, pentru a se procesa fiecare coada din scheduler
* se calculeaza media timpilor de asteptare , se scrie in fisierul de iesire , dupa care acesta se inchide
* main : creeaza un nou ClientGenerator , care are ca parametri fisierele de intrare si de iesie primite ca argumente la executie ; porneste thread-ul principal

1. **Rezultate**

Dupa cum se poate observa , programul primeste ca argument orice fisier de intrare cu date de testare a algoritmului. Dupa finalizarea aplicatiei , algoritmul implementat a fost testat cu cele trei fisiere puse la dispozitie in indrumatorul de laborator : in-test-1.txt , in-test-2.txt , in-test-3.txt. Toate cele trei fisiere de iesire rezultate au fost verificate , iar datele se conformeaza cerintei temei.

1. **Concluzii**

In urma rezolvarii acestei teme mi-am solidificat cunostintele legate de lucrul cu thread-uri , pe care le-am dobandit la finalul semestrului trecut la materia POO. De asemenea , am invatat sa structurez rezolvarea in cat mai multe clase generice care sa indeplineasca anumite obiective.

In ceea ce priveste dezvoltarea ulterioara , consider ca programul poate fi extins prin a oferi utilizatorului posibilitatea de a distribui clientii conform altor politici , nu neaparat in functie de timpul de asteptare la coada.

1. **Bibliografie**

* Pentru lucru cu thread-uri

<https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#interrupt()>

* Pentru lucru cu structuri thread safe :

<https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html>

<https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/AtomicInteger.html>

* Pentru lucru cu fisiere

<https://docs.oracle.com/javase/7/docs/api/java/io/PrintWriter.html>

* Pentru realizarea unui executabil .jar

<https://docs.oracle.com/javase/tutorial/deployment/jar/basicsindex.html>

* Pentru structurarea programului conform unui strategy pattern

<https://en.wikipedia.org/wiki/Strategy_pattern>

* Pentru intelegerea conceptului de concurenta in programare

https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html