**Queue Simulation**

**Assignment 2**

**Bilc Sergiu Ion**

**30224**

**An: II**

**Cuprins**

1. Obiectivul temei
2. Analiza problemei
3. Proiectare
4. Implementare
5. Concluzii
6. Bibliografie
7. **Obiectivul temei**

**Cerinta**: Design and implement a simulation application aiming to analyze queuing based systems for determining and minimizing clients waiting time.

**Obiective:** Se doreste crearea unei aplicatii Java, care sa realizeze simularea unui supermarket care dispune de mai multe cozi. Aplicatia ar trebui sa simuleze o serie de clienti care ajung la intrarea in cozi, decid care e cea mai buna coada in care sa intre, astept servirea clentilor dinaintea lor, mai apoi fiind si ei serviti, parasind in cele din urma magazinul.

Deasemenea, se mai doreste oferirea posibilității de a realiza operațiile de bază pe cozi: adăugare, respectiv ștergere, într-o manieră eficientă din punct de vedere al timpului. Acest aspect implică folosirea concurenței în Java: multithreading, variabile atomice, colecții pentru concurență (BlockingQueues).

1. **Analiza problemei**

Cozile sunt peste tot in jurul nostru de mai bine de 60 de ani, iar cu totii suntem nemultumiti de ideea de a sta la coada. Practic traim atat intr-o epoca a tehnologiei si a internetului cat si intr-o epoca a supermarketurilor de dimensiuni tot mai mari si a cosumismului, in care oamenii petrec un timp foarte mare pentru as putea procura obiectele si alimentele de care are nevoie. Tocmai tehnologia incearca sa rezolve aceasta problema a statului la coada prin eficientizarea comenzilor online cu livrare la domiciliu. Tot in vederea diminuarii acestei probleme vine si aceasta aplicatie, ce consta, mai exact, in simularea unui astfel de magazin si in minimizarea timpului de asteptare pentru fiecare client.

Datele de intrare date de utilizator:

* Intervalul de timp care determina diferenta de ajungere la intrarea in cozi dintre doi clienti consecutivi.
* Intervalul de timp care determina timpul de servire al fiecarui client
* Numarul de cozi
* Timpul de rulare al aplicatiei

Datele de iesire:

* Timpul mediu de asteptare pentru un client pentru fiecare coada.
* Timpul mediu de servire al clientilor pentru fiecare coada.
* Timpul in care fiecare coada este goala.
* Jurnal de evenimente
* Animatie grafica care prezinta evolutia cozilor
* “Ora de varf”, momentul in care sunt cei mai multi clienti in cozi.

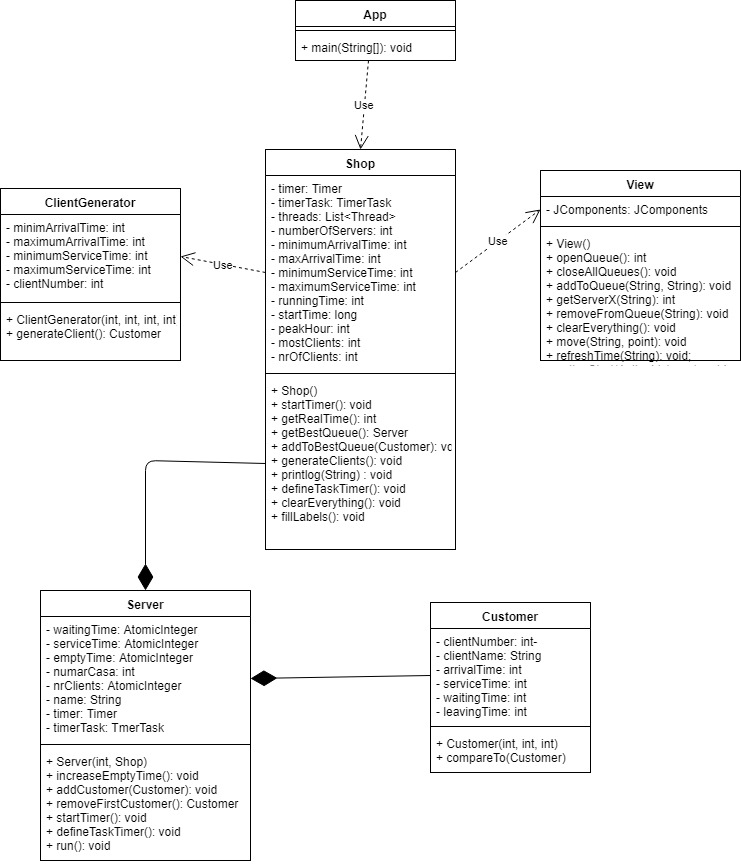
Pentru a putea implementa acest lucru avem nevoie sa cream/modelam obiectele necesare. In primul rand trebuie sa stabilim cum reprezentam clientii, astfel incat sa putem gestiona clientii in cadrul cozilor.

Un alt lucru ce trebuie gestionat este modelarea cozilor. Aceste cozilor pentru sa gestioneze clientii aflati in coada respective, si totodata sa fie idependente de celelalte cozi, astfel incat sa poata functiona simultan.

De asemenea, a fost nevoie sa stabilesc cum vor fi reprezentate grafic cozile, astfel incat sa aiba un aspect placut, dar sa fie si concludent problemei noastre.

Cea mai mare provocarea este realizarea aplicatiei astfel incat cozile sa functioneze concurent, utilizandu-ne de Thread-uri si sincronizarea acestora.

1. **Proiectare**



* 1. **Diagrama UML**
  2. **Modelare**

Am structura proiectul folosindu-ma de arhitectura MVC (Model - View - Cotroller ). Astfel, aplicatia este aranjata in 3 pachete: Models ( care reprezinta datele, obiectele, folosite de aplicatie: Customer, Server, ClientGenerator), Views(“vederile”, sectiunea grafica) si Controllers( controlorul care preia intrarile de la utilizator si le transpune in modelele cu care lucram, mai apoi ocupandu-se si de operatiile effectuate ) .

* Clasa Customer: clasa utilizata pentru modelarea clientilor, fiecare client fiind caracterizat printr-un nume ( id ) , timpul la care ajunge in coada, si timpul necesar ca coada s ail proceseze
* Clasa Server: clasa care modeleaza o coada. Aici vom gestiona un queue de Customeri, fiecarui server fiindu-I atribuit cate un thread, astfel incat aceasta sunt procesate in mod concurent.
* Clasa ClientGenerator: Clasa folosita pentru generarea random a clientilor.
* Clasa View: clasa in care am realizat interfata grafica
* Clasa Shop: aceasta clasa realizeaza gestiunea aplicaticatie, actiunea butonului de start si introducerea clientilor in cozile potrivite.
  1. **Structuri de date**

Principala structura de date folosita este Queue-ul, implmentat folosind un LinkedList . Un LinkedList este o colectie de elemente stocate ca o lista inlantuita, permitand in acelasi timp accesul rapid la date, dar o stergere si o insertie mai lenta. Aceasta clasa este continuta in spatiul System.Collections si ne permite sa construim o coada de dimensiune variabila, la care putem adauga noi oricand elemente. Cozile functioneaza pe principiu FIFO, first in, first out, astfel incat elementele pot fi adaugate doar la sfarsitul listei, iesind din coada doar elementul din capul listei. Pentru datele initiale, cum ar fi threadurile, serverele am folosit deasemenea si ArrayList-ul.

* 1. **Interfete folosite**

Pe parcursul poriectului am folosit doua interfete deja existente: Runnable si ActionListener. Cea dintai a fost folosita pentru crearea unor thread-uri pentru fiecare server in parte. Practic, fiecare Server este o clasa de tipul runnable, care se executa in mod concurent cu celelalte threaduri din interiorul aplicatiei.

1. **Implementare**

* **Clasa Customer:**

Clasa Customer reprezinta structura de date de baza folosita pentrue realizarea acestui proiect. Aceasta este o clasa simpla, care are doar 3 atribute, dintre care doua sunt de tip integer si reprezinta timpul la care un client ajunge in coada si timpul de procesare a unui client de catre o coada, iar un atribut este de tip String, si reprezinta numele clientului. Aceasta clasa reprezinta tranpunera in programarea orientate obiect a celei mai simple unitati dn care este formata o coada, si anume clientul. Pe parcursul proiectului am lucrat in repetate randuri cu obiecte de timp Customer si cu Liste sau Queue-uri de obiecte Customer.

* **Clasa Server:**

Clasa principala a proiectului, in care se stocheaza si se implementeaza cozile de client, fiecare coada fiind caracterizata de care o astfel de structura(Customers). Aceasta clasa este de un tip mai special, aceasta functie implementand interfata Runnable, fiecare Server este o clasa de tipul runnable, care se executa in mod concurent cu celelalte threaduri din interiorul aplicatiei. Atunci cand se va crea un Thread nou cu un obiect din clasa Server, metoda run() a acestei clase se va executa, in parallel cu alte fire de executie aflate in momentul curent in lucru. Operatiile importante ale aceste clase sunt cea de adaugare a unui client nou la coada curenta, operatie care se executa sincronizat pentru aceasta clasa, astfel incat doar o singura coada poate adauga un client la un moment de timp, lucru ce garanteaza siguranta Thread-ului. Cealalta operatie importanta a clasei curente eliminarea unui client din coada. Practic, coada activa in momentul respective va procesa clientul curent, (isi va oprii activitatea pana cand trece o perioada de timp egala cu timpul de servire a clientului care urmeaza sa fie eliminate din coada), iar mai apoi il va elimina.

**Metode importante:**

* *addCustomer(Customer): void* : metoda care primeste ca parametru un client, iar mai apoi il adauga pe acesta in capatul cozii curente, printand in jurnalul de activitatii faptul ca un client a intrat in coada, si procesand variabile intermediare care vor fi la finalul rularii afisate(waitingTime, serviceTime, nrClients). Aceasta metoda este sincronizata, deoarece nu se pot adauga clienti in mai multe cozi simultan.
* *getWaitingTime()*: int : metoda ce returneaza timpul de asteptare al cozii curente la momentul actual de timp. Acesta este dat de suma timpului de procesare a tuturor clientiilor aflat acum in coada.
* *getAvgWaitingTIme()*: float: metoda ce returneaza timpul mediu de asteptare pentru coada curenta.
* *getAvgServiceTime()*: float: metoda care returneaza timpulmediu de serivire a clientilor ce au trecut prin aceasta coada.
* *getEmptyTime()*: float: returneaza timpul in care aceasta coada a fost goala.
* *removeFirstCustomer()*: Customer: elimina primul element din coada in caazul in care coada nu este goala.
* *defineTaskTImer()*: void: metoda in care se defineste TimerTask-ul pentru threadul current. In cazul de face, acest TimerTask va executa eliminarea clientilor din coada, intr-un mod correct si sincronizat. Cat timp timpul current este mai mic decat timpul maxim de rulare al aplicatiei, atata timp cat coada nu este goala, se face procesarea unui client( se astepta un numar de secunde egal cu timpul de servire a clientului din varful cozii), apoi acesta este eliminat din coada, atat din punct de vedere al structurii de date, cat si din interfata grafica.
* *startTimer()*: void: metoda care porneste Timer-ul. Acest lucru consta in executarea TimerTask-ului la fiecare secunda care trece, incepand cu secunda 0.
* *Run()*: metoda run() apeleaza metoda defineTaskTimer, iar mai apoi startTimer().
* *Shutdown()*: metoda care opreste thread-ul current atunci cand aceasta metoda este apelata. Acest lucru o sa se intample atunci cand timpul current o sa depaseasca timpul final al aplicatiei.
* **Clasa Shop:**

Clasa Shop este folosita ca un Controller pentru aceasta aplicatie. Aceasta gestioneaza functionarea aplicatiei. Acest lucru consta in preluarea datelor de intrare, crearea unui numar de case egal cu numarul de case introduse de utilizator, crearea cate unui thread pentru fiecare Server, generarea clientilor utilizandu-ne de ClientGeneratorul amintit mai sus, adaugarea clientilor generati in coada de client ce urmeaza sa intre in cozile de la case, stabilirea TaskTimer-ului, pornirea thread-urilor si pornirea Timer-ului. Aceasta clasa gestioneaza generarea clientilor intr-un mod aleator, stabilirea pentru fiecare client coada cea mai buna in care acesta sa fie adaugat, in functi de timpul cel mai mic de asteptare, iar mai apoi adaugare clientului la coada respectiva. Tot in aceasta clasa este gestionat si timpul de rulare al aplicatiei. Pentru acest lucru ne-am folosit System.curretTimeMillis(), metoda care returneaza timpul current al sistemului.

**Metode importante:**

* + - *getRealTime(): long:* metoda care returneaza timpul curent al aplicatiei, raportat la timpul initial, adica timpul in care aplicatia a inceput sa ruleze. Practic, am scazut din timpul curent al sistemului, timpul initial.
    - *getBestQueue():* Server: aceasta metoda stabileste care este coada cea mai buna pentru clientul urmator, in functie de timpul minim de asteptare pentru fiecare coada. Tot in aceasta metoda am gestionat variabile ce reprezinta datele de iesire a aplicatiei( peekHour, timpul in care coada a fost goala, etc).
    - *addToBestQueue(Customer):* void: metoda care adauga la coada cea mai buna, clientul curent. Tot in aceasta metoda sunt apelatesi metodele de adaugare a clientului la interfata grafica.
    - *generateClients():* void: metoda care genereza client random, utilizand obiectul de tip ClientGenerator, acesti client fiind stocati intr-o lista de Customeri.
    - *printLog(String):* void: metoda care adauga String-ul transmis ca parametru la JTextArea-ul folosit pe post de jurnal de activitati.
    - *defineTaskTImer()*: void: metoda in care se defineste TimerTask-ul pentru threadul current. In cazul de fata, acest TimerTask va executa adaugarea clientilor in coada cea mai buna, intr-un mod correct si sincronizat. Cat timp timpul current este mai mic decat timpul maxim de rulare al aplicatiei, si atata timp cat coada nu este goala, se face procesarea unui client( se astepta un numar de secunde egal cu timpul de ajungere a acestuia la intarea in cozi), apoi acesta este aduagat in coada cea mai buna, atat din punct de vedere al structurii de date, cat si din interfata grafica. Atunci cand timpul curent o sa fie mai mare decat timpul final de rulare a aplicatiei, threadurile o sa fie oprite, cozile o sa fie inchise, iar datele de iesire vor fi afisate in interfata.
    - *startTimer():*void: metoda care porneste Timer-ul. Acest lucru consta in executarea TimerTask-ului la fiecare secunda care trece, incepand cu secunda 0.
    - *clearEverything(): void:* metoda care este folosita la refolosirea aplicatiei in cazul in care se doreste rerularea aplcaitiei dupa ce aceasta isi termina activitatea. Aceasta metoda reinitializeaza toate structurile de date folosite, si curata toate componentele din interfata grafica.
    - *fillLabels():* void: metoda folosita la adaugarea la interfata grafica a datelor de iesire pentru fiecare coada(ex: EmptyTime,AvgServiceTime, AvgWaitingTime).
* **Clasa View:**

Aceasta clasa este interfata grafica a aplicatiei, folosita pentru a reprezenta grafic evolutia cozilor.

**Metode importante:**

* *addToQueue(String, String): void:* metoda folosita la adaugarea client la coada potrivita din punct de vedere grafic. Acest lucru consta in adaugarea unui JLabel cu numele clientului pe care dorim sa il adaugam, la Server-ul cu numele pe care il dam ca parametru.
* *getServerX(String):* int: Returneaza pozitia pe axa X a serverului dat ca parametru.
* *removeFromQueue(String):* Stergerea din varful cozii date ca parametru a clientului din varful cozii.
* *Move(String, Point):* void: metoda folosita la realizarea mutarii unui client din coada de client care asteapta sa intre in cozile caselor, la pozitia exacta din coada potrivita in care urmeaza sa fie adaugat. Pozitia in care acesta trebuie sa ajunga este calculate in afara metodei, in functie de coada in care clientul urmeaza sa fie introdus si de gradul de umplere al cozii respective.

1. **Concluzii**

Această aplicație are rolul de a simula, cât ai verosimil posibil, un episod din cadrul unei instituții în interiorul căreia lumea stă la coadă, cu timp de așteptare minim. Desigur, în lumea reală nu se cunoaște cu exactitate timpul de procesare al fiecărui client.

Această temă m-a ajutat să înțeleg mai bine conceptul de concurență în Java: thread, tultithreading, suspendarea thread-urilor, sincronizarea metodelor și folosirea variabilelor atomice.

O dezvoltare ulterioară a aplicației ar presupune extinderea numărului de cozi, nu un număr foarte mare totuși, interfața nu ne-ar permite, și introducerea în interfață a unui ComboBox prin intermediul căruia să putem alege strategia de alegerea a cozii pentru fiecare client.

1. **Bibliografie**

[1] <https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html>

[2] [http://coned.utcluj.ro/~salomie/PT\_Lic/4\_Lab/](http://coned.utcluj.ro/~salomie/PT_Lic/4_Lab/%20)

[3] <http://www.coned.utcluj.ro/~marcel99/PT/Tema%202/Java%20Concurrency.pdf>