**DOCUMENTATIE TEMA 2**

**QUEUES SIMULATOR**

**Csillag Szabolcs Andras**

**Grupa 30227**

**Profesor: Ana-Maria Nanes**

Cuprins:

1. Cerintele temei …………………………………………………………………………......3

2. Obiective ……………………………………………………………………………………. 3

2.1 Obiectiv principal ………………………………………………………………………...………….........3

2.2 Obiective secundare ………………………………………………………………………………….......3

3. Analiza problemei ………………………………………… …………………… ……….. 4

3.1 Use case-uri / scenarii …………………………………………………… ……………… ……………4

4.Proiectare ………………………………………………………… …………………… …………....4

5.Implementare ………………………………………………………… ……………… … ………....6

6.Testare ………………………………………………………… ……………………… …… …....8

7.Dezvoltari ulterioare si Concluzii…………………………………………………………… ……....9

8.Bibliografie ………………………………………………………… ………………………… ……....9

# Cerintele Temei

( cerintele problemei )

Implementati o aplicatie care simuleaza analizarea unui sistem de cozi ( queues) pentru a determina si minimiza clientilor perioada de asteptare (waiting time) . Cozile sunt destul de comune si sunt implementate in viata de zi cu zi, in domeniile din viata reala. Aplicatia trebuie sa genereze random n clienti si sa ii distribuie treptat in coada ideala cu timpul de asteptare minim.El trebuie sa intre la un moment tarrival si sa stea in fata cozii un anumit tservice.

Toti clientii ar trebui sa fie generati random cand simularea incepe si sunt caracterizati de 4 parametrii(id :care e cuprins intre 1 si N , arrivalTask:timpul de simulare cand sunt pregatiti sa intre in una dintre cozi, serviceTask: intervalul de timp necesar pentru servirea clientului respectiv de casier, waitingPer: perioada de asteptare din momentul intrarii in cozi si pana la momentul plecarii clientului)

Aplicatia urmareste timpul total petrecut de fiecare client / customer / task in cozi si calculeaza timpul mediu de asteptare.

Fiecare client este adaugat la coada cu minimul de waitingTime, cand arrivalTask este egal sau mai mare ca tsimulation (tarrival>=tsimulation)

# Obiective

## Obiectiv Principal:

(Obiectivul principal pe care il urmarim in acest proiect)

Obiectivul principal unei cozi este de a produce loc unui “client” sa astepte pana primeste un “service”.Managementul de cozi este bazat pe sistemul de minimizare a timpului in care clientii sai asteapta in fata cozii ,pana sunt serviti . (ex: cozile din magazinele din Lidl, atunci cand nu er covid-19 prezent ) . O metoda eficienta pentru a face acest lucru de minimizare a perioadei de asteptare este de a adauga mai multe servere, mai multe cozi in sistem( fiecare coada e considerat ca are un anumit procesr,adica thread) ,dar aceasta metoda creste costul de “service suplier”. Cand un server nou este adaugat , clientii care sunt in asteptare vor fi distribuiti pe rand in cozile existente .

Pe langa cele descrise la cerintele temei , trebuie sa folosim multithread-uti pentru implementare cozilor.

## Obective Secundare:

( pasii necesari pentru a atinge obiectivul principal )

|  |  |  |
| --- | --- | --- |
| **Obiectiv Secundar** | **Descriere** | **Capitol** |
| Generarea randm a n clienti | Aplicatia simuleaza o serie de n clienti “numberOfClients” care sosesc la un anumit timp in cozi “arrivalTask” si stau in fata unei cozi “serviceTask” ,si dupa servirea lor =>parasesc coada | 4 |
| Alegerea structurilor de date | Structurile de date folosite pentru a duce la capat obiectivul principal | 4 |
| Impartirea pe clase | Folosirea unui Scheduler ,care oerecum aste indica cum se executa aplicatia.Folosirea unui SimulationManager ,Server,Task etc. | 6 |
| Dezvoltarea algoritmilor | Vor fi descrise structurile de date necesare pentru atingerea obiectivului principal, schema UML ( diagrama de clase ) precum si algoritmii folositi pentru realizarea operatiilor. | 6 |
| Implementarea solutiei | Vor fi descrise fiecare clasa cu campurile si metodele importante precum si descrierea interfetei utilizator; | 6 |
| Testare | Voi testa cele 3 exemple manual “in-test-1.txt” “in-test-2.txt” “in-test-3.txt” ,fiecare cu iesirile sale : “out-test-1.txt” “out-test-2.txt” “out-test-3.txt”. | 8 |

# Analiza Problemei

( use case-uri / scenarii care au o probabilitate mare sa apara )

**3.1 Use case-uri / Scenarii**

Utilizarea acestui program se face prin introducerea in primul fisierul text selectat ,in aceasta ordine urmatorii operanzi , toti fiind valori integer

1. Numarul de Clienti
2. Numarul de cozi (queues)
3. Numarul maxim de simulare (timpul maxim pana cand se va simula programul)
4. minArrivalTime urmat de maxArrivalTime (range-ul pentru care se va genera random arrivalTime pentru fiecare client/Task)
5. minServiceTime urmat de maxServiceTime ( range-ul pentru care se va genera random serviceTime pentru fiecare client/Task)

Al doilea fisier text ,trebuie sa aiba un nume,daca acesta nu exista se va genera automat cu rezultatele in interiorul sau.

Programul se ruleaza si astfel rezultatul se va afla in al doilea fisier, se va afisa in fiecare timp clientii care inca asteapta si fiecare coada cu clientii sai .

(Pentru logica programului folosesc : Client este egal cu un Task)

# Proiectare

Aici voi descrie etapele de proiectare folosite pentru realizarea acestui proiect ,dar mai ales gandirea constructiei ,si metodele alese.

## Structuri de date

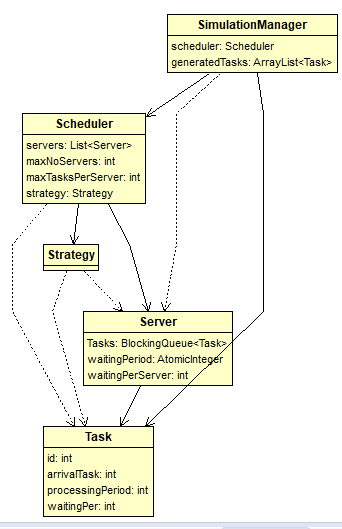
(Ce structuri de date am folosit ?)

Ca si structura de data principala ce am folosit-o este un ArrayList de Servere(deoarece un sistem de cozi este format din mai multe servere ,adica un server este oarecum o coada, un arraylist ar fi fost cel mai potrivit in rezolvarea acestei probleme,fara sa ma complic cu un HashSet sau TreeSet , ArrayList este cea mai usoara si mai eficienta varianta in momentul actual ) .

Un server este format :

* dintr-o coada blocata de tipul Task , un anumit Task are id,arrivalTask,processingPeriod si waitingPer
* un waitingPeriod de tipul AtomicNumber ,care este oarecum o variabila volatila ,modificarea ei este salvata in toate threadurile!

## Diagrama de clase

Diagrama de clase,adica schema UML a proiectului am generat-o in ecplise si apareau prea multe detalii. Dar ca sa incapa frumos pe pagina ,am decupat un pic de jos.

ConcreteStrategyQueue si ConcreteStrategyTime implementeaza interfata Strategy

SelectionPolicy este doar o enumeratie cu doua variabile.

Serverul contine un BlockingQueue de tipul Task,cozile care vor fi implementate

Scheduler contine o lista de servere si o strategie

SimulationManager contine un scheduler si o lista de Task-uri ,lista de asteptare

## Algoritmi

* In acest proiect nu am folosit un algoritm anume ,cum ar fi Greed sau Dijkstra , am folosit doar logica matematica folosind o mai multe cozi ,si in fiecare se adaug clientii care se afla in coada mare de generatedTasks.
* Sunt prezentate in clasele detaliate din sectiunea urmatoare

# Implementare

**5.1 Clasa ConcreteStrategyQueue :**

-implementeaza interfata Strategy si are o singura metoda suprascrisa care ar trebui sa adauge un Task la coada unde sunt mai putini clienti, nu am implementat-o pentru ca nu o folosesc in acest assignment.

**5.2 Clasa ConcreteStrategyTime:**

-implementeaza interfata Strategy

-are o singura metoda care o ruprascrie : addTask(servers, t) ,aceasta adauga taskul t la serverul cu waitingPeriod cea mai mica.

**5.3 Interfata Strategy**

-are doar o metoda neimplementata ,care urmeaza sa fie suprascrisa de descendentii sai : addTask(servers,t);

**5.4 Enum SelectionPolicy**

-are doar doua variabile: SHORTEST\_TIME , SHORTEST\_QUEUE

**5.5 Clasa Task**

-variabile: id( id-ul unic al clientului) ,arrivalTask(momentul in care taskul intra in una din servere ), processingPeriod ( timpul necesar cat acest task va sta in fata cozii ) ,waitingPer ( diferenta dintre timpul in care taskul intra in cozi si acesta pleaca ) ,toate sunt de tipul int ;

-un constructor cu parametrii (id ,arrival\_min, arrival\_max , service\_min ,service\_max ) unde waitingPer e initializat cu 0 ,id este preluat ,arrivalTask va fi egal cu un nr random intre cele doua valori arrival\_min si arrival\_max , processingPeriod tot asa un nr random intre service\_min si service\_max ;

-are metode :gettere in afara de id,pt toti parametrii , toString ,incrementeazaPer() care creste waitingPer ,decrementeazaProces() si pentru ca aceasta clasa implementeaza clasa Comparable are si metoda comapreTo)

**5.6 Class Server**

-are o coada de tipul Blocking ,care va contine mai multe Task-uri

-are un AtomicInteger waitingPeriod ,care indica cat mai e de asteptat pe serverul respectiv

-are un int : waitingPerServer , cat s-a asteptat pe server

-constructorul ,fara parametrii ,initializeaza variabilele instanta

-metoda addTask ,care adauga un task la serveru curent,si se adauga la waitingPeriod, processPeriod din taskul respectiv primit ca argument

-metoda run suprascrisa ,pentru ca clasa implementeaza Runnable :

-metoda toString ,getWaitingPeriod, getWaitingPerServer, incrementeazaPer ( )

**5.7 Class Sceduler**

-are o lista de servere: servers , maxNoServers,maxTasksPerServer (astea 2 de tip int ) si are o anumita strategie ,adica variabila strategy de tip Strategy

-Constructorul primeste peremterii doi integeri pentru a initializa cele doua var instanta declarate , alocam spatiu pentru servers cu ajutorul maxNoServers si apoi pentru fiecare maxNoServers facem un server nou si il adaugam in servers

-metoda changeStrategy ,folosesc strategy pentru a efectua o schimbare de strategie care sa puna de ex un client in coada mai scurta , nu unde este timpul mai scurt,dar aceasta metoda nu se foloseste in programul actual

-un getter pentru servers , si disPatchTask(t) care va pune in unul dintre servere taskul t in functie de strategia actuala

**5.8 Class SimulationManager**

-implementeaza si ea Runnable ,asemanator ca Server

-este clasa cea mai complexa ,care leaga tot ,are toate variabilele instanta care ar trebui primite din fisierul de intrare : numberOfClients , numberOfServers , timeLimit, minArrivalTime , maxArrivalTime , minServiceTime , maxServiceTime ; pe langa astea are si averageWaitingTime, care va fi rezultatul timpului mediu care se asteapta la cozi .

-are un selectionPolicy,un scheduler,o lista de Task-uri : generatedTasks,vectori de thread-uri,String afis :la care se vor concatena raspunsurile scrise in fiseriul out si PrintStream gchar :cu ajutorul care scriem in fisier

-Constructorul primeste 2 parametrii :stringul cu fisierul de intrare si stringul cu fiserul de iesire ,aceasta citeste datele si pregateste scrierea

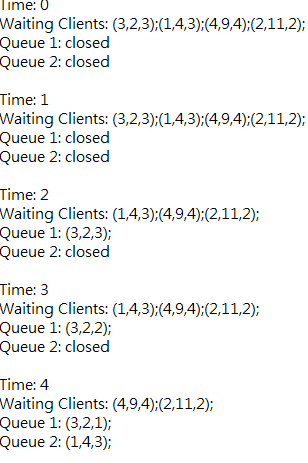
-metoda generateNRandomTasks face ceea ce credem genereaza cele N task-uri si mai ales ii sorteaza,pentru a fi mai usor de urmarit arrivalTask

-apare metoda main ,caredoar construieste un SimulationManager,face un thread pe seama lui si il porneste

-metoda run ,este cea mai complexa metoda , ea incepe cu un currentTime=1 si parcurge un while cat timp acesta e mai mic ca timeLimit-ul citit din fisier , din while mai iesim astfel sau avem un break daca nu mai sunt nici clienti in asteptare si nici in cozile implementate , luam primul Task din lista de asteptare si daca acesta are arrivalTime egal cu currentTime ,atunci il adaugam in una din servere si desigur il scoatem din lista de asteptare . Construim pas cu pas iesirea necesara in fisierul out si asteptam 1 sec.

# Testare

Pentru verificarea acestui miniprogram / assignment vom folosi cele 3 teste : “in-test-1.txt” “in-test-2.txt” “in-test-3.txt” ,fiecare cu iesirile sale : “out-test-1.txt” “out-test-2.txt” “out-test-3.txt”. Ultimele doua sunt mai complicate cu nr de client si cozi mai multe ,asa ca vom discuta primul exemplu , cu n = 4 , q = 2 ,tsim = 60 , minArrival = 2 ,maxArrival = 30 , minService = 2 ,maxService = 4 ; Se vor genera 4 clienti random ,cu arrivalTask si processingPeriod cuprinse intre rangeurile sale .Pornim simularea si obtinem un rezultat asemanator de fiecare data , rezultatele in sine vor diferi de fiecare data din cauza ca folosim random :



# Concluzii si Dezvoltari Ulterioare

Acest program este foarte util in viata reala , de exemplu la un supermarket cum ar fi Lidl clientii ar intra in perioada de asteptare cand intra in magazin, cand isi fac cumparaturile si vor sa plateasca la casa de obicei sunt mai multe cozi la case ,acestia se uita unde sunt produse mai putine pe banda si acolo intra ,nu neaparat unde sunt mai putini client, astfel pusi intr-o coada se incrementeaza waitingPeriod sic and ajung la caserie ,mai au nevoie de un timp pana se scaneaza produsele,pana plateste si apoi iese din coada ,asta se intampla folosirea variabilei processingPeriod . Astfel fara sa ne dam seama folosim un system de cozi generate random in fiecare zi . Totusi este optim acest algoritm.

Observatie : Treadurile nu sunt deschise/inchise cum arc ere exact cerinta , eu am le-am deschis la inceput si incheiat la final ,merge bine programul ,nu am stat sa ma mai complic cu rezolvarea aceste dileme.Referitor la tema anterioara nu am mai pus copy-paste din cod in documentatie ,diagrama UML am incercat sa o fac mai vizibila si pentru ca nu aveam gettere si settere la tema1 a fost doar o neatentie ,probabil trebuia sa fie 10 ,dar la asta nu simt ca am muncit de 10,pentru ca nu am inteles exact daca ar trebui sa deschid/inchid intre timp threadurile,de ce nu numai la inceput si sfarsit? Adica und ear fi eficienta mai buna…

# Bibliografie

M-am folosit un pic din documentatia temei anterioare pentru a oferi un cuprins asemanator . Pentru cod am fost inspirit din pdf-ul primit de professor pe site : java concurrency . Scheletul e acela ,eu am modificat codul essential pentru a personaliza.

Pentru lucrul cu fisiere m-am orientat din linkul: https://www.dopopan.ro/liceu/F3.pdf

Acel java concurrency mentionat: <http://www.coned.utcluj.ro/~salomie/PT_Lic/4_Lab/Assignment_2/Java_Concurrency.pdf>