**DOCUMENTATIE TEMA 2**

**SIMULARE CASE SUPERMARKET**

**Petra Adrian Mariel**

**30225**

**Indrumator Laborator Moldovan Dorin**

Contents

[1. Cerinte Functionale 3](#_Toc476131445)

[2. Obiective 3](#_Toc476131446)

[2.1. Obiectiv Principal: 3](#_Toc476131447)

[2.2. Obective Secundare: 3](#_Toc476131448)

[3. Analiza Problemei 3](#_Toc476131449)

[4. Proiectare 3](#_Toc476131450)

[4.1. Structuri de date 3](#_Toc476131451)

[4.2. Diagrama de clase 3](#_Toc476131452)

[4.3. Algoritmi 3](#_Toc476131453)

[5. Implementare 4](#_Toc476131454)

[6. Testare 4](#_Toc476131455)

[7. Concluzii si Dezvoltari Ulterioare 4](#_Toc476131456)

[8. Bibliografie 4](#_Toc476131457)

# Cerinte Functionale

Proiectati si implementati o aplicatie de tip simulare care sa analizeze un sistem de tip FIFO( first in first out ), adica de tip coada, pentru determinarea si minimizarea timpului de asteptare al clientiilor.

# Obiective

## Obiectiv Principal:

Scopul principal consta in realizarea unei aplicatii in Java care realizeaza o simulare a unei cozi de supermarkeet. Formarea acesteia presupune respectarea paradigmei de programare orientata pe obiect si principiile pe care aceasta le pune la dispozitie . Pe langa interactiunea dintre clase , obiecte si metode , aplicatia presupune cunostinte pentru manipularea obiectelor clasei Thread sau interfetei Runnable , alegerea unor structuri de date adecvate pentru reprezentarea cozilor si organizarea intr-o interfata prietenoasa pentru a putea fi usor urmarita de catre utilizator.

## Obective Secundare:

|  |  |  |
| --- | --- | --- |
| **Obiectiv Secundar** | **Descriere** | **Capitol** |
| Dezvoltarea de use case-uri si scenarii | Dezvoltarea unei liste de comportare pe care ne asteptam ca programul sa o aplice | 3 |
| Alegerea structurilor de date | Se va prezenta proiectarea OOP a aplicatiei, diagramele UML de clase si de pachete, structurile de date folosite si interfetele definite. | 4 |
| Impartirea pe clase | Alegerea potrivita a claselor cat si motivarea acestora | 4 |
| Dezvoltarea algoritmilor | Descrierea si motivarea alegerii algoritmilor potriviti | 4 |
| Implementarea solutiei | Gandirea celei mai optime strategii de rezolvare a problemei propuse | 5 |

# Analiza Problemei

Un use case este o metodologie utilizată în analiza sistemului pentru a identifica , clarifica și organiza cerințele sistemului . Un use case este alcătuit dintr-un set de posibile secvențe de interacțiuni între sisteme și utilizatori într-un anumit mediu și legate de un anumit scop . Se compune dintr-un grup de elemente (de exemplu clase și interfețe) care pot fi utilizate împreună într-un mod care va avea un efect mai mare decât suma elementelor separate combinate . Use case-ul ar trebui să conțină toate activitățile sistemului care au o semnificație pentru utilizatori. Un use case poate fi considerat ca o colecție de scenarii posibile legate de un anumit scop , într-adevăr , use case și scopul sunt uneori considerate a fi sinonime .

Activarea:   
 Rularea Programului

Preconditii:  
 Schimbarea parametriilor din RandomClientGenerator pentru serviceTime si newClientWaitTime cat si alegerea numarul de case deschise.

Post-conditii :  
 La finalul rularii programului avem un jurnal de activitate in consola.

Decursul normal :  
 1 . Dupa rularea programului se va deschide o fereastra care va indica timpul curent al simularii (intre 0 si 599) si cel mai nou client, cat si fiecare casa de marcat cu clientul pe care il serveste la momentul X al timpului curent.

2. Ferestra de gui isi va da refresh de fiecare cand se incrementeaza timpul curent cu 1 (cu setarile de acum , in fiecare secunda ), unde isi va schimba textul dupa cum urmeaza:

2B. Timpul curent tot timpul va creste cu 1.

2C. Cel mai nou client creat se va schimba doar daca a trecut timpul de la newClientWaitTime.

2D. Casa de marcat isi va schimba textul doar daca a trecut timpul serviceTime al clientului curent, daca a trecut clientul va iesi din coada si va merge urmatorul client in capul cozii ,daca coada este goala atunci vom avam o instiintare de tipul „ NO CUSTOMER ”. De asemeanea alaturi de fiecare casa avem o celula unde ne este specificat clientul actual cu id-ul sau si serviceTime-ul.

3. La fiecare tic de ceas (currentTime ) consola va afisa urmatoarele : currentTime sub forma -----X----- , urmat de crearea unui nou client daca este creat, afisat sub forma „ I've made the Client: Client X with Y serviceTime” apoi hotaram in ce coada il punem si afisam in consola „X in coada Z” unde Z € [1,5] ,dupa care afisam noul wait time pana la urmatorul client „ new wait time: T” si daca un client a iesit din coada se va afisa „I've removed Client X from coada Z”. Iar dupa ce toate ticurile de ceas sunt terminate ( currentTime = 600 ) apare „The Store is closed”.

## Structuri de date

Pentru implementarea aplicatiei am utilizat structura de Model-View-Controller intru - cat ofera accesibilitate atat pentru utilizator , cat si pentru implementarea proiectului . Clasele sunt mai bine organizate , iar interactiunile intre obiecte sunt mai evidente . Modelul contine clasele Client , RandomClientGenerator, care reprezinta tot ce tine de gestionarea cozilor si crearea de noi clienti cu valori random . Inferfata grafica este reprezentata prin DigitalWatch , ea constituie principala vedere a utilizatorului . Intre Model si View exista o legatura stabilita prin Controller, clasa Main . Acesta pe baza actiunilor utilizatorului preia datele introduse in Model si le transmite mai departe catre View pentru a fi simulate . Ca si structuri de date am folosit : In client int pentru timpul de servire(serviceTime) si String pentru id. In ClientGenerator am folosit o structura de tip DigitalWatch pentu gui , o structura de tipul Queue<Client> pentru cozi , int pentru time, serviceTime, newClientWaitTime, waitTime, currentTime, passedTimeX, idnum, coadaXwaitTime, coadaXstartTime si un Thread pentru intarzierea sistemului. Pentru gui am folosit un layout de tip grid de marime 4X4 unde am asezat 16 Jbuttons.

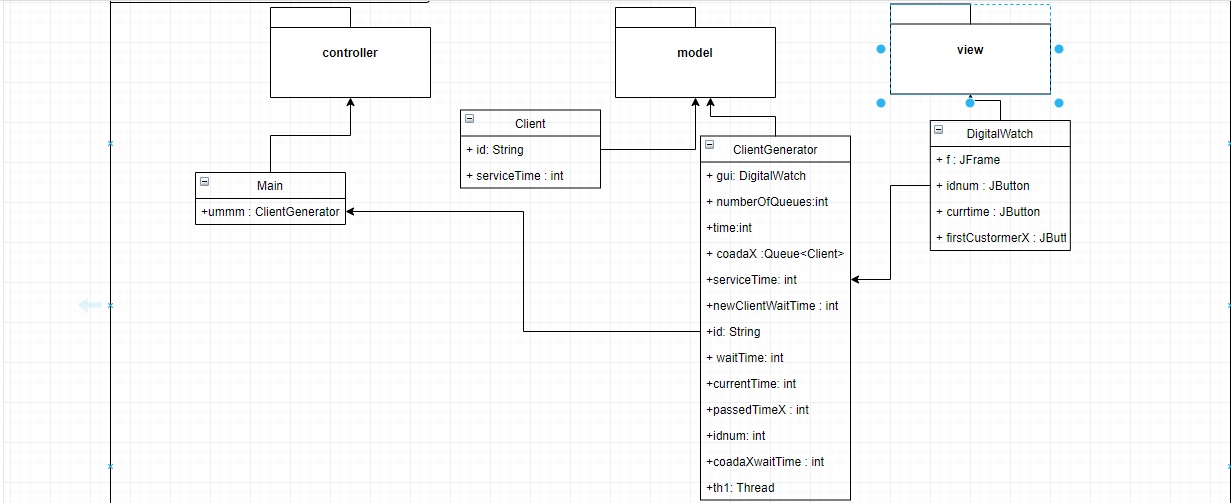
**Proiectare clase**

Aplicatia mea este alcatuita din 3 pachete denumite sugestiv dupa modul de implementare ales ( Model-View-Controller ) fiecare continand clasele utilizate in tot procesul de calcul.

**Relatii**

In cadrul claselor implementate se observa usor relatiile de dependenta pe diagrama UML prezentata mai sus, cele mai evidente sunt dependentele intre clasele , Client , ClientGenerator ,dar si ClientGenerator si DigitalWatch .

## Diagrama de clase

Din aceasta diagrama puteam vedea ca package ul aplicatie contine clasele Monom , Model , si Polinom ( mosteneste Monom ), este importata in clasa Controller . Package ul controller contine clasele ascultator si controller , unde Controller foloseste ascultator . Si package ul gui care contine clae Menu ; gui este importat de asemenea in clasa controller . 

# Implementare

Clasa Client

Are doua campuri unul de intreg si unul de String. Reprezinta cea mai mica subunitate din proiect.

Constructorul clasei primeste doi parametri, un intreg si un String, care vor reprezenta atributele obiectului in ordinea prezentata mai sus .

Avem getters pentru Id cat si pentru serviceTime , aceste atribute fiind private .

Ca si metode avem o singura metoda care returneaza 1 sau 0 ,depinzand daca obiectul este gol sau nu .

Clasa ClientGenerator

Ca si atribute are 5 private Queue < Client >, un private DigitalWatch , un private Thread , un private String si 22 de intregi si implementeaza interfata Runnable.

Metoda run() reprezinta principala functie si in ea am un timer care merge pe un interval de la 0 la 599, intr-o bucla for unde apelam metodele removeClient(), addNewClient, si reimprospatam interfata grafica, mai avem si un bloc try catch pentru din cauza apelarii functiei de sleep a thread-ului care ne opreste programul pentru X milisecunde. Dupa iesirea din bucla se va printa in consola „The Store is closed”.

Metoda addNewClient() reprezinta adaugarea unui nou client in una din cozi, doar in momemntul in care a trecut timpul de newClientWaitTime , avem un switch pentru fiecare din cozi , unde variabila de alegere este data de functia getMinimumServiceTime. Pentru a adauga un nou Client in coada ne folosim de functia add a tipului de variabila Queue<> , creez un nou timp de asteptare pentru urmatorul client, dupa ce ies din switch creez un nou serviceTime pentru clientul urmator.

Metoda removeClient() reprezinta scoaterea din coada a unui clientului din varf, dupa inplinirea timpului de asteptare, tinand cont ca daca avem mai multe cozi putem avea mai multi clienti care sa iasa din cozi ,trebuie sa facem o verificare pentru fiecare coada in parte. In cazul in care nu exita coadaX sau nu are clienti in momentul curent am conditia ca si coadaX.isEmpty() sa fie 0, doar daca aceasta conditie este implinita vom merge si vom vedea daca timpul pe care primul client la petrecut la casa de marcat este acealasi cu timpul alocat (serviceTime) ,daca este adevarat atunci il vom scoate din coada si vom anunta in consola .

Metoda getMinimumServiceTime() este o metoda de verificare a celui mai mic timp de asteptare dintre toate cozile disponibile, folosita pentru a putea vedea unde punem urmatorul client, doarece numarul de cozi poate sa difere vom avea nevoie de diferite configuratii ale if-urilor .

Clasa Main

Este folosita pentru a putea incepe programul, prin creearea unui obiect de tipul ClientGenerator si prin apelarea functiei run() a acestuia .

Clasa DigitalWatch

Este folosita pentru interfata grafica. Acesta contine un frame de tipul JFrame care are un layout de tipul grid 4X4 adica 16 celule, pe care eu le-am facut butoane de tipul JButton primul buton reprezinta timpul curent al aplicatiei , cel de-al doilea buton reprezinta numarul de clienti creati ( cat si Id-ul celui mai nou client ) , iar celelate 10 butoane sunt impartite pentru fiecare casa de marcat in parte astfel avand 2 butoane pe casa. Primul buton reprezinta Clientul care este servit in momentul x al aplicatiei si este redat prin id-ul acestuia cat si serviceTime-ul acestuia , celalalt buton reprezinta numarul de clienti aflati in coada de asteptare .

Avem o singura metoda si anume refresh ( int id , int currTime , Queue<Client> first , Queue<Client> second , Queue<Client> third , Queue<Client> fourth , Queue<Client> fifth) ,care primeste id-ul celui mai nou client , timpul curent si toate cozile . Daca avem vreo coada goala vom afisa pe butonul respectivei cozi „NO CUSTOMER”, daca avem ceva in coada vom afisa pe butoanele fiecarei cozii clientul din varful cozii cu .peek().getId() ( peek() returneaza elementul din varful cozii ,si eu avand un element de tip Client am nevoie de getId() pentru a face rost de id-ul sau ) si vom afisa numarul de clienti in coada cu .size() (care returneaza numarul de elemente dintr-o structura).

# Concluzii si Dezvoltari Ulterioare

O aplicatie care ne simuleaza demersul caselor de marcat dintr-un supermaket pare a fi o aplicatie interesanta pentru a putea analiza cand este nevoie sa se deschida o alta casa . Ca si dezvoltari ulterioare s-ar ideal ar fi ca datele precum numarul de case de marcat , timpul maxim de serviceTime si timpul maxim de newClientWaitTime sa fie introduse prin interfata grafica ,direct de catre utilizator . O alta idee de dezvoltare ar putea fi deschiderea automata de case de marcat dupa cum urmeaza : Sa fie introdus un numar maxim de de asteptare ,iar daca o casa ajunge la acel numar atunci se va deschide o alta casa de marcat dupa un timp specific , in timpul ala specific mai vin oameni la prima casa de marcat , dupa terminarea timpului specific clientii de la prima casa de marcat se vor muta la cea proaspat deschisa, nu toti doar cei care sunt dupa limita timpului de asteptare , si nu neaparat in ordine, iar daca dupa terminarea timpul specific casa 1 a ajuns sub limita de asteptare , casa proastpat deschisa va mai sta un timp x pentru asteptarea clientiilor . Daca cele doua case de marcat nu fac impreuna peste limita de asteptare a unei case de marcat ,atunci una din ele se va inchide .

O alta imbunatatire ar putea fi introdusa la interfata grafica , unde chiar ar putea fi puse niste case de marcat( figuri geometrice ex : patrat ) si clienti ( ex : puncte ) pe care sa fie scris id-ul lor .

# Bibliografie

* 1. <https://www.tutorialspoint.com/java/java_multithreading.html>
  2. <http://tutorials.jenkov.com/java-concurrency/creating-and-starting-threads.html>
  3. <https://docs.oracle.com/javase/tutorial/essential/concurrency/runthread.html>
  4. <https://softwareengineering.stackexchange.com/questions/312177/queue-vs-threads>
  5. <https://stackoverflow.com/questions/2332537/producer-consumer-threads-using-a-queue>
  6. <https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html>
  7. <https://www.tutorialspoint.com/java/java_multithreading.html>
  8. <https://www.w3schools.com/java/java_switch.asp>
  9. <https://stackoverflow.com/questions/14463289/check-object-empty>
  10. <https://docs.oracle.com/javase/8/docs/api/java/awt/GridLayout.html>
  11. <https://docs.oracle.com/javase/7/docs/api/javax/swing/JFrame.html>
  12. <https://www.javatpoint.com/java-jbutton>
  13. <https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html>
  14. <http://www.javaprogrammingforums.com/whats-wrong-my-code/12283-checkout-line-simulation.html>
  15. <http://www.ntu.edu.sg/home/ehchua/programming/java/j4a_gui.html>
  16. <https://codereview.stackexchange.com/questions/83190/simple-gui-animation-in-java>
  17. <https://stackoverflow.com/questions/5887709/getting-random-numbers-in-java>
  18. https://www.oreilly.com/library/view/java-threads-second/1565924185/ch04s04.html