**Capitolul 1.Obiectivul temei**

Obiectivul principal al lucrarii este calculul timpului de astepare mediu pe care un client il petrece asteptand intr-o coada (sistem cu principiu de coada) pana cand ajunge sa fie servit (procesat), selectarea cozii prin mecanismul firelor de executie fiind tot un subiect dezvoltat in aceasta lucrare.

**1.1.Obiectivele secundare sunt adresate in urmatoarele capitole :**

**Cap.2 : Analiza problemei ,modelare ,scenarii ,cazuri de utilizare**

Cadrul de cerinte functionale e realizat de culculul timpilor de asteptare,servire si a celor in care cozile sunt goale, la nivelul fiecarei cozi sau global (media pe cozi) . Se vor prezenta descrieri de use-case impreuna cu lista pasilor de executie a fiecarui use-case.

**Cap.3** : **Proiectarea diagramelor UML , a algoritmilor si intefetelor utilizator**

Se vor prezenta decizii legate de proiectarea claselor , structurilor de date, interfetelor utilizate in descrierea problemei , precum si a relatiile dintre acestea , organizarea in pachete precum si diagramele de clase si pachete utile in modelarea OOP a acestor sisteme bazate pe cozi.

**Cap.4 : Implementarea propriu-zisa**

Prezentarea campurilor , constructorilor , metodelor importante care realizeaza corespondenta OOP-sisteme bazate pe cozi .Tot aici se va prezenta interfata cu utilizatorul si cateva exemple de utilizare.

**Nota:**

Intrucat firele de executie nu au un comportament determinist rezultatele jUnit ar putea diferi pentru aceleasi date de intrare motiv pentru care nu vor fi discutate in aceasta lucrare.

**Cap5. : Concluzii**

In acest capitol , se discuta concluzii , precum si dezvoltari posibile ulterioare.

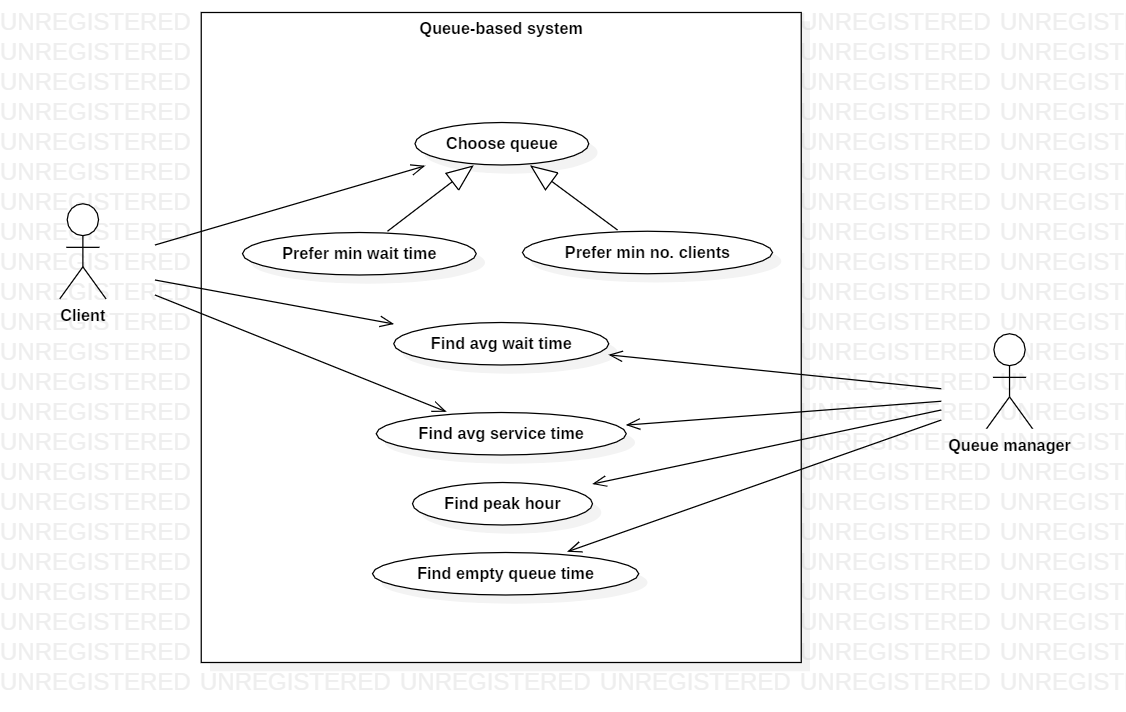
**Cap6. : Bibliografie**

**Capitolul 2: Analiza problemei, modelare, scenarii, cazuri de utilizare**

Se prezinta in continuare functionalitatile oferite de aplicatie sub forma unei diagrame ce prezinta principalele cazuri de utilizare ale unui sistem bazat pe cozi .

Diagrama contine 7 cazuri de utilizare dintre care 2 extind unul dintre cazuri , datorita faptului ca strategia prin care un client alege o coada poate fi de doua feluri : clientul alege coada la care ar astepta cel mai putin in functie de cantitatea de procesare pe care o au clientii din fata lui , clientul alege coada cu numarul minim de clienti din fata lui.

Actorii sistemului sunt : clientul si managerul cozii .Clientul poate fi interesat de timpul pe care il petrece in coada , sau de durata timpului de procesare pe care o are in coada in functie de coada aleasa. Managerul cozii are nevoie de a prezenta un raport legat de timpii medii de asteptare respectiv de servire a unui client din coada in care acesta e manager , precum si timpul in care au fost cei mai multi clienti prezenti in coada sau timpul in care coada a fost goala.



Nota: Aplicatie ruleaza pe mai multe fire de executie ; numarul lor e egal cu numarul de cozi ,existand astfel un manager pentru fiecare coada in executie . Comportamenul fiecarei cozi e similar motiv pentru care diagrama de mai sus arata astfel.

Se prezinta urmatoarele use case-uri :

**• Use Case : Choose queue**

**• Primary Actor: Clientul care se aseaza la coada**

**• Secondary actor: Managerul cozii**

**• Main Success Scenario :**

**1.** Utilizatorul este ghidat de interfata grafica pentru introducerea datelor de simulare .

**2.** Utilizatorul introduce datele .

**3.** Datele sunt procesate de clasele implicate (se stabileste strategia de alegere a cozii) .

**4**. Se porneste frame-ul simularii (sunt generate casele de marcat, clientii ,managerii de case).

**5.** Se alege casa de marcat preferata .

**6.** Managerul cozii proceseaza datele clientului asezat la coada .

**7.** Se genereaza statistici .

**• Use Case: Find average wait time**

**• Primary Actor: Clientul din coada**

**• Secondary actor: Managerul cozii**

**1.** Se stabileste timpul de asteptare pentru fiecare client din coada

**1.1.** Timpul de asteptare = timpul de procesare al clientului nou venit + timpul de asteptare pentru clientul din fata lui

**2.** Se realizeaza media timpilor de asteptare din coada

**3.** Managerul cozii proceseaza acest timp .

**4**. Se genereaza statistici .

**Nota:** Use case-ul „Find average service time” e similar cu cel de procesare al timpului de asteptare atat ca media se va face doar cu timpii de procesare pentru fiecare client in parte .

**• Use Case: Find peak hour**

**• Secondary actor (only) : Managerul cozii**

1. Se delimiteaza unitatea de timp pentru care se calculeaza numarul de clienti ( cod : ”timeSlice”)
2. Se imparte timpul de simulare in unitatile de timp stabilite mai sus
3. Rezulta mai multe intervale de timp canditat
4. Pentru fiecare interval de timp canditat se cauta clientii din coada care au timpii de sosire cuprinsi in acest interval
5. Se cauta maximul numarului de clienti din aceste intervale
6. Se genereaza statistici

**Nota :** Generarea statisticilor inseamna adunarea de informatii din diverse use case-uri si centralizarea lor pentru prezentarea lor la finalul simularii .

**• Use Case: Find empty queue time**

**• Secondary actor (only) : Managerul cozii**

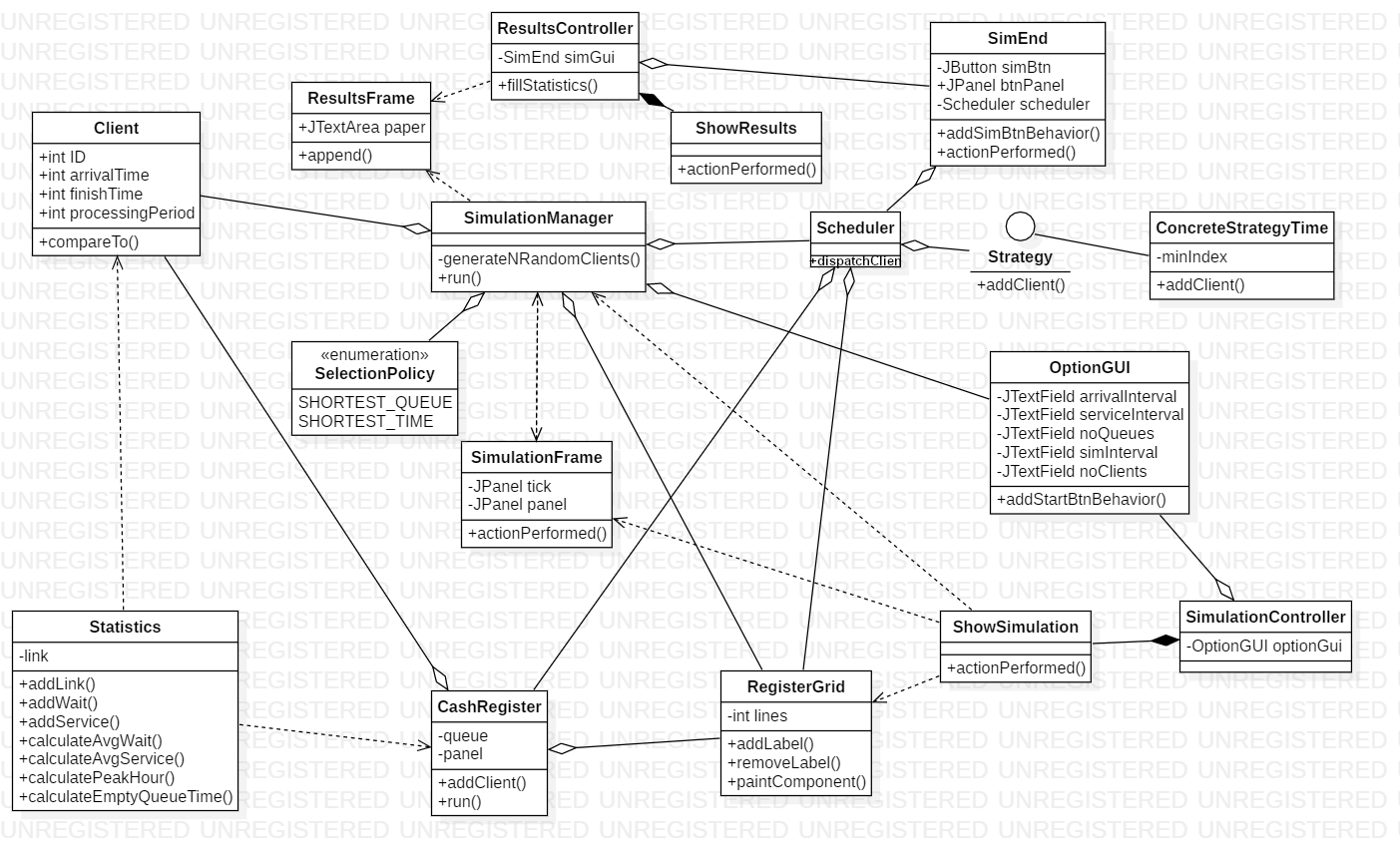
1. Se parcurge fiecare coada in acelasi timp ( managerul simularii )
2. Daca o coada nu are niciun client intr-o unitate de timp, timpul cautat creste cu o unitate
3. Se realizeaza suma timpilor cautati
4. Se genereaza statistici

**Capitolul 3. Proiectare (decizii de proiectare, diagrame UML, structuri de date, proiectare clase, interfete, relatii, packages, algoritmi, interfata utilizator)**



Proiectul e structurat in 3 pachete care prezinta arhitectura Model-View-Controller a proiectului. Se pot observa urmatoarele relatii de dependenta intre pachete din diagrama de pachete :

**Nota :** Se poate observa ca atat view cat si model depind una de cealalta

Se detaliaza in continuare continutul fiecarui pachet in urmatoarea diagrama de clase :

**Nota:** Nu toate variabilele instanta si metodele au fost incadrate in diagrama.

**Pachetul model :**

Pachetul contine urmatoarele clase : Client, CashRegister, Scheduler, Statistics, CocreteStrategyTime ; interfata Strategy si enumerarea SelectionPolicy.

Scheduler contine lista caselor de marcat pe care initiaza fire de executie individuale. Client contine informatii despre ID-ul clientilor implicati in simulare,timpii de sosire,finalizare si procesare. Clasa CashRegister contine coada de clienti iar clasa Statistics are metode utile pentru generarea de statistici la finalul simularii. ConcreteStrategyTime implementeaza interfata Strategy si implicit metoda addClient care adauga un client la coada cu timpul minim de asteptare.

**Pachetul view :**

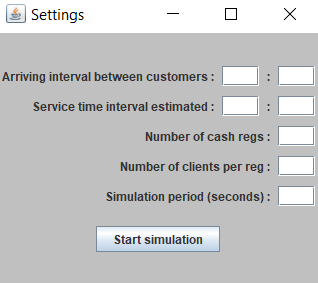
Pachetul contine clasele constituente ale interfetei grafice : OptionGUI (prima fereastra care se deschide la rularea aplicatiei) in ale carei campuri utilizatorul introduce datele pentru simulare , RegisterGrid care realizeaza dinamica aplicatiei(animatia cu casele de marcat si clientii), SimEnd e un panou care se afiseaza la terminarea simularii si ResultsFrame care dupa apasarea unui buton de confirmare afiseaza informatiile rezultate din simulare si statistici.

**Pachetul controller :**

Are doua clase utile : SimulationController si ResultsController care ataseaza ascultatori butoanelor din panoul de optiuni si cel de la finalul simularii . Totodata clasele din pachetul controller realizeaza tranzitia corecta a actiunilor din interfata grafica .

**Capitolul 4.Implementare**

Urmeaza detalii legate de implementarea celor mai importante clase , a celor mai importante campuri si metode prin parcurgerea pas cu pas a unui caz normal de functionare a aplicatiei .

La pornirea aplicatiei apare un prompt pentru introducerea datelor necesare simularii. Aceste date fac legatura cu urmatoarele campuri :

**public class OptionGUI extends JFrame {  
...  
 private JTextField arrivalDataFrom;  
 private JTextField arrivalDataTo;  
 private JTextField serviceDataFrom;  
 private JTextField serviceDataTo;  
 private JTextField noQueuesData;  
 private JTextField simData;  
 private JTextField noClientsData;**

SimulationController adauga ascultator butonului de Start simulation, iar la apasarea acestuia se executa urmatorii pasi prezenti in ascultatorul implentat de SimulationController :

**RegisterGrid regGrid=new RegisterGrid(noQ);** // se pregateste continutul panoului de animatii **new SimulationFrame(regGrid);** // se afiseaza frame-ul pentru animatii **SimulationManager gen=new SimulationManager(optionGui,regGrid);** // se pregateste managerul simularii  **Thread t=new Thread(gen);** // acesta urmareste toate cozile in acelasi timp **t.start();**

La instantierea unui nou SimulationManager sunt generati clientii prin metoda proprie generateNRandomClients :

**for(int i=0;i<numberOfClients\*numberOfRegs;i++){  
 int arrInt=(maxArriveTime-minArriveTime);  
 int procInt=(maxProcTime-minProcTime);  
 int arr=((int)(Math.*random*()\*10))%arrInt+minArriveTime;   
 int pro=((int)(Math.*random*()\*10))%procInt+minProcTime;   
 global+=arr;  
 Client c=new Client((i+1),global,pro);  
 generatedClients.add(c);  
}  
...**

arrInt si procInt sunt intervalele in care se genereaza numere la intamplare; aceste numere reprezinta timpii de sosire si de procesare a clientilor noi generati carora se atribuie un id .

Dupa executarea ultimei linii din constructorul lui SimulationManager , metoda run intra in executie :

**while(*currentTime*<timeLimit){  
 for(int i=0;i<generatedClients.size();i++) {  
 if ( generatedClients.get(i).getArrivalTime() == *currentTime*) {  
 scheduler.dispatchClient(generatedClients.get(i));  
 ...  
 }**

**...**

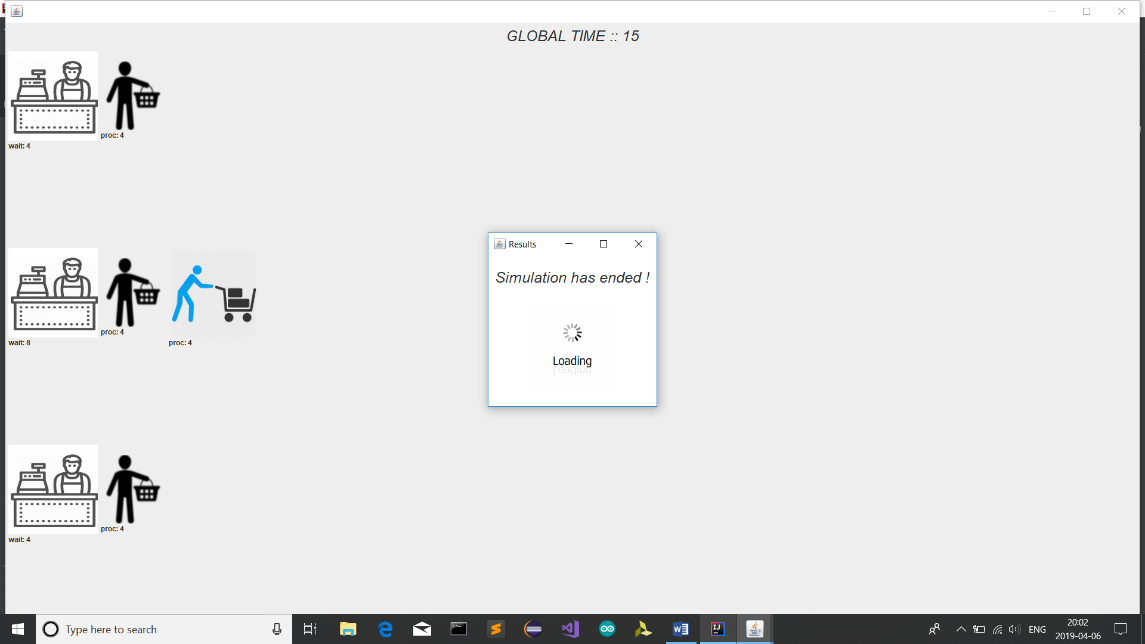
**currentTime++;**

**Thread.sleep(1000);**

Se adauga cate un client o data la 1 secunda in coada aleasa de strategia implementata , in cazul nostru strategia se refera la alegerea cozii cu timpul minim de asteptare , metoda suprascrisa de clasa ConcreteStrategyTime care implementeaza metoda addClient din interfata Strategy :

**public void addClient(List<CashRegister> regs, Client c) {  
 minWaitTime=regs.get(*minIndex*).getWaitingPeriod().get();  
 for(int i=0;i<regs.size();i++){  
 int val=regs.get(i).getWaitingPeriod().get();  
 if(val < minWaitTime){  
 minWaitTime= val ;  
 *minIndex*=i;  
 }  
 }  
 regs.get(*minIndex*).addClient(c);  
}**

Se observa ca prima linie din metoda se refera la faptul ca urmeaza sa se faca o comparatie cu ultima coada aleasa anterior prin indexul sau minim. Indexul minim se stabileste in felul urmator : Se compara fiecare coada cu fiecare pentru a stabili care e coada cu timpul minim de asteptare si i se tine minte indexul dupa care la coada cu acel index se adauga un client.

Pasii de mai sus se tot repeta pana cand timpul global ajunge la limita impusa de utilizator. Acest lucru poate fi observat aici :

Fiecare casa de marcat reprezinta o coada si un fir de executie care lucreaza independet unul fata de altul dar supravegheati de SimulationManager.

Casa de marcat e reprezentata de clasa CashRegister care foloseste ca structura de date un BlockingQueue pentru modelarea cozii de clienti . Ca metode importante regasim addClient, si run , fiind vorba de o clasa care implementeaza interfata Runnable :

**public void addClient(Client newClient){**  
 **queue.put(newClient);   
 waitingPeriod.addAndGet(newClient.getProcessingPeriod();  
 panel.addLabel(Integer.*parseInt*(this.name)-1,newClient,waitingPeriod.intValue());**Se aduaga un obiect de tip Client in coada , se incrementeaza perioada de asteptare a cozii cu timpul de procesare al clientului nou venit si se creeaza o imagine in RegisterGrid pentru acest client , imagine care va fi pusa in dreptul imaginii reprezentative pentru aceasta coada in aceeasi clasa.

Ultimul pas al executiei aplicatiei este generarea de informatii referitoare la simulare . Pentru aceasta Clasa Statistics este responsabila de generarea lor si clasa ResultsFrame de afisarea lor intr-un JTextArea.

Metodele clasei Statistics sunt statice si publice pentru a putea fi accesate de diversele clase cu care intra in contact , si apelate in ordinea fireasca a executiei aplicatiei .Printrea acestea enumeram :

**public static String getGlobalPeakHour(int timeSlice) {  
 int a=0;  
 int b=timeSlice;  
 int max=0;  
 int c=a;  
 while (b<=*simInterval*-timeSlice){   
 int value=*getNumberOfClientsInInterval*(a, b);  
 if(value >= max){  
 max=value;  
 c=a;  
 }  
 a=b;  
 b+=timeSlice;  
 }  
 return ""+c+" to "+(c+timeSlice);  
}**

Aceasta metoda parcurge timpul de simulare cu pasul egal cu timeSlice , astfel se imparte in mai multe intervale. Pentru fiecare interval se numara clientii care au timpul de sosire in acest interval . Dintre acest numere se preia maximul si se returneaza , valoarea de return fiind folosita de clasa ResultsFrame amintita mai sus.

**public static void addWait(CashRegister cr,int waitPer){  
 if(*waitPerQueue*.get(cr)!=null) {  
 int old = *waitPerQueue*.get(cr);  
 old+=waitPer;  
 *waitPerQueue*.put(cr, old);  
 }  
 else  
 {  
 *waitPerQueue*.put(cr, waitPer);  
 }  
}**

Metoda adauga timp de asteptare la o coada , dupa cum ii spune si numele . Se cauta coada in variabila waitPerQueue care e un HashMap intre CashRegister si un Integer , dupa care se actualizeaza valoarea de le acea coada din acest camp.

**public static int calculateAvgWaitPerQueue(CashRegister cr){  
 if(*waitPerQueue*.get(cr)!=null && *clientsInIntervalPerQueue*.get(cr)!=null) {  
 int total= *waitPerQueue*.get(cr);  
 return (total / *clientsInIntervalPerQueue*.get(cr));  
 }  
 return -1;  
}**

Metoda preia numarul de clienti care au participat pe timpul simularii prin campul clientsInInterval de tipul HashMap<CashRegister,Integer> si imparte timpul total de asteptare pe coada la acest numar,dupa care il returneaza.

**Capitolul 5. Concluzii**

Subiectul proiectului este complex , si prezinta multe posibilitati de implementare insa varianta aleasa a adus urmatoarele cunostiinte dobandite : realizarea de diagrame de pachete, diagrame use case , metode de implementare a sistemelor bazate pe cozi respectand paradigma OOP, realizarea de fire de executie si cel mai important : realizarea corespunzatoare a unui proiect Java de la proiectare pana la implementare ,testare si crearea unei documentatii.

Proiectul poate fi dezvoltat in multe directii , dar propunerea adusa ar fi o mai buna parametrizare a interfetei grafice , generarea mai multor informatii relevante pentru simulare , si introducerea unor module care sa permita monitorizarea mai multor magazine in acelasi timp ( fire de executie pentru fiecare magazine ).

**Capitolul 6:Bibliografie**

Interfata cu utilizatorul:-componenta JScrollPane :

<https://stackoverflow.com/questions/7766844/java->

Utilizarea AtomicInteger :

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

Utilizarea ArrayBlockingQueue :

<http://tutorials.jenkov.com/java-util-concurrent/arrayblockingqueue.html>