**DOCUMENTAŢIE TEMA 2**

**APLICAŢIE DE SIMULARE**

**A UNEI COZI DE AŞTEPTARE**

**Nume prenume : Udrea Marius**

**Grupa :30229**

**Profesor Laborator :Teodor Petrican**

Contents

[1. Cerinţe Funcţionale 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. Concluzii si Dezvoltări Ulterioare 4](#_Toc476131456)

[7. Bibliografie 4](#_Toc476131457)

# Cerinte Functionale

(deduse din barem si cerinţa problemei)

Proiectarea și implementarea unei aplicații de simulare care vizează analiza sistemelor bazate pe așteptare pentru determinarea și minimizarea timpului de așteptare al clienților.

Spre exemplu , un client se aşează la coadă la un magazin , după un timp x este servit şi vine un alt client în locul lui .Folosind aceste date putem afla timpul mediu de aşteptare la o coada , pentru a-l putea reduce;

# Obiective

## Obiectiv Principal:

(obiectivul principal al proiectului)

Cozi sunt frecvent utilizate pentru a modela domenii din lumea reală. Obiectivul principal al unei coazi este de a oferi un loc pentru un "client" să aștepte înainte de a primi un "serviciu"..Gestionarea sistemelor bazate pe coada de așteptare este interesată de minimizarea timpului pe care clienții lor îl așteaptă în cozile de așteptare înainte ca aceştia să fie serviţi. O modalitate de a minimiza timpul de așteptare este adăugarea mai multor servere, adică mai multe cozi în sistem (fiecare coadă este considerată ca având un procesor asociat), dar această abordare mărește costurile furnizorului de servicii. Când se adaugă un nou server, clienții care așteaptă vor fi distribuiți în mod egal la toate cozile curente disponibile.

Aplicația ar trebui să simuleze o serie de clienți care sosesc pentru serviciu, introducând cozi, așteptând, servind și părăsind în final coada. Urmărește timpul petrecut de clienți în așteptare în cozi și scoate timpul mediu de așteptare. Pentru a calcula timpul de așteptare, trebuie să știm timpul de sosire, timpul de ieşire și timpul de servire. Timpul de sosire și timpul de servire depind de clienții individuali - când apar și de câte servicii au nevoie. Timpul de ieşire depinde de numărul de cozi, de numărul de clienți din coadă și de nevoile lor de serviciu.

**Date de intrare:**

- intervalul minim și maxim de timp de sosire între clienți;

- timpul minim și maxim de servire;

- numărul de cozi;

- interval de simulare;

- alte informații pe care le considerați necesare;

**Date de ieşire:**  
 - media timpului de așteptare, timpului de servire și timpului de așteptare pentru cozile de așteptare 1, 2 și 3 pentru intervalul de simulare și pentru un interval specificat (pot fi luate în considerare și alte informații utile);

- înregistrarea evenimentelor și a datelor principale ale sistemului;

- Evoluția coazi;

- oră de vârf pentru intervalul de simulare;

## Obective Secundare:

(pasii care trebuie urmati pentru a atinge obiectivul principal)

|  |  |  |
| --- | --- | --- |
| **Obiectiv Secundar** | **Descriere** | **Capitol** |
| Alegerea structurilor de date |  | 4 |
| Impartirea pe clase |  | 4 |
| Dezvoltarea algoritmilor |  | 4 |
| Implementarea solutiei |  | 4 |
| Testare |  | 6 |

# Analiza Problemei

Prin analiza problemei, ne referim la un prim set abstract de operații și proprietăți prin care încercăm să depistăm eventualele însușiri și comportamente ale proceselor necunoscute. Programarea orientată ne oferă aici un avantaj clar, tocmai fiindcă ea permite să taclăm problema de la un nivel superiror, fără a mai fi constrâși, într-o așa măsură, de caracteristicile tehnice.

Această strategie de conceptualizare, mai poartă numele și de bottom-up design. Este foarte avantajoasă din prisma găsirii componentelor constituente, deoarece pot fi găsite, relativ ușor, structuri cu o legătură directă în lumea reală( obiecte, acțiuni etc.). Din păcate această versatilitate vine cu prețul complexității, ea crescând pe măsură ce se avansează pe nivelele inferioare.

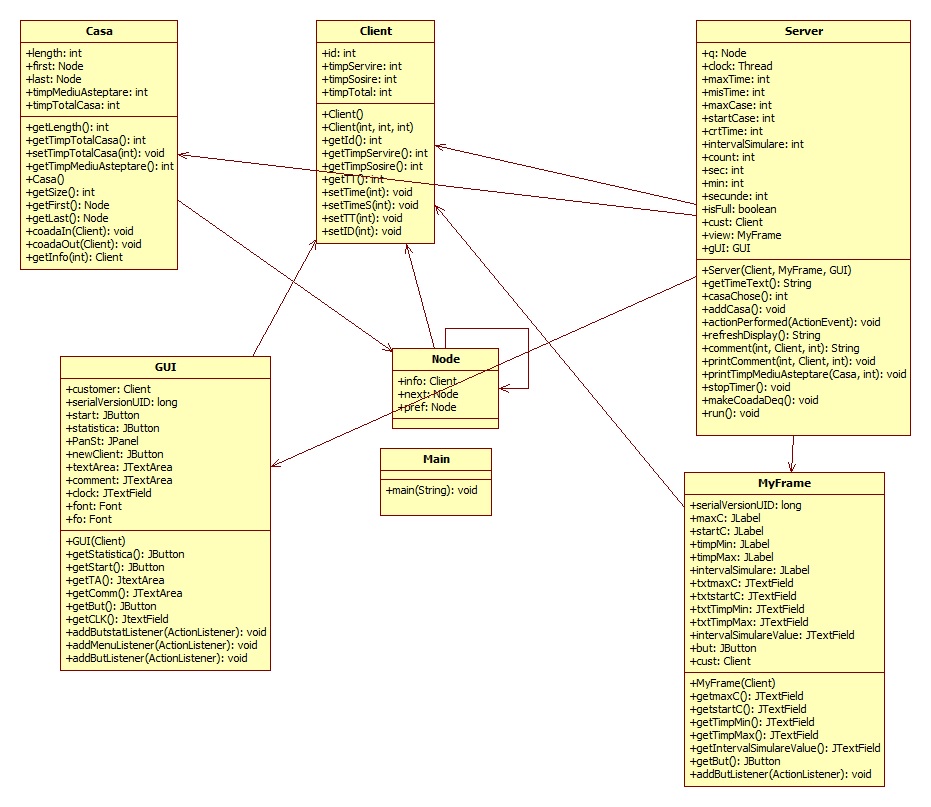
De cele mai multe ori se pornește de la specificația proiectului, căutându-se:

* Substantive, care devin eventuale clase candidat
* Verbe ce ar putea juca rolul metodelor din clasă.

Odată realizat pasul de mai sus, ar trebui să avem o idee foarte generală asupra problemei. Pasul următor constă în descrierea funcțională a acesteia. Ce trebuie să facă aplicația la intrările X?

Programul va putea fi accesat de un număr ridicat de persoane, de aceea interfața cu utilizatorul devine punctul de pornire al proiectului. Ea trebuie să permită, într-o manieră cât mai convenabilă, comunicarea utilizatorului cu aplicația.

# Proiectare



Aplicaţia de simulare va avea 7 clase:

1.Clasa **Node –** cu ajutorul ei definim a structura de tip lista dublu înlanţuită

2.Clasa **Casa** – aceasta clasă crează o coadă de aşteptare unde vor intra şi ieşi “clienţi”, şi ne oferă informaţi despre timpul total şi timpul mediu de aşteptare

3.Clasa **Client –** ne ofera informaţii despre fiecare client cum ar fi :id , momentul în care s-a aşezat la coadă , momentul în care a plecat , dar şi timpul cît a fost servit

4.Clasa **MyFrame -**cu ajutorul ei se creează panoul prin care introducem date în proiect. Tot aici se instanțiază o parte din asculatători. Ei au rolul de a înregistra acţiunile utilizatorilor şi de a le transmite mai departe

5.Clasa **Server-** are rolul de a crea o instanţă de tip thread pentru un timer , dar şi pentru a afişa un log de evenimente , cu ajutorul ei se creează interfața grafică folosită în tot proiectul.

6.Clasa **GUI -**   aici se instanțiază o parte asculatătorii folositi pentru a adauga clienţi , pentru a începe simularea sau pentru a afişa statistici

7.Clasa **Main –** clasa principal a proiectului

4.1 Clasa Node

Creaza o structură de tip lista dublu înlanţuită pe care o vom folosi să ţimen informaţii despre ordinea clienţilor

Client **info**;  
Node **next**, **prev**;

4.2 Clasa Casa

Creăm o structură de tip FIFO pentru a simula intrarea şi ieşirea clienţilor de la coada.

Node **first**;  
Node **last**;

Operaţia de incrementare : verificăm dacă coada e goală , şi dacă e inserăm la început ,altfel

inserăm în continuare.

**public void** coadaIn (Client info){**assert** info != **null** ;  
 **int** sizePre = getSize();  
 **if** (sizePre == 0){ Node node = **new** Node();  
 node.**info** = info;  
 node.**next** = **null**;  
 node.**prev** = **null**;  
 **first** = node;  
 **last** = node;  
 **length**++;  
 }  
 **else**{ Node n = **last**;  
 Node node = **new** Node();  
 node.**info** = info;  
 node.**next** = **null**;  
 node.**prev** = n;  
 n.**next** = node;  
 **last** = node;  
 **length** ++;  
 }

Operaţia de decrementare :verificăm dacă coada mai are elemente , şi dacă mai are ştergem primul element şi îl mutăm pe urmatorul în locul lui , altfel coada e goală.

**public void** CoadaOut(){**assert** getSize() > 0;  
 **if** (**this**.**first** != **this**.**last**)  
 {Object info = getInfo(0);  
 Node n = **first**;  
 **first** = n.**next**;  
 **first**.**prev** = **null**;  
 **length**--;  
 **assert** n.**info** == info;  
 }  
 **else** { **first** = **null**;  
 **last** = **null**;  
 **length** = 0;  
 }  
}

4.3 Clasa Client

Este clasa care crează şi generează clienţii care se aşează la coada.De asemenea ne oferă informaţii utile despre fiecare client cum ar fi :id-ul , momentul în care s-a aşezat la coadă , momentul în care a plecat , dar şi timpul cît a fost servit

**public int** getID(){  
 **return this**.**id**;  
}  
**public int** getTimpServire(){  
 **return this**.**timpServire**;  
}  
**public int** getTimpSosire(){  
 **return timpSosire**;  
}  
**public int** getTT(){ **return timpTotal**; }

4.4 Clasa MyFrame

Este clasa care creaza panel-ul în care utilizatorul v-a introduce date .De asemenea crează ascultatorii , şi unul dintre butoane şi anume butonul “ok” care începe simularea.

**public** JTextField getmaxC(){  
 **return txtmaxC**;  
}  
**public** JTextField getstartC(){  
 **return txtstartC**;  
}  
**public** JTextField getTimpMin(){  
 **return txtTimpMin**;  
}  
**public** JTextField getTimpMax(){  
 **return txtTimpMax**;  
}  
**public** JTextField getIntervalSimulareValue(){  
 **return intervalSimulareValue**;  
}  
**public** JButton getBut(){  
 **return this**.**but**;  
}  
  
**public void** addButListener(ActionListener act){  
 **but**.addActionListener(act);  
}

4.5 Clasa Server

Este clasa care instanţiază un thread pentru un timer care va funcţiona ca parametru de testare în cadrul proceselor de testare.

Thread **clock**;

Metoda de rulare va acţiona astfel : va compara valoarea timerului cea introdusa de utilizator pentru intervalul de simulare , şi cît timp e mai mică aceasta îi va comanda ceasului să se incrementeze cu 1 secundă dupa care va “adormi” pentru 1000 milisecunde.La final va comanda decrementarea cozii şi reîmprospatarea desenului .

**public void** run() {  
  
 **while** (**clock** != **null** && **secunde** <= **intervalSimulare**) {  
 **gUI**.getCLK().setText(getTimeText());  
 **try** {  
 **clock**.*sleep*(1000);  
 } **catch** (InterruptedException e) {  
 System.***out***.println(e);  
 }  
 **if** (**sec** < 59)  
 **sec**++;  
 **else** {  
 **min**++;  
 **sec** = 0;  
 }  
 **secunde**++;  
 makeCoadaDeq();  
 refreshDisplay();  
 **gUI**.getTA().setText(refreshDisplay());  
 }

Pe de altă parte tot în această clasă se crează şi log-ul de evenimente care afişează cînd un “client” vine la coadă , cînd iese , calculează timpul mediu de aşteptare şi îl afişează ,desenează o “casă”, şi reîmprospatează desenul .

4.6 Clasa GUI

Această clasă e responsabilă pentru o butoanele din cadru interfaţei grafice şi anume cele pentru începerea simulării, pentru a adăuga un client nou , dar şi cel pentru a arăta statistici.Deasemenea clasa face şi design-ul la butoane , la bordure etc.

**this**.setSize(1000,800);  
**this**.setTitle(**"Magazin"**);  
**this**.setLayout(**null**);  
**this**.setBackground(**new** Color(0, 0, 0));  
**textArea**.setFont(**new** Font(**"Serif"**,Font.***PLAIN***,30));  
**textArea**.setBackground(**new** Color(204, 204, 255));  
**comment**.setFont(**new** Font(**"Serif"**,Font.***ITALIC***,15));

4.7 Clasa Main

Această clasă e cea mai important deoarece ea porneşte execuţia programului.Ea conţine metoda main care este apelata la începutul programului.

**public static void** main(String[] args) {  
  
 Client cust = **new** Client();  
 GUI gUI = **new** GUI(cust);  
 MyFrame frame = **new** MyFrame(cust);  
 gUI.setDefaultCloseOperation(gUI.***EXIT\_ON\_CLOSE***);  
 Server control = **new** Server(cust, frame, gUI);  
}

# Concluzii si Dezvoltari Ulterioare

Această temă m-a învătat despre importanţa thread-urilor , şi mai mult despre împarţirea problemei în mai multe subprobleme. Nu o dată, a trebuit să elimin bucăți semnificative de cod fiindcă încercam să compensez lipsa de rigurozitate din faza de planificare cu clauze suplimentare.Acest proiect a reprezentat o provocare pentru mine ,pe care nu ştiam dacă o pot duce pînă la capăt deoarece nu mai lucrasem înainte cu thread-uri .

Sper că această aplicaţie va constitui un model de dezvoltare pentru o alicaţie care să îmbunatăţească timpii de aşteptare pentru magazine (cum ar fi LIDL-ul din Sigma unde trebuie să aştepţi foarte mult timp deoarece nu stiu cănd să deschidă o nouă casă).

# Bibliografie

-ceva tutorial pe YouTube

-ceva modele de proiecte de pe Google