DOCUMENTATIE

TEMA 2

NUME STUDENT: MITROI BIANCA

GRUPA:30229

# CUPRINS

[1. Obiectivul temei 3](file:///C:\Users\Mitroi%20Bianca\OneDrive\Desktop\fisiere%20facultate%20anul%202%20sem%202\TPF\laboratoare%20tpf\PT2021-2022_Documentation_Template.doc#_Toc95297885)

[2. Analiza problemei, modelare, scenarii, cazuri de utilizare 4](file:///C:\Users\Mitroi%20Bianca\OneDrive\Desktop\fisiere%20facultate%20anul%202%20sem%202\TPF\laboratoare%20tpf\PT2021-2022_Documentation_Template.doc#_Toc95297886)

[3. Proiectare 5](file:///C:\Users\Mitroi%20Bianca\OneDrive\Desktop\fisiere%20facultate%20anul%202%20sem%202\TPF\laboratoare%20tpf\PT2021-2022_Documentation_Template.doc#_Toc95297887)

[4. Implementare 7](file:///C:\Users\Mitroi%20Bianca\OneDrive\Desktop\fisiere%20facultate%20anul%202%20sem%202\TPF\laboratoare%20tpf\PT2021-2022_Documentation_Template.doc#_Toc95297888)

[5. Rezultate 13](file:///C:\Users\Mitroi%20Bianca\OneDrive\Desktop\fisiere%20facultate%20anul%202%20sem%202\TPF\laboratoare%20tpf\PT2021-2022_Documentation_Template.doc#_Toc95297889)

[6. Concluzii 16](file:///C:\Users\Mitroi%20Bianca\OneDrive\Desktop\fisiere%20facultate%20anul%202%20sem%202\TPF\laboratoare%20tpf\PT2021-2022_Documentation_Template.doc#_Toc95297890)

[7. Bibliografie 16](file:///C:\Users\Mitroi%20Bianca\OneDrive\Desktop\fisiere%20facultate%20anul%202%20sem%202\TPF\laboratoare%20tpf\PT2021-2022_Documentation_Template.doc#_Toc95297891)

# Obiectivul temei

Atat din punct de vedere academic cat si practic, efectuarea temei ajuta la intelegerea conceptului de fir de executie a unui program si cum acesta dispune la un anumit moment de timp de anumite resurse puse la dispozitie - memorie - , dar si a celui de rulare concurenta a unor secvente de cod, concurenta fiind intre mai multe fire de executie. Aplicabilitatea practica rezulta chiar din scenariul de implementat, si anume simularea cozilor ce se formeaza la casele de marcat ale unui magazin.

# Analiza problemei, modelare, scenarii, cazuri de utilizare

# Analiza problemei

Pentru a se pastra generalitatea si pentru usurinta scrierii, problema are anumite limite. Evolutia in timp a cozilor se poate vedea numai in intervalul de timp pe care utilizatorul programului il cere. Clientii au un anumit interval de timp la care pot sosi la case si o anumita perioada de timp in care pot fi serviti, aceste valori fiind specificate de asemenea de catre utilizator. Programul genereaza clientii in limitele stabilite de utilizator si ii pune la cozi in functie de strategia implementata. La momentul sosirii este asezat la casa cu timpul de asteptare cel mai mic. Clientii odata generati sunt ordonati crescator in functie de timpul de sosire si daca mai multi clienti sosesc in acelasi timp, acestia sunt sortati crescator in functie de timpul de serviciu. Mereu primului client din coada ii scade timpul de serviciu, iar timpul de asteptare la coada scade odata cu acesta. Cand clientul a fost servit, adica timpul lui de serviciu este 0, acesta este scos din coada si se trece la urmatorul client. Simularea e opreste in momentul in care cronometrul ajunge la limita maxima specificata sau cand nu mai exista client ce trebuie serviti.

# Modelarea

*La momentul pornirii aplicatiei, utilizatorul are la dispozitie o fereastra cu sapte campuri in care poate complete pe rand:*

* *Number of queues – numarul de cozi pe care il va avea magazinul;*
* *Number of clients – numarul de clienti pe care il va avea magazinul;*
* *Simulation time – timpul maxim pe care se poate desfasura simularea;*
* *Smallest arrival time – clientii pot ajunge cel mai devreme la acest timp;*
* *Largest arrival time - clientii pot ajunge cel mai tarziu la acest timp;*
* *Smallest service time – durata servirii clientilor nu poate fi mai mica decat numarul specificat in acest camp*
* *Largest service time – durata servirii clientilor nu poate fi mai mare decat numarul specificat in acest camp*

*La momentul apasarii butonului Continue, se incepe simularea. Numele fiecarei cozi este specificat in partea stanga a ferestei nou deschise, urmand in dreptul lor sa vina la momentele de timp corespunzatoare clientii repartizati la cozile care au la momentul sosirii clientului timpul de astepatre cel mai mic. Clientii sunt afisati cu id, timp de sosire si timp de serviciu in aceasta ordine cu spatiu. Delimitarea clientilor la cozi se face prin virgula. In momentul cand clientul este primul la coada, timpul de serviciu al acestuia incepe sa scada pe masura ce este incrementat cronometrul, urmand sa dispara din coada cand timpul lui de serviciu ajunge la 0.*

*Sub cozi sunt afisate in ordine de la stanga la dreapta cronometrul simularii, timpul mediu de asteptare la coada calculat ca media aritmetica a timpilor de asteptare ai tuturor cozilor si timpul mediu de serviciu calculat ca media aritmetica a timpilor de serviciu ai clientilor care sunt primii la coada, toate acestea fiind rezultate afisate in timp real.*

# Cazuri de utilizare

*In functie de ce introduce utilizatorul de la tastatura, dar si in functie de ce timpi genereaza algoritmul pentru fiecare client, exista mai multe scenarii posibile pe care programul le gestioneaza corespunzator:*

* *Daca utilizatorul nu introduce nimic intr-un anumit camp, i se va afisa un mesaj de eroare in care se va cere sa se introduca un numar valid*
* *Daca de asemenea se introduce numere prin care nu se poate realiza implementarea, se va specifica acest lucru*
* *In cazul in care anumiti clienti vor fi asezati la coada sau vor ramane ordonati in ce a generat programul, iar timpul de simulare a ajuns la limita maxima, fereastra va “ingheta” in formula in care era la momentul terminarii simularii*
* *Daca in schimb, toti clientii au fost serviti, dar timpul de simulare nu s-a terminat, simularea se va intrerupe*
* *Daca mai multi clienti sosesc in acelasi timp, timpul de simulare nu va creste pana ce toti clientii cu acelasi timp de sosire ca timpul current de simulare nu vor fi sortati la cozi.*

# Implementare

*Interfata grafica*

*Graphical user interface, text, application, email

Description automatically generated*

Graphical user interface, text

Description automatically generated

*Diagrama UML*

Graphical user interface

Description automatically generated with medium confidence

# Proiectare

# Model

Campurile din toate clasele create sunt private, ceea ce insemna ca sunt necesare gettere si settere pentru a le accesa, respective a le modifica.

# a.1 Client

Campuri: -

private int id; // id-ul clientului  
private int arrival; // momentul sosirii  
private int service; // timpul de serviciu

Metode: -

public int getId*(){* return this.id;  
*}*public int getArrival*(){* return this.arrival;  
*}*public int getService*(){* return this.service;  
*}*public void setService*(){* service--;  
*}*@Override  
public int compareTo*(*Client o*) {* if*(*this.arrival == o.arrival*)* return this.service - o.service;  
 return this.arrival - o.arrival;  
*} // metoda ce reprezinta criteriile de sortare a clientilor*public String toString*() {* return "" + id + " " + arrival + " " + service + " ";  
*} // afiseaza clientii in modul specificat anterior la modelarea problemei*

# a.2 Server

Campuri:

private BlockingQueue*<*Client*>* clients; // lista de clienti a fiecarei cozi  
private AtomicInteger waitingTime; // timpul de asteptare la o coada  
private Thread thread = new Thread*(*this*)*; // firul de lucru ce se ocupa de //scaderea timpului de serviciu, a timpului de asteptare la coada si de //scoaterea clientilor din coada

Metode:

public void addClients*(*Client newClient*)  
{* clients.add*(*newClient*)*;  
 waitingTime.set*(*getWaitingTime*()* + newClient.getService*())*;  
*}  
//adauga clientii in coada si creste timpul de asteptare*public BlockingQueue*<*Client*>* getClients*()  
{* return this.clients;  
*}*public int getWaitingTime*()  
{* return waitingTime.intValue*()*;  
*}*@Override  
public void run*() {* while*(*true*) {* Client c = clients.peek*()*;  
 while*(*c != null && c.getService*()* != 0*) {* try *{* Thread.*sleep(*1000*)*;  
 *}* catch *(*InterruptedException e*) {* // *TODO Auto-generated catch block* e.printStackTrace*()*;  
 *}* c.setService*()*;  
 waitingTime.set*(*getWaitingTime*()* - 1*)*;  
 *}* clients.poll*()*;  
 *}  
  
}//metoda declansata de firul de lucru ce are ca target coada (camp al clasei Server)*

# View

# b.1 View

Campuri – Label-urlie corespunzatoare numelor caselor si cele care afiseaza cozile, respectiv cele care specifica timpul de simulare, timpul mediu de astrptare, timpul mediu de servire a unui client si cel care arata lista de client ordonati si de pus la cozi

Metode –

public void setLblNewLabel*(*String s, int i*) {* labels.get*(*i*)*.setText*(*s*)*;  
*}*public void setTimeLabel*(*String s*) {* lblNewLabel\_6.setText*(*s*)*;  
*}*public void setClientLabel*(*String s*) {* lblNewLabel\_8.setText*(*s*)*;  
*}*public void setAverageWaitingLabel*(*String s*) {* lblNewLabel\_11.setText*(*s*)*;  
*}*public void setAverageServiceLabel*(*String s*) {* lblNewLabel\_12.setText*(*s*)*;  
*}*

# b.2 InputView

Campuri – Label-urlie corespunzatoare parametrilor ce trebuie introdusi de la tastature, campurile in care se face introducerea si butonul care porneste simularea

Metode – Cele care returneaza sirurile de caractere introduce de la tastaura si cea care receptioneaza apasarea butonului Continue.

# Test

# c.1 Test

package test;  
import java.io.FileWriter;  
import java.io.IOException;  
  
import controller.Controller;  
public class Test *{* public static void main*(*String*[]* args*) {* Controller controller = new Controller*()*;  
  
 *}  
}*

# Controller

# e.1 Controller

package controller;  
import java.util.Random;  
  
import javax.swing.JOptionPane;  
  
import view.InputView;  
import view.View;  
import java.awt.event.\*;  
import java.io.File;  
import java.io.FileWriter;  
import java.io.IOException;  
import java.util.ArrayList;  
import java.util.Collections;  
//import java.util.Random;  
//import java.util.Timer;  
//import java.util.TreeSet;  
  
import model.Client;  
import model.Server;  
//import test.StoreManager;  
public class Controller implements Runnable*{* private InputView inputView = new InputView*()*;  
 private View view = new View*()*;  
 private ArrayList*<*Server*>* servers = new ArrayList*<*Server*>()*;  
 private ArrayList*<*Client*>* clients = new ArrayList*<*Client*>()*;  
 private ArrayList*<*Client*>* served = new ArrayList*<*Client*>()*;  
 private int simTime;  
 private Thread t = new Thread*(*this*)*;  
 private String results;  
 private FileWriter fileWriter;  
 int numberQueues;  
 int numberClients;  
 int BAT;  
 int EAT;  
 int BST;  
 int EST;  
  
 public String getResults*() {* return results;  
 *}* public Thread getThread*() {* return t;  
 *}* public Controller*(){* inputView.addButtonListener*(*new ButtonListener*())*;  
 results = "";  
 try *{* fileWriter = new FileWriter*(*"C:\\Users\\Mitroi Bianca\\OneDrive\\Desktop\\fisiere facultate anul 2 sem 2\\TPF\\salvez\_proiecte\_tp\_in\\Magazin\\src\\test\\results.txt", false*)*;  
  
 *}* catch *(*IOException e1*) {* // *TODO Auto-generated catch block* e1.printStackTrace*()*;  
 *}  
 }* public int getTime*(){* return simTime;  
 *}* public ArrayList*<*Server*>* getServers*(){* return servers;  
 *}* public ArrayList*<*Client*>* getClients*(){* return clients;  
 *}* public class ButtonListener implements ActionListener*{* public void actionPerformed*(*ActionEvent e*)  
 {* view.setClientLabel*(*clients.toString*())*;  
 inputView.getFrame*()*.setVisible*(*false*)*;  
 view.getFrame*()*.setVisible*(*true*)*;  
  
// try {  
// numberQueues = Integer.parseInt(inputView.getTextFieldText());  
// if(numberQueues <= 0 || numberQueues > 20) {  
// JOptionPane.showMessageDialog(view.getFrame(), "Number of queues is invalid!");  
// view.getFrame().setVisible(false);  
// inputView.getFrame().setVisible(true);  
// }  
// }catch(NumberFormatException ee) {  
// JOptionPane.showMessageDialog(view.getFrame(), "Please write a number of queues!");  
// view.getFrame().setVisible(false);  
// inputView.getFrame().setVisible(true);  
// }  
//  
// try {  
// numberClients = Integer.parseInt(inputView.getTextField\_1Text());  
// if(numberClients <= 0 || numberClients > 1000) {  
// JOptionPane.showMessageDialog(view.getFrame(), "Number of clients is invalid!");  
// view.getFrame().setVisible(false);  
// inputView.getFrame().setVisible(true);  
// }  
// }catch(NumberFormatException ee) {  
// JOptionPane.showMessageDialog(view.getFrame(), "Please write a number of clients!");  
// view.getFrame().setVisible(false);  
// inputView.getFrame().setVisible(true);  
// }  
//  
// try {  
// simTime = Integer.parseInt(inputView.getTextField\_2Text());  
// if(simTime <= 0 || simTime > 200) {  
// JOptionPane.showMessageDialog(view.getFrame(), "Simulation time is invalid!");  
// view.getFrame().setVisible(false);  
// inputView.getFrame().setVisible(true);  
// }  
// }catch(NumberFormatException ee) {  
// JOptionPane.showMessageDialog(view.getFrame(), "Please write a simulation time!");  
// view.getFrame().setVisible(false);  
// inputView.getFrame().setVisible(true);  
// }  
//  
// try {  
// BAT = Integer.parseInt(inputView.getTextField\_3Text());  
// if(BAT <= 0 || BAT > simTime) {  
// JOptionPane.showMessageDialog(view.getFrame(), "Smallest arrival time is invalid!");  
// view.getFrame().setVisible(false);  
// inputView.getFrame().setVisible(true);  
// }  
// }catch(NumberFormatException ee) {  
// JOptionPane.showMessageDialog(view.getFrame(), "Please write the smallest arrival time!");  
// view.getFrame().setVisible(false);  
// inputView.getFrame().setVisible(true);  
// }  
//  
// try {  
// EAT = Integer.parseInt(inputView.getTextField\_4Text());  
// if(EAT <= BAT || EAT > simTime) {  
// JOptionPane.showMessageDialog(view.getFrame(), "Largest arrival time is invalid!");  
// view.getFrame().setVisible(false);  
// inputView.getFrame().setVisible(true);  
// }  
// }catch(NumberFormatException ee) {  
// JOptionPane.showMessageDialog(view.getFrame(), "Please write the largest arrival time!");  
// view.getFrame().setVisible(false);  
// inputView.getFrame().setVisible(true);  
// }  
//  
// try {  
// BST = Integer.parseInt(inputView.getTextField\_5Text());  
// if(BST <= 0 || BST > simTime) {  
// JOptionPane.showMessageDialog(view.getFrame(), "Smallest service time is invalid!");  
// view.getFrame().setVisible(false);  
// inputView.getFrame().setVisible(true);  
// }  
// }catch(NumberFormatException ee) {  
// JOptionPane.showMessageDialog(view.getFrame(), "Please write the smallest service time!");  
// view.getFrame().setVisible(false);  
// inputView.getFrame().setVisible(true);  
// }  
//  
// try {  
// EST = Integer.parseInt(inputView.getTextField\_6Text());  
// if(EST <= 0 || EST > simTime) {  
// JOptionPane.showMessageDialog(view.getFrame(), "Largest service time is invalid!");  
// view.getFrame().setVisible(false);  
// inputView.getFrame().setVisible(true);  
// }  
// }catch(NumberFormatException ee) {  
// JOptionPane.showMessageDialog(view.getFrame(), "Please write the largest service time!");  
// view.getFrame().setVisible(false);  
// inputView.getFrame().setVisible(true);  
// }  
  
 simTime = 60;  
 numberQueues = 5;  
 numberClients = 50;  
 BAT = 2;  
 EAT = 40;  
 BST = 1;  
 EST = 7;  
  
 Random random = new Random*()*;  
  
 for*(*int i = 0; i < numberQueues; i++*)  
 {* Server s = new Server*()*;  
 servers.add*(*s*)*;  
 *}* for*(*int i = 0; i < numberClients; i++*)  
 {* int arrival = random.nextInt*()* % EAT;  
 while*(*arrival < BAT*)* arrival += BAT;  
 int service = random.nextInt*()* % EST;  
 while*(*service < BST*)* service += BST;  
 Client c = new Client*(*i, arrival, service*)*;  
 clients.add*(*c*)*;  
  
 *}* Collections.*sort(*clients*)*;  
 view.setClientLabel*(*clients.toString*())*;  
 t.start*()*;  
  
 *}  
 }* @Override  
 public void run*() {* int currentTime = 0;  
 int peakHourTime = 0;  
 int maxLength = 0;  
 int currentQueue = 0;  
 int minTime;  
 int i = 0;  
  
  
 while*(*currentTime <= simTime*) {* while*(*i < clients.size*()) {* while*(*clients.get*(*i*)*.getArrival*()* == currentTime + 1*) {* minTime = 1000;  
  
 for*(*int j = 0; j < servers.size*()*; j++*) {* if*(*servers.get*(*j*)*.getWaitingTime*()* < minTime*) {* minTime = servers.get*(*j*)*.getWaitingTime*()*;  
 currentQueue = j;  
 *}* if*(*servers.get*(*j*)*.getClients*()*.size*()* > maxLength*) {* maxLength = servers.get*(*j*)*.getClients*()*.size*()*;  
 peakHourTime = currentTime ;  
 *}  
 }* servers.get*(*currentQueue*)*.addClients*(*clients.get*(*i*))*;  
 view.setLblNewLabel*(*servers.get*(*currentQueue*)*.getClients*()*.toString*()*, currentQueue*)*;  
 Client cl = new Client*(*clients.get*(*i*)*.getId*()*, clients.get*(*i*)*.getArrival*()*, clients.get*(*i*)*.getService*())*;  
 served.add*(*cl*)*;  
 i++;  
 String c = "";  
 for*(*int k = i; k < clients.size*()*; k++*)* c += clients.get*(*k*)*.toString*()*;  
 view.setClientLabel*(*c*)*;  
  
 if*(*i == clients.size*())* break;  
 *}* String c = "";  
 for*(*int k = i; k < clients.size*()*; k++*)* c += clients.get*(*k*)*.toString*()*;  
 view.setClientLabel*(*c*)*;  
  
 results += "\nCurrent time: " + Integer.*toString(*currentTime*)* + "\n";  
  
 float averageWaitingTime = 0;  
 float serviceTime = 0;  
 for*(*int j = 0; j < servers.size*()*; j++*) {* averageWaitingTime += servers.get*(*j*)*.getWaitingTime*()*;  
 serviceTime += servers.get*(*j*)*.getClients*()*.peek*()* == null ? 0 : servers.get*(*j*)*.getClients*()*.peek*()*.getService*()*;  
 view.setLblNewLabel*(*servers.get*(*j*)*.getClients*()*.toString*()*, j*)*;  
 results += "Queue: " + j + " clients waiting: " + servers.get*(*j*)*.getClients*()*.toString*()* + "\n";  
 *}* averageWaitingTime /= servers.size*()*;  
 serviceTime /= servers.size*()*;  
 view.setAverageServiceLabel*(*Float.*toString(*serviceTime*))*;  
 view.setAverageWaitingLabel*(*Float.*toString(*averageWaitingTime*))*;  
 results += "Average waiting time in real time: " + Float.*toString(*averageWaitingTime*)*;  
// System.out.println(results);  
// try {  
// fileWriter.write(results);  
// } catch (IOException e) {  
// // *TODO Auto-generated catch block*// e.printStackTrace();  
// }  
 currentTime++;  
 view.setTimeLabel*(*Integer.*toString(*currentTime*))*;  
  
 try *{* Thread.*sleep(*1000*)*;  
 *}* catch *(*InterruptedException e*) {* // *TODO Auto-generated catch block* e.printStackTrace*()*;  
 *}  
  
 }* float averageWaitingTime = 0;  
 float serviceTime = 0;  
 int nullServers = 0;  
 results += "\nCurrent time: " + Integer.*toString(*currentTime*)* + "\n";  
 for*(*int j = 0; j < servers.size*()*; j++*) {* if*(*servers.get*(*j*)*.getClients*()*.peek*()* == null*)* nullServers++;  
 averageWaitingTime += servers.get*(*j*)*.getWaitingTime*()*;  
 serviceTime += servers.get*(*j*)*.getClients*()*.peek*()* == null ? 0 : servers.get*(*j*)*.getClients*()*.peek*()*.getService*()*;  
 results += "Queue: " + j + " clients waiting: " + servers.get*(*j*)*.getClients*()*.toString*()* + "\n";  
  
 view.setLblNewLabel*(*servers.get*(*j*)*.getClients*()*.toString*()*, j*)*;  
 *}* if*(*nullServers == servers.size*())* break;  
 averageWaitingTime /= servers.size*()*;  
 view.setAverageWaitingLabel*(*Float.*toString(*averageWaitingTime*))*;  
 view.setAverageServiceLabel*(*Float.*toString(*serviceTime*))*;  
 results += "Average waiting time in real time: " + Float.*toString(*averageWaitingTime*)*;  
// System.out.println(results);  
// try {  
// fileWriter.write(results);  
// } catch (IOException e) {  
// // *TODO Auto-generated catch block*// e.printStackTrace();  
// }  
  
 currentTime++;  
 view.setTimeLabel*(*Integer.*toString(*currentTime*))*;  
  
 try *{* Thread.*sleep(*1000*)*;  
 *}* catch *(*InterruptedException e*) {* // *TODO Auto-generated catch block* e.printStackTrace*()*;  
 *}  
  
  
 }* float service = 0;  
 for*(*int l = 0; l < served.size*()*; l++*)* service += served.get*(*l*)*.getService*()*;  
 service /= served.size*()*;  
 results += "\nAverage waiting time: " + service + "\n" + "Peak hour: " + peakHourTime;  
 System.*out*.println*(*results*)*;  
 try *{* fileWriter.write*(*results*)*;  
 *}* catch *(*IOException e*) {* // *TODO Auto-generated catch block* e.printStackTrace*()*;  
 *}* try *{* fileWriter.close*()*;  
 *}* catch *(*IOException e*) {* // *TODO Auto-generated catch block* e.printStackTrace*()*;  
 *}  
 }  
}*

# Rezultate

Cele 3 fisiere corespunzatoare pentru cele 3 cazuri de test specificate in cerinta temei contin rezultatele acestora, pentru fiecare moment de timp fiind afisate cozile, timpul mediu de asteptare, timpul mediu de serviciu, iar la finalul fisierului sunt specificate ora de varf si timpul mediu de asteptare pentru intraga simulare, adica pentru clientii care nu au fost asezati la cozi.

# Concluzii

*Sistemul implementat functioneaza conform tuturor specificatiilor mentionate in resursele puse la dispozitie*

# Bibliografie

https://dsrl.eu/courses/pt/