**DOCUMENTATIE TEMA 2**

**QUEUES SIMULATOR**

Cuprins:

1. Obiectivul temei
2. Analiza problemei
3. Proiectare
4. Implementare
5. Concluzii
6. Bibliografie

# Obiective

# Cerinte Functionale

Implemenatrea unui simulator pentru timpul de asteptarea si distibuirea clientilor la coada unui magazine. Se cunosc numarul de clienti si numarul de case deschise de care dispune magazinul. Fiecare client este distibuit la o casa goala daca acest lucru este posibil. Coada foloseste principiul FIFO (first in first out). Trebuie stiut un timp mediu de asteptare pentru fiecare client.

Programul genereaza clientii random si ii pune pe acestia intr-o coada. Se cunoast timpul in care clientii ajung la magazin.

## Obiectiv Principal:

(obiectivul principal al proiectului)

Obiectivul principal al acestui proiect este de a simula cat mai correct principiul cozii. Se doreste implementrarea cat mai eficienta a cozii magazinului, astfel incat timpul de asteptare si distribuirea clientilor sa fie cat mai bun pentru fiecare persoana.

Fiecare magazine dispune de un numar de case care sunt sau nu deschise, in functie de numarul de client care se afla in magazine.

## Obective Secundare:

( pasii pe care trebuie sa-i urmam pentru a atinge obiectivul principal )

|  |  |
| --- | --- |
| **Obiectiv Secundar** | **Descriere** |
| Dezvoltarea de use case-uri si scenarii | Într-un sistem software un ‘use-case’ este o listă de acțiuni sau pași de eveniment care definesc în mod obișnuit interacțiunile dintre un rol ( cunoscut ca un actor în limbajul unificat de modelare ( UML ) ) și un sistem in atingerea unui obiectiv.  In computere, un scenariu este o naratiune a interactiunilor dintre roluri ( actori in UML ) si sistem. In cazul nostru rolul - un utilizator al aplicatiei. |
| Alegerea structurilor de date | Structurile de date folosite pentru a duce la capat obiectivul principal |
| Impartirea pe clase | Folosirea unui MVC ( Model – View – Controller) pentru a putea forma un GUI ( Graphic User Interface ), a clientilor care intra in magazin. |
| Dezvoltarea algoritmilor | Vor fi descrise structurile de date necesare pentru atingerea obiectivului principal, schema UML ( diagrama de clase ) precum si algoritmii folositi pentru realizarea simularii. |
| Implementarea solutiei | Vor fi descrise fiecare clasa cu campurile si metodele importante precum si descrierea interfetei. |
| Testare | Vor fi descrise cateva scenarii de testare , folosind ca instrument un Junit Test. |

# Analiza problemei

**2.1. Thread-uri / Scenarii**

“Multithreading” înseamnă capacitatea unui program de a executa mai multe secvenţe de cod în acelaşi timp. O astfel de secvenţă de cod se numeşte fir de execuţie sau thread. Limbajul Java suportă multithreading prin clase disponibile în pachetul java.lang. În acest pachet există 2 clase Thread şi ThreadGroup, şi interfaţa Runnable.

Clasa Thread şi interfaţa Runnable oferă suport pentru lucrul cu thread-uri ca entităţi separate, iar clasa ThreadGroup pentru crearea unor grupuri de thread-uri în vederea tratării acestora într-un mod unitar. Există 2 metode pentru crearea unui fir de execuţie: se creează o clasă derivată din clasa Thread, sau se creează o clasă care implementeză interfaţa Runnable.

# Proiectare

( etapele de proiectare )

## Structuri de date

(structurile de date folosite in program)

Ca si structura de data principala pe care am folosit-o este un ArrayList pentru generarea clientilor (deoarece intr-un magazine intra mai mult client, si cred ca este cea mai ok metoda pentru generarea acestora intre arrival si arrivalMin si serviceMAX si serviceMin).

ArrayList<Client> generateRandom(**int** nrClienti, **int** arrivalMAX, **int** arrivalMin,**int** serviceMAX, **int** serviceMin, ArrayList<Client> waitingClients)

1. **Implemaentare**

Clasele care fac parte din acest proiect sunt : clasa Client, clasa Actiuni, clasa Interface, clasa Coada, clasa Task.

* **Client** : aici se stabileste timpul la care clientul intra in magazine si timpul de servire al acestuia.

**public class** Client **implements** Comparable<Client>{  
 **private int arrivalTime**;  
 **private int serviceTime**;  
  
 Client (**int** arr, **int** ser) {  
 setArrivalTime(arr);  
 setServiceTime(ser);  
 }  
  
 **public int** getArrivalTime() {  
 **return arrivalTime**;  
 }  
 **public void** setArrivalTime(**int** arrivalTime) {  
 **this**.**arrivalTime** = arrivalTime;  
 }  
 **public int** getServiceTime() {  
 **return serviceTime**;  
 }  
 **public void** setServiceTime(**int** serviceTime) {  
 **this**.**serviceTime** = serviceTime;  
 }  
 */\*\*  
 \* Metoda care verifica valoarea componentelor.  
 \*/* @Override  
 **public int** compareTo(Client arg) {  
 **if**(**arrivalTime** > arg.**arrivalTime**) **return** 1;  
 **else if** (**arrivalTime** == arg.**arrivalTime**) **return** 0;  
 **else return** -1;  
  
 }  
 */\*\*  
 \* Metoda folosita pentru reprezentarea unui client.  
 \*/* **public void** represent() {  
 System.***out***.println(**"arr = "** + getArrivalTime() + **" ser = "** + getServiceTime());  
 }  
}

* **Actiuni** : aceasta clasa cuprinde metoda de generare random a clientilor si asezarea acestora in coada, si numarul total al cozilor.

**public class** Actiuni {  
 **static int** *sumaMAX* = 0;  
 **static int** *secunda* = 0;  
 **static int** *minut* = 0;  
 **static int** *lastTime* = 0;  
 **static int** *nrAdaugari* = 0;  
 **static int** *sumaAverage* = 0;  
 */\*\*  
 \* Metoda ce genereaza random clienti intre arrivalMAX si arrivalMin si serviceMAX si serviceMin  
 \*/* **public** ArrayList<Client> generateRandom(**int** nrClienti,**int** arrivalMAX,**int** arrivalMin,**int** serviceMAX, **int** serviceMin, ArrayList<Client> waitingClients) {  
 Random random = **new** Random();  
  
 **while**(nrClienti > 0) {  
 **try**{  
 **int** randomArrival = random.nextInt((arrivalMAX - arrivalMin) + 1) + arrivalMin;  
 **int** randomService = random.nextInt((serviceMAX - serviceMin) + 1) + serviceMin;  
 waitingClients.add(**new** Client(randomArrival,randomService));  
 nrClienti--;  
 }**catch**(Exception E) {}  
 }  
 **return** waitingClients;  
  
 }

**public static void** sumaTotalaCozi(ArrayList<Coada> cozi,**int** timeS, **int** timeM) {  
 System.***out***.println(**"Entered suma totala cozi"**);  
  
 **int** suma = 0;  
 **for**(Coada a : cozi) {  
 suma = suma + a.getSuma();  
 }  
 **if**(suma > *sumaMAX*) {  
 *sumaMAX* = suma;  
 *secunda* = timeS;  
 *minut* = timeM;  
 }  
  
}

**public static void** incChanges(ArrayList<Coada> cozi, **int** secunde) {  
 *nrAdaugari*++;  
 System.***out***.println(**"La secunda "** + secunde + **" a avut loc adaugare"**);  
 **for**(Coada a : cozi) {  
 **if**(a.getSize() > 1)  
 *sumaAverage* = *sumaAverage* + a.getSuma() - a.getCurrent(); }  
}

* **Interface** ( am ales sa implementez o interfata deoarece la citirea din fisier nu imi afisa nimc in fisierul de iesire) aceasta metoda cuprinde efectiv functiile pentru simularea functionarii programului. Se deschide o fereastra in care trebuie intrudes numarul de cozi, numarul de client, timpul de sosire si timpul de servire. Apoi dup ace se introduce correct aceste date se deschide fereastra de simulare unde exista butonul stop pentru intreruperea acesteia. Datele de iesire vor fi vizibile intr-un document text. Clientii sunt pusi in coada, iar dupa acestia sunt transmisi catre thread-uri.

**public void** addComponentsToPane(Container pane) {  
 **try**{  
 **writer** = **new** PrintWriter(**"Output.txt"**, **"UTF-8"**);  
 }**catch**(Exception E) {System.***out***.println(**"File not found"**);}  
  
 */\*  
 \* Adaugam TextField pentru timer.  
 \*/* **time** = **new** JTextField(3);  
 **time**.setMargin(**new** Insets(0,0,0,0));  
  
 */\*  
 \* ArrayList-urile pentru a memora threadurile, clientii si restul datelor  
 \*/* **random** = **new** Random();  
 **threads** = **new** ArrayList<Task>();  
 **first** = **new** ArrayList<Boolean>();  
 **waitingClients** = **new** ArrayList<Client>();  
 **textFields** = **new** ArrayList<JTextField>();  
 **labels** = **new** ArrayList<JLabel>();  
  
 **if** (***RIGHT\_TO\_LEFT***) {  
 pane.setComponentOrientation(ComponentOrientation.***RIGHT\_TO\_LEFT***);  
 }  
 **if**(***shouldfill***){  
 **c**.**fill** = GridBagConstraints.***HORIZONTAL***;  
 }  
  
 pane.setLayout(**new** GridBagLayout());  
 */\*  
 \* Adaugam label pentru numarul de cozi.  
 \*  
 \*/* **lnrCozi** = **new** JLabel(**"Numarul de cozi:"**,JLabel.***LEFT***);  
 **c**.**gridx** = 0;  
 **c**.**gridy** = 0;  
 **c**.**fill** = GridBagConstraints.***LAST\_LINE\_END***;  
 **c**.**anchor** = GridBagConstraints.***FIRST\_LINE\_START***;  
 pane.add(**lnrCozi**, **c**);  
  
 */\*  
 \* Adaugam TextField pentru numarul de cozi.  
 \*  
 \*/* **nrCozi** = **new** JTextField(3);  
 **c**.**gridx**++;  
 **c**.**anchor** = GridBagConstraints.***NORTH***;  
 pane.add(**nrCozi**, **c**);  
  
 */\*  
 \* Label si TextField pentru numaurul de clienti.  
 \*/* **lnrClienti** = **new** JLabel(**"Numarul de clienti:"**, JLabel.***LEFT***);  
 **c**.**gridx** = 0;  
 **c**.**gridy** = 1;  
 **c**.**fill** = GridBagConstraints.***LAST\_LINE\_END***;  
 pane.add(**lnrClienti**, **c**);  
  
 **nrClientiText** = **new** JTextField(3);  
 **c**.**gridx**++;  
 **c**.**anchor** = GridBagConstraints.***NORTH***;  
 pane.add(**nrClientiText**, **c**);  
  
 */\*  
 \* Label si TextField pentru campurile de introducere a intervalului pentru Arrival time.  
 \*/* **larrivalTime** = **new** JLabel(**"Arrival time"**);  
 **c**.**gridx** = 0;  
 **c**.**gridy** = 2;  
 pane.add(**larrivalTime**, **c**);  
  
 **arrivalTimeMin** = **new** JTextField(3);  
 **c**.**gridx**++;  
 pane.add(**arrivalTimeMin**, **c**);  
  
 **arrivalTimeMAX** = **new** JTextField(3);  
 **c**.**gridx**++;  
 pane.add(**arrivalTimeMAX**, **c**);  
  
 */\*  
 \* Label si TextField pentru campurile de introducere a intervalului pentru Arrival time.  
 \*/* **lserviceTime** = **new** JLabel(**"Service time"**);  
 **c**.**gridy**++;  
 **c**.**gridx** = 0;  
 pane.add(**lserviceTime**, **c**);  
  
 **serviceTimeMin** = **new** JTextField(3);  
 **c**.**gridx**++;  
 pane.add(**serviceTimeMin**, **c**);  
  
 **serviceTimeMAX** = **new** JTextField(3);  
 **c**.**gridx**++;  
 pane.add(**serviceTimeMAX**, **c**);  
  
 */\*  
 \* Buton pentru simulare care contine un ActionListener.  
 \*/* **ok** = **new** JButton(**"Simulare"**);  
 **ok**.setFont(**ok**.getFont().deriveFont(19.0f));  
 **ok**.setBackground(**new** Color(209,132,160));  
 **ok**.setMargin(**new** Insets(0,0,0,0));  
 **c**.**gridx** = 5;  
 **c**.**gridy** = 4;  
 ActionListener action;

**int** arrivalMin = Integer.*parseInt*(**arrivalTimeMin**.getText());  
**int** arrivalMAX = Integer.*parseInt*(**arrivalTimeMAX**.getText());  
**int** serviceMin = Integer.*parseInt*(**serviceTimeMin**.getText());  
**int** serviceMAX = Integer.*parseInt*(**serviceTimeMAX**.getText());  
**nrClienti** = Integer.*parseInt*(**nrClientiText**.getText());  
  
**if**(arrivalMAX < arrivalMin) {  
 popUp(**mesajOrdineIncorecta**, **true**);  
 resetAll();  
}  
**if**(serviceMAX < serviceMin) {  
 popUp(**mesajOrdineIncorecta**, **true**);  
 resetAll();  
}  
  
*//Generare random a clientilor***waitingClients** = **actiune**.generateRandom(**nrClienti**, arrivalMAX, arrivalMin, serviceMAX, serviceMin, **waitingClients**);  
  
*//Sortam lista*Collections.*sort*(**waitingClients**);  
  
*//reprezentare***writer**.println(**"Arrival Time Service Time"**);  
**for**(Client a : **waitingClients**) {  
 a.represent();  
 **writer**.println(**" "** + a.getArrivalTime() + **" "** + a.getServiceTime());  
}  
**writer**.println();  
pane.add(Box.*createRigidArea*(**new** Dimension(30,40)));  
  
*//adaugare semi-dinamica a textfieldurile cu text***cozi** = **new** ArrayList<Coada>();  
**textToDisplay** = **new** ArrayList<String>();  
*//Parcurgem numarul de cozi si adaugam componente necesare in ArrayList-s***for**(**int** j = 1; j <= **numarCozi**; j++) {  
 **boolean** prim = **false**;  
 **first**.add(prim);  
 **emp**.add(**false**);  
 **casa** = **new** JTextField(10);  
 **cozi**.add(**new** Coada());  
 **cozi**.get(j - 1).setIndex(j-1);  
  
 **textFields**.add(**casa**);  
 **casa**.setEditable(**false**);  
 JLabel server = **new** JLabel(**"Server "**+j);  
 **labels**.add(server);  
  
 *//Adaugam in panel labelurile* **c**.**gridy**++;  
 **c**.**gridx** = 0;  
  
 pane.add(server, **c**);  
 **c**.**gridx**++;  
 pane.add(**casa**, **c**);  
  
 **textToDisplay**.add(**""**);  
}  
*/\*  
 \* Verificam daca numarul de clienti este mai mic decat  
 \* numarul de cozi pentru a putea afisa corect dinamic.  
 \*/***if**(**nrClienti** < **numarCozi**) {  
 **for**(**int** j = 0; j < **numarCozi** - **nrClienti**; j++) {  
 **emp**.set(j, **true**);  
 }  
}  
  
*/\*  
 \* Adaugam butonul de stop si ii adugam un ActionListener  
 \*/***stop** = **new** JButton(**"STOP"**);  
**c**.**gridy**++;  
**c**.**gridx** = 4;  
**stop**.setFont(**stop**.getFont().deriveFont(19.0f));  
**stop**.setBackground(**new** Color(209,132,160));  
**stop**.setMargin(**new** Insets(0,0,0,0));  
pane.add(**stop**, **c**);  
**stop**.addActionListener(**new** ActionListener() {  
  
 @Override  
 **public void** actionPerformed(ActionEvent e) {  
 System.***out***.println(**"Entered reset"**);  
 resetAll();  
  
  
 }

*//adaugam butonul de START*pane.add(**ok**, **c**);  
  
**c**.**gridx**= 0;  
**c**.**gridy**++;  
**time**.setEditable(**false**);  
pane.add(**time**, **c**);  
System.***out***.println(**"Start = "** + **start**);  
*/\*\*  
 \* Action Listener-ul necesar timerului.  
 \*/*ActionListener ceas = **new** ActionListener() {  
  
 @Override  
 **public void** actionPerformed(ActionEvent arg0) {  
 System.***out***.println(**"Entered ActionPerformed from clock"**);  
  
 **if** (**start**) {  
 refreshGUI();  
  
 **if**(**sec** == 60) {  
 **clock**.restart();  
 **sec** = 1;  
 **minute**++;  
  
 }**else sec**++;

*/\*\*  
 \* Afisam intr-un fisier de iesire starea cozilor si timpul.  
 \*/***try**{  
 **if**(!**cozi**.isEmpty()) **writer**.println(**"Time ="** + **sec**);  
 **for**(**int** i = 0; i < **nrClienti**; i++) {  
  
 **writer**.println(**cozi**.get(i).toString());  
  
 }  
}**catch**(Exception e) {}

*/\*\*  
 \* Metoda care citeste elemente din cozi si le transmite threadurilor.  
 \*/***private void** changeData() {  
 System.***out***.println(**"Entered Change Data"**);  
 **int** j = 0;  
 System.***out***.println(**"clienti ramasi = "** + **waitingClients**.size() + **" cozi ramase = "** + **cozi**.size());  
  
 **while**(j < **cozi**.size()) {  
 **if**(**waitingClients**.isEmpty() && **cozi**.isEmpty()) {  
 System.***out***.println(**"entered mesajFinal"**);  
 }  
 System.***out***.println();  
 **try**{  
 **if**(**cozi**.get(j).elemLeft() <=0 && **waitingClients**.size() > 0)  
 **first**.set(j, **false**);  
 }**catch**(Exception E) {};  
  
 **if**(!**first**.get(j)) {  
 **try**{  
 **int** curent = **cozi**.get(j).getCurrent();  
 **threads**.get(j).setCurrent(curent);  
 }**catch**(Exception E){j++;  
 **continue**;  
 }  
  
 **first**.set(j, **true**);  
  
 }  
 **else** {  
 **try**{  
 **if**(**threads**.get(j).getCurrent() <= 0){  
 **cozi**.get(j).removeClient();  
 System.***out***.println(**"A eliminat un client"**);  
  
 **if**(**cozi**.get(j).elemLeft() > 0) {  
  
 **threads**.get(j).setCurrent(**cozi**.get(j).getCurrent());  
  
 }  
 **else** {  
 **if**(**waitingClients**.isEmpty())  
 **cozi**.get(j).removeClient();  
  
 **else continue**;  
 }  
 }  
  
 **else** {  
 **int** curent = **threads**.get(j).getCurrent();  
 **cozi**.get(j).updateCurrent(curent);  
 }  
 }**catch**(Exception E) {  
 **emp**.set(j, **true**);  
  
 **if**(**actiune**.checkIfEmpty(**emp**)) {  
 popUp(**mesajSfarsitSimulare**, **false**);  
 j = **nrClienti** + 1;  
 resetAll();  
  
  
 }  
 **else** j++;  
 System.***out***.println(**"Entered big catch"**);  
 **continue**;}  
  
 }  
 j++;  
 }  
 refreshGUI();  
}

*/\*\*  
 \* Metoda ce genereaza GUI  
 \*/***private void** createAndShowGUI() {  
 JFrame frame = **new** JFrame(**"Simulare"**);  
 frame.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);  
 addComponentsToPane(frame.getContentPane());  
 frame.setPreferredSize(**new** Dimension(600, 700));  
 frame.pack();  
 frame.setLocationRelativeTo(**null**);  
 frame.setVisible(**true**);  
  
  
}  
*/\*\*  
 \* Metoda care reseteaza toate campurile.  
  
 \*/***private void** resetAll() {  
 **try** {  
 System.***out***.println(**"Entered resetAll"**);  
  
 **first**.clear();  
 **sec**= 0;  
 **start** = **false**;  
 **waitingClients**.clear();  
 **serving** = **false**;  
 **set** = **false**;  
 **minute** = 0;  
 **numarCozi** = 0;  
 **emp**.clear();  
 **for**(JTextField a : **textFields**)  
 a.setVisible(**false**);  
 **for**(JLabel a : **labels**)  
 a.setVisible(**false**);  
 **textFields**.clear();  
 **labels**.clear();  
 **larrivalTime**.setVisible(**true**);  
 **lserviceTime**.setVisible(**true**);  
 **arrivalTimeMAX**.setVisible(**true**);  
 **arrivalTimeMin**.setVisible(**true**);  
 **serviceTimeMAX**.setVisible(**true**);  
 **serviceTimeMin**.setVisible(**true**);  
 **arrivalTimeMAX**.setText(**""**);  
 **arrivalTimeMin**.setText(**""**);  
 **serviceTimeMAX**.setText(**""**);  
 **serviceTimeMin**.setText(**""**);  
 **lnrCozi**.setVisible(**true**);  
 **nrCozi**.setVisible(**true**);  
 **stop**.setVisible(**false**);  
 **ok**.setVisible(**true**);  
 **simulating**.setVisible(**false**);  
 **nrCozi**.setText(**""**);  
 **time**.setText(**""**);  
 **clock**.stop();  
 **writer**.close();  
 System.*exit*(1);  
 } **catch** (Exception e) {  
 System.***out***.println(**"Entered Exception in resetAll"**);  
 e.printStackTrace();  
 }  
  
}

* **Coada** : este clasa care modeleaza comportarea unei cozi, unde se pun clientii in coada si se seteaza timpul de servire al acestora.

**public class** Coada {  
 **private int cur1**;  
 **private** ArrayList<Client> **clienti** = **new** ArrayList<Client>();  
 **private int index**;  
  
 **public int** getCurrent() {  
 **return clienti**.get(0).getServiceTime();  
 }  
  
 **public void** updateCurrent(**int** cur2) {  
 **clienti**.get(0).setServiceTime(cur2);  
  
  
 }  
  
 **public void** setCurrent(**int** current) {  
 **this**.**cur1** = current;  
 }  
  
  
 **public void** addClient(Client c) {  
 **clienti**.add(c);  
 }  
  
 **public** Client getClient(**int** index) {  
 **return clienti**.get(index);  
  
 }  
  
 **public int** getSize() {  
 **return clienti**.size();  
 }  
  
 **public int** getSuma(){  
 **int** sumaServiceTime = 0;  
 **if**(**clienti**.isEmpty()) **return** 0;  
 **else** {  
 **for**(Client a : **clienti**) {  
 sumaServiceTime = sumaServiceTime + a.getServiceTime();  
 }  
 **return** sumaServiceTime;  
 }  
 }  
  
 **public** String toString() {  
 **if**(**clienti**.isEmpty()) **return "0"**;  
 String string = **""**;  
 **for**(Client a : **clienti**) {  
 string = string + **" "** + a.getServiceTime();  
 }  
 **return** string;  
 }  
  
 **public void** setIndex(**int** i) {  
 **this**.**index** = i;  
 }  
  
 **public int** getIndex() {  
 **return index**;  
 }  
  
 **public void** removeClient() {  
 **clienti**.remove(0);  
 }  
 **public int** elemLeft() {  
 **return clienti**.size();  
 }  
  
}

* **Task** : aceasta clasa implementeaza modelarea thread-urilor

**public class** Task **extends** Thread {  
 **private int threadIndex**;  
 **private int current** = 0;  
 **private boolean goThread** = **false**;  
  
 Task(**int** threadIndex) {  
  
 **this**.**threadIndex** = threadIndex;  
 }  
 **public void** run() {  
  
 **while**(**true**) {  
 **if**(**goThread**)  
 **current**--;  
  
 **try** {  
 Thread.*sleep*(1000);  
 } **catch** (InterruptedException e) {  
 System.***out***.println(**"Exceptie in thread"**);  
 }  
  
 }  
 }  
  
 **public int** getThreadIndex() {  
 **return threadIndex**;  
 }  
  
 **public void** setCurrent(**int** curent) {  
 **this**.**current** = curent;  
 **if**(curent != 0) **goThread** = **true**;  
 }  
 **public int** getCurrent() {  
 **return current**;  
 }  
}

## 

1. **Concluzii**

Cu ajutorul acestui proiect am reusit sa dezvolt o aplicatie care utilizeaza thread-uti, pana acum nu am folosit foarte des acest lucru in programele dezvoltate anterior. Am incercat sa implementez cat mai corect operatiile cerute in ducumentatie acestui proiect, desi la unele am avut probleme, de exemplu cu citirea datelor dintr-un fisier si afisarea acestora in alt fisier. Nu se afisa nimic in fiserul de iesire, si din aceasta cauza am folosit interfata grafica. De aceea cred ca tema realizata de mine mai are nevoie de imbunatatiri pentru o dezvoltare ulterioara cat mai corecta si simplista. Sper ca proiectul sa fie usor de inteles pentru utilizatori, problemele pe care le rezolva fiind de nivel mediu. Am incercat sa folosesc o gandire cat mai realista sis a ma gandesc la exemplul din viata reala.

1. **Bibliografie**

Resursle care m-au ajutat in dezvoltarea acestui priect sunt:

<https://ro.wikipedia.org/wiki/Fir_de_execu%C8%9Bie>

<http://coned.utcluj.ro/~salomie/PT_Lic/4_Lab/Assignment_2/Java_Concurrency.pdf>