Tema 2

I. INTRODUCERE

În acească temă se va implementa simularea unor cozi de servire a clienților. Scopul este de a observa comportamentul unui număr diferit de cozi supus unor condiţii distincte. Se vor realiza statistici relevante cu scopul găsirii celei mai bune combinații de servire a clienților.

Se vor analiza aspecte precum:

1. Interfața grafică;

2. Reprezentarea unue cozi de clienți în java;

3. Aspecte legate de threaduri pentru paralelizarea funcțiilor;

4. Sincronizare între funcționalotățile proiectului;

5. Concluzie şi analiza a rezolvării problemei;

II. ANALIZA PROBLEIMEI ȘI A CAZURILOR POSIBILE

Textul problemei: „Design and implement a simulation application aiming to analyze queuing based systems for determining and minimizing clients' waiting time. “

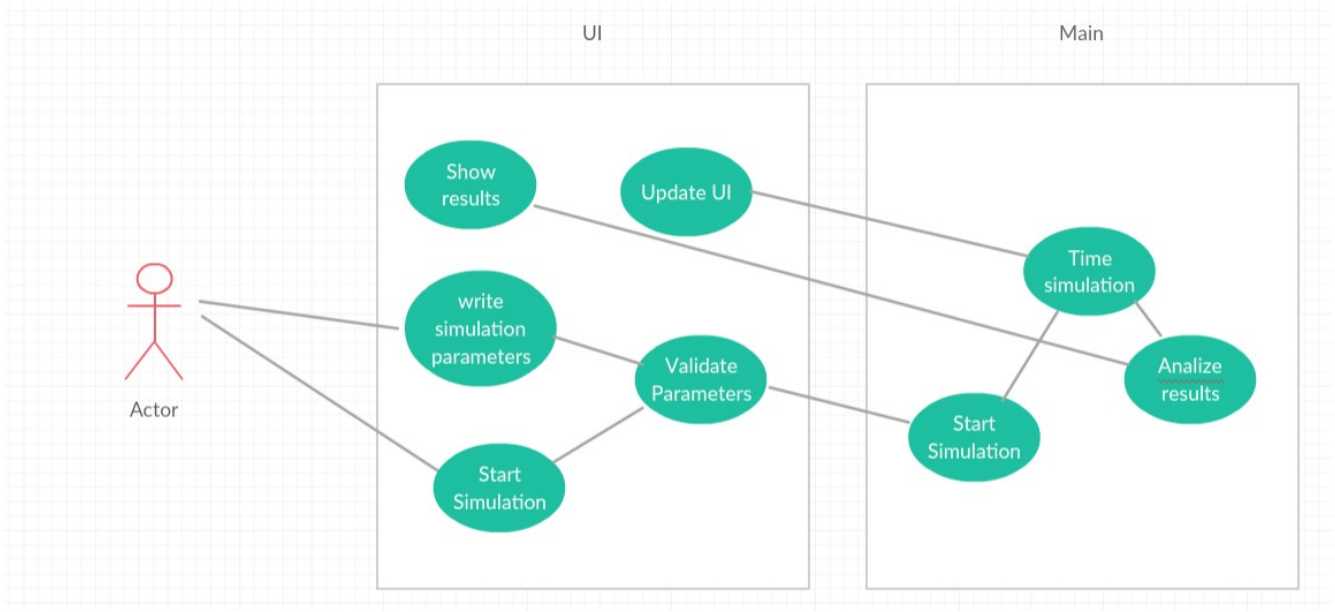
Input data:
- Minimum and maximum interval of arriving time between customers;
- Minimum and maximum service time;
- Number of queues; - Simulation interval;
- Other information you may consider necessary;

Minimal output:
- The average of waiting time, service time and empty queue time for 1, 2 and 3 queues for the simulation interval and for a specified interval (other useful information may be also considered);
- Log of events and main system data;
- Queue evolution;
- Peak hour for the simulation interval;

Pentru a realiza această aplicație va fi nevoie de o înțelegere a conceptelor legate de cozi(FIFO) și a nevoilor ce pot exista legate de acestea.

În principal, operațiile pe cozi sunt enqueue, și dequeue ce vor fi utilizare la intrarea unui client în unitate respectiv la plecarea lui.

Use-case



Utilizatorul nu are alte posibilități decât să introducă date și să ceară rezultate, astfel că erorile pot interveni doar la furnizarea greșită a datelor. Utilizatorul are de completat numărul de cozi, timpul simulat, viteza de simulare, timpul minim și maxim de sosire al clienților dar și minimul și maximul volumului de servicii dorite de clienți.

Exemplu : Din versiunea naturala(3x^3 + 2x^2 – 3x + 6) => 3*x^3 +2*x^2 -3*x^1 +6*x^0

III. PROIECTARE

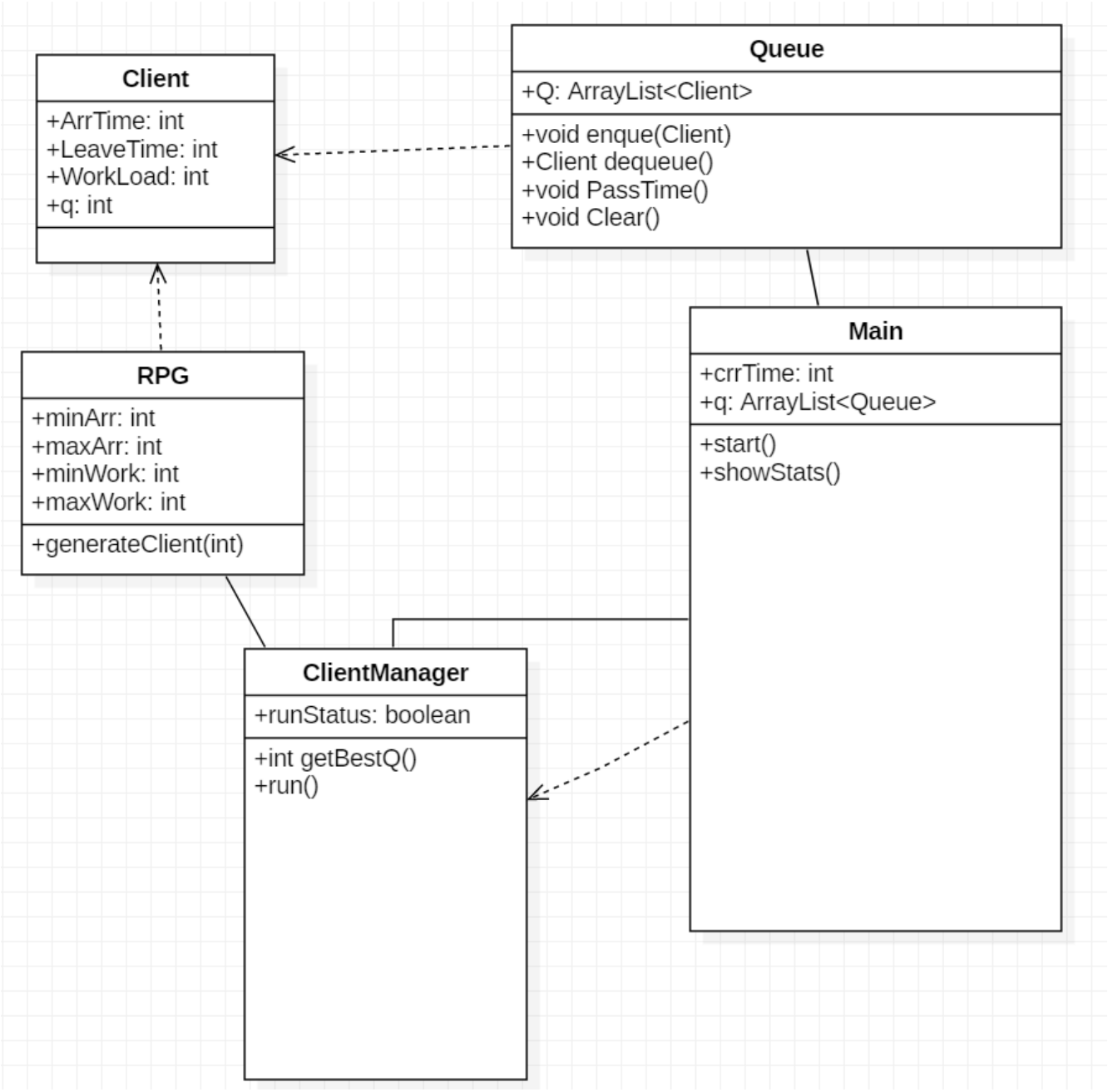
Pentru proiectarea aplicației va fi nevoie de implementarea a doua clase importante, anume clasa Client și clasa Queue(coadă). Aceste clase vor servi la stocare clienților veniți dar și la furnizarea de date necesare realizării statisticilor.

O altă clasă esențială aplicației este generatorul aleator de persoane(RPG). Acesta are rolul de a furniza, la intervale aleatoare de timp, persoane cu volume aleatoare de nevoi ce vor popula cozile unită ii.

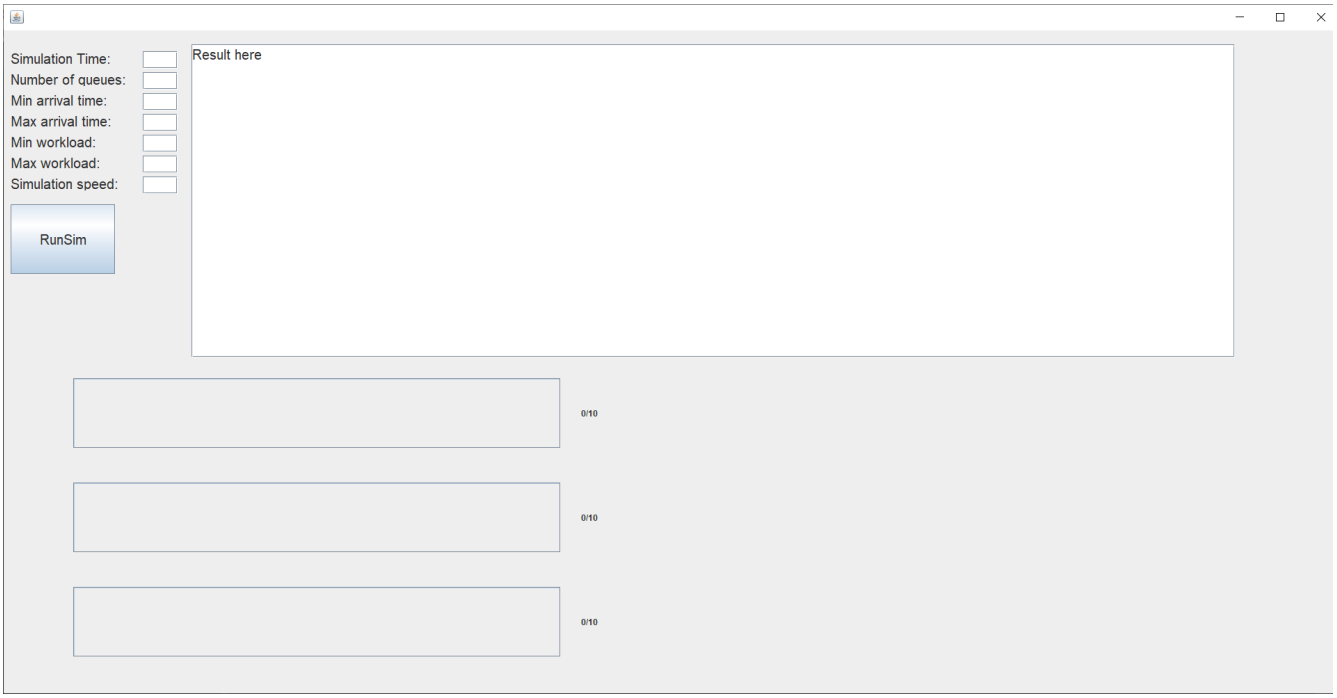
Pentru paralelizarea funcționalităților am utilizat threaduri și SwingWorker pentru sincronizarea UI cu datele oferite de aplicație. Threadul de lucru principal se află în clasa Main și dă naștere ulterior diferitelor threaduri în funcție de nevoi.

Pe lângă aceste clase simple va exista și clasa ClientManager care are scopul de a utiliza RPG-ul și de a gestiona intrarea și ieșirea din coadă a clieților.

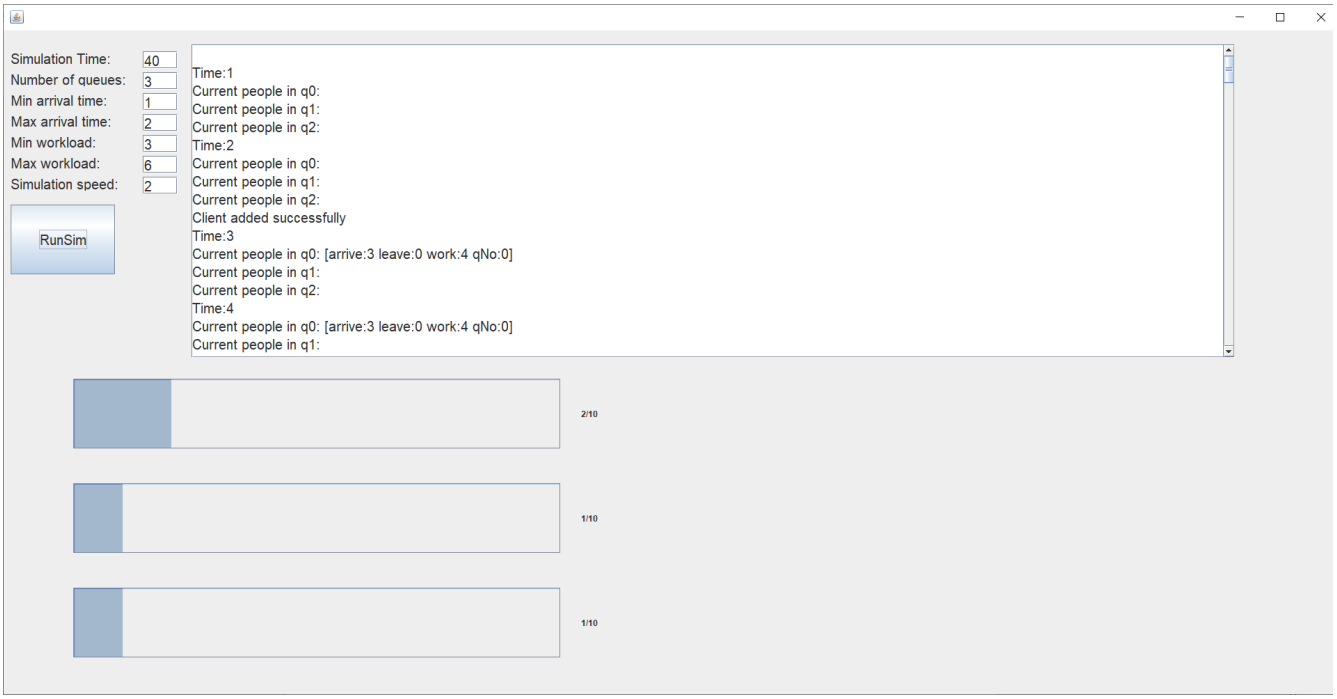
Aici am realizat diagrama UML inițială a programului. Clasa Queue va conține o listă de clienți și metode pentru gestionarea acestor, dar și a cozii în sine. Clasa Client conține toate informațiile ce definsc un client, și toate operațiile ce pot interveni asupra acestora.



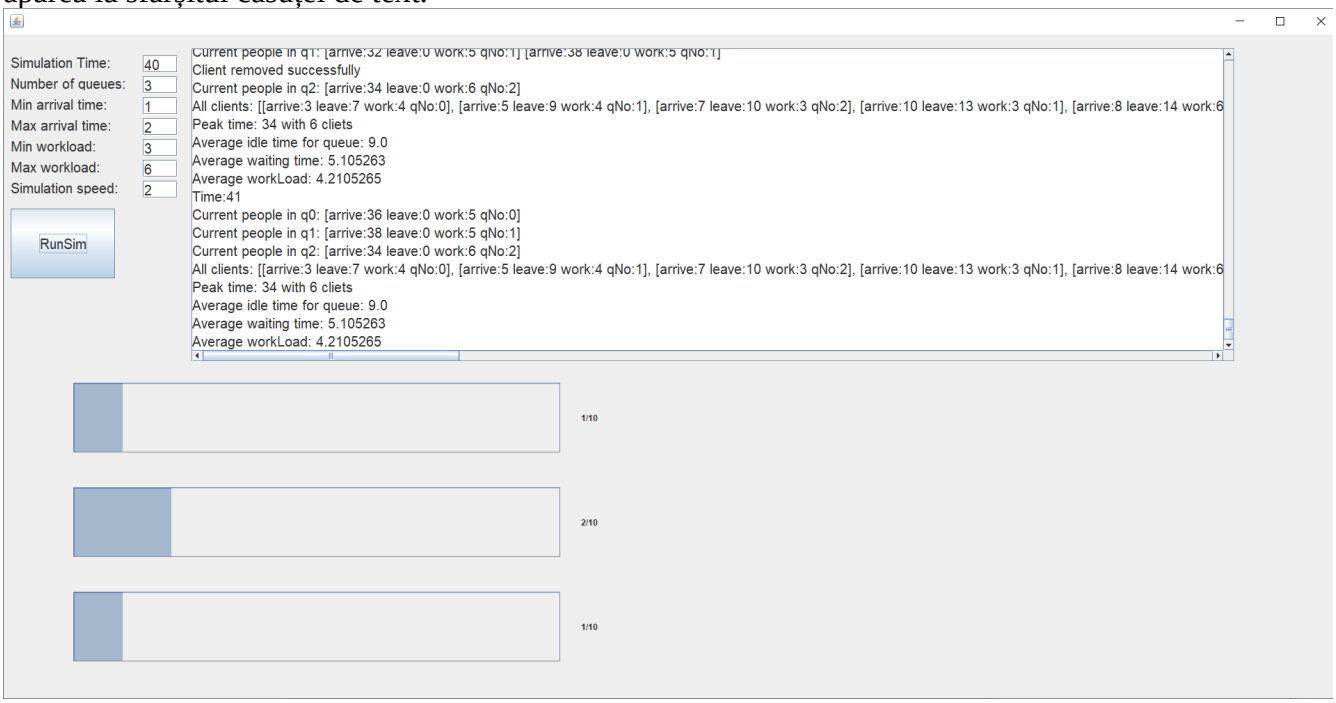
Pentru interfața grafică o să abordez un design simplu, punând accentul pe utilitate și funcționalitate.



Astfel UI-ul prezită un singur buton și 7 casete de text unde utilizator va configura după bunul plac simularea. După configurare se apasă butonul RunSim care va porni efectiv simularea cu datele furnizate.



Pe parcursul simulării, mesajele programului şi parcursul acestuia se vor afişa într-un al 8-lea TextBox. Evoluţia cozilor se va reprezenta grafic cu ajutorul ProgressBar-urilor. În final, statisticile vor apărea la sfârşitul căsuţei de text.



III. IMPLEMENTARE

Clasa Client conţine momentul venirii(ArrTime), timpul plecării(LeaveTime), cantitatea de servicii(WorkLoad), cantitatea de muncă rămasă(crrWork) dar şi coada la care a stat(q).

Această clasă este de tipul comparable pentru a ordona mai uşor în liste clienţii. Clasa utilă sintezei datelor este getWaitTime(), ce ne oferă diferenţa dintre timpul plecării şi timpul sosirii.

```
public int getWaitTime() throws ClientInQException
{
	if(leaveTime != 0)
		return leaveTime - arrTime;
	else throw(new ClientInQException());
```

Clasa Queue reprezintă o coadă de aşteptare efectivă ce funcţionează pe principiul FIFO(First în first out). Astfel, mereu vor pleca clienţi din capul cozii şi vor veni în coada ei.

```
public void enq(Client c)
{
	Q.add(Q.size(), c);

}

public Client deque()
{
	if(Q.size() != 0)
	{
		Q.get(0).setLeaveTime(Main.crrTime);
		Main.allClients.add(Q.get(0));
		System.out.println("Client removed successfully");
```

```
                Screen.resText.setText(Screen.resText.getText() + "\n" + "Client
                                                                removed successfully");
                return Q.remove(0);
        }

        else
                return null;
```

Pe lângă aceste funcționalități, clasa Queue are o metodă ce simulează trecerea timpului în cadrul cozii. Această metoda, passTime(), are grijă să scadă volumul curent de muncă al clientului din capul listei și în cazul finalizării serviciului să scoată clientul din coadă. De asemenea, pentru statistică, această metodă verifică la fiecare apel starea cozii. În cazul în care coada este goală, variabila idelTime este incrementată pentru ulterioarele calcule statistice.

```
public void passTime()
{
        if(Q.size() == 0)
        {
                idleTime++;
        }
        if(Q.size() > 0)
        {

                if(Q.get(0).getCrrWork() == 0)
                {
                        deque();
                }
                else
                {
                        Q.get(0).setCrrWork(Q.get(0).getCrrWork() - 1);
                }

        }
}
```

Clasa Screen are în ea toate elementele grafice ale programului și butonul ce deține cotrolul asupra începerii simularii. Această are funcția start() ce inițializează toate componentele grafice și este pornită de către funcția main din clasa Main.

Clasa ClientManager are rolul de a furniza treptat dar aleator câte un client cozilor implicate în simulare, dar și de a amplasa eficient în cozi respectivii clienți. Această clasă reprezintă un thread sincron cu trecerea timpului din clasa Main și acționează doar la intervale bine definite de timp. Sincronizarea este dată de logica thread-ului dar și de utilizarea specificatorilor synchronized asupra metodelor care necesită acces multiplu din diferite thread-uri.

Clasa Main este pricipalul thread al programului și simulează trecerea timpului cu ajutorul funcției sleep(). La fiecare unitate de timp trecută, acest thread cere celorlalte threduri o actualizare a datelor. Astfel se actualizează atât interfața grafică o dată cu trecerea timpului dar și cozile efective. La finalul execuției clasa Main invocă metoda showStats() care acum, cu toate datele la îndemână, poate oferi interfeței grafice statisticile obținute.

```
public static void showStats()
{
        float idleAv = 0;
        float waitAv = 0;
```

```
float workAv = 0;
for(int i = 0 ; i < noQ; i++)
{
	idleAv += q.get(0).getIdleTime();
}
System.out.println("Average idle time for queue: " + idleAv/noQ);
s.resText.setText(s.resText.getText() + "\n" + "Average idle time for queue:
                                                    " + idleAv/noQ);
for(Client c: allClients)
{
	try {
		waitAv += c.getWaitTime();
	} catch (ClientInQException e) {
		e.printStackTrace();
	}
	workAv += c.getWorkLoad();
}
System.out.println("Average waiting time: " + waitAv/allClients.size());
s.resText.setText(s.resText.getText() + "\n" + "Average waiting time: " +
                                                waitAv/allClients.size());
System.out.println("Average workLoad: " + workAv/allClients.size());
s.resText.setText(s.resText.getText() + "\n" + "Average workLoad: " + workAv/
                                                    allClients.size());
```

VI. CONCLUZII

După finalizarea proiectul am realizat importanța unei planificări solide, înainte de a începe efectiv scrierea codului, am învățat importanța utilizării thread-urilor pentru paralelizarea acțiunilor dar și modul de funcționare al acestora.

Pentru o dezvoltare ulteroioară, putem vorbi o intrefață grafică mai interactivă, un număr nedefinit de cozi dar și de algoritmi de eficacizare a programului. Pe lângă toate acestea ar putea fi adăugate diferite statistici suplimentare, cum ar fi statistici legate de fiecare coadă, de clienți dar și introducerea unor „ore de vârf” în datele furnizate de utilizator.

VII. BIBLIOGRAFIE

https://stackoverflow.com/questions/5453925/newline-n-in-textfield-java
https://stackoverflow.com/questions/12715432/cannot-update-swing-component-under-a-heavy-process
https://alvinalexander.com/java/java-swingutilities-invoke-later-example-edt
https://coderanch.com/t/534847/java/JProgressBar-doesn-show-progress
https://creately.com/app