



# **SIMULATOR DE COZI**

## **-Documentație-**

**Student:**

Pop Ruxandra Maria

**Univeristatea Tehnică din Cluj-Napoca  
Facultatea de Automatică și Calculatoare  
Secția Calculatoare și Tehnologia Informației  
Anul 2, Grupa 30226**

# Cuprins

1.Obiectivul temei .....	3
2.Analiza problemei .....	4
3.Proiectare .....	8
4.Implementare .....	13
5.Rezultate .....	18
6.Concluzii .....	20
7.Bibliografie .....	20

# 1.Obiectivul temei

## 1.1.Obiectiv principal

Obiectivul principal al acestei teme de laborator a fost să propunem, să proiectăm și să implementăm un simulator de cozi, care să determine și să minimizeze timpul de așteptare a clienților. Acest simulator primește ca date de intrare, numărul de clienți, numărul de cozi, timpul maxim de simulare, iar pentru clienți se specifică și timpul minim/maxim de așteptare, precum și timpul minim/maxim de servire. Proiectul trebuie să genereze, astfel clienți cu date aleatorii care să respecte aceste intervale de timp. La început toate cozile sunt închise, fiind deschise atunci când este adăugat primul client, și se închid din nou când pleacă ultimul client din ea. A trebuit implementată o strategie prin care fiecare client să fie direcționat spre coada cu cel mai mic număr de clienți/cu cel mai mic timp de așteptare. Fiecare coadă are un thread propriu asociat și funcționează independent de celelalte.

## 1.2.Obiective secundare

**Reprezintă pașii care trebuie urmați pentru atingerea obiectivului principal.**

Obiectiv Secundar	Descriere	Capitol
<b>Alegerea structurilor de date</b>	Am folosit multiple structuri de date, în special cele de tip thread safe.	3
<b>Impărțirea pe clase</b>	Am împărțit programul în diferite clase, ușor de înțeles, și implementate după modelul din prezentarea suport.	4
<b>Implementarea soluției</b>	Am folosit paradigma OOP, cu multiple clase și metode, care vor fi prezentate mai jos.	4
<b>Testarea</b>	Am introdus multiple date de intrare, încercând să acopăr cât mai multe cazuri posibile. Totodată am realizat și o testare cu JUnit.	5

## 2. Analiza problemei

Pentru a înțelege problema și scopul ei, este necesar să definim câteva aspecte teoretice importante legate de cozi și cozi.

**Cozile sunt utilizate pentru modelarea domeniilor din lumea reală.**

Obiectivul principal al unei cozi este de a oferi un loc pentru ca un *client* să aștepte înainte de a primi un *serviciu*. Fiecare coadă are clienți care așteaptă să fie serviți/procesați. Acest client poate alege la ce coadă să stea în funcție de timpul de așteptare al cozi, va alege mereu coada cu timpul de așteptare cel mai mic.

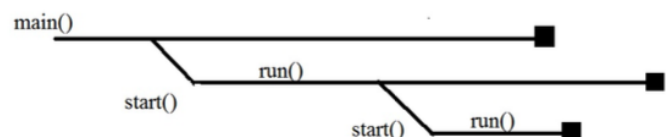
**Thread reprezintă o succesiune secvențială de instrucțiuni care sunt executate în cadrul unui proces.**

Crearea unui nou fir de execuție necesită mai puține resurse decât crearea unui nou proces. Există în cadrul unui proces, adică fiecare proces are cel puțin un thread. Fiecare thread este asociat cu o instanță a clasei Thread.

**Pentru a defini un thread:**

- Extend Thread
- Implement Runnable

Fiecare fir suprascrie metoda `run()`  
=> aceasta va fi executată în timpul ciclului de viață al firului



Aplicația prezintă următoarele :

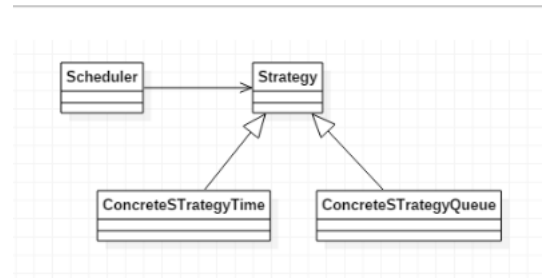
- Există un timp de simulare care începe de la 0 și se continuă până la timpul maxim definit de către noi, incrementarea lui marchează trecerea timpului.
- Task-urile reprezintă clienții, și sunt generate în mod random de către aplicație, fiecare task este caracterizat de:

- un id
- de arrive time, care ia valori între arrive time maxim/minim definite de către noi
- de service time, care ia valori între service time maxim/minim definite de către

noi

-Server care sunt cozile, servește fiecare client în funcție de ordinea în care au fost adăugați. Adăugarea lor se face atunci când timpul de așteptare a clientului este egal cu timpul de simulare la momentul actual. Când se termină timpul de servire/procesare a clientului, acesta este scos din coadă, și serverul se închide temporar.

-Scheduler trimite task-uri către servere în conformitate cu strategia stabilită de către noi. Folosește un Strategy Patter.



-Aplicația se sfârșește atunci când nu mai sunt clienți de procesat sau când sa terminat timpul maxim de simulare definit de către noi.

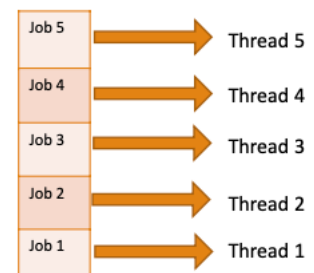
-Aplicația prezintă și o interfața grafică de tip User Friendly, care să fie ușor de înțeles și ușor de utilizat de către utilizatori. Datele de ieșire sunt afisate atât în interfața, cât și într-un fișier de ieșire. Fiecare secțiune din fișier, va avea următoarea structură:

-Momentul curent al simulării = un întreg.

-Waiting clients = sunt task-urile(clienți) care sunt reprezentați de către (id,timpul de așteptare,timpul de procesare), așteptând să fie adăugați în coadă.

-Queue (un întreg=care simbolizează nr cozi): clienți care se află la momentul actual în coadă.

**Fiecare server are asociat un thread =>multithreading.**



## Cerințe de funcționare

**Identificarea cerințelor de funcționare constituie un element critic în dezvoltarea unui sistem software.**

*Înainte să ne apucăm de proiectarea și implementarea propriu zisă a sistemului , este necesar să cunoaștem ce cerințe trebuie acesta să îndeplinească ,pentru a descoperii ce trebuie să facă sistemul ,cum trebuie să funcționeze ,pentru a stii ce obiective dorim să atingem la sfârșitul implementari.*

*În continuare voi prezenta ,sub forma unui tabel ,cerințele sistemului și constrângerile,pe care le-am identificat în urma analizei problemei.*

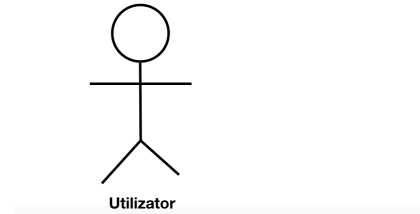
Descriere	
<b>Cerințe funcționale</b>	<ul style="list-style-type: none"><li>-Simulatorul trebuie să permită utilizatorului să introducă numărul clienților care urmează să fie procesați, precum și numărul coziilor în care urmează să fie inserați .</li><li>-Simulatorul trebuie să permită utilizatorului să introducă timpul maxim de simulare a aplicației, timpul la care va avea loc sfârșitul simulării.</li><li>-Simulatorul trebuie să permită utilizatorului să introducă timpul maxim/minim de arrive ,precum și timpul maxim/minim de service.</li><li>-Simulatorul trebuie să genereze aleatoriu un set de clienți , și să ii adauge în coada corespunzătoare(astfel încât timpul de asteptare sa fie minim) , la momentul de timp care este egal cu timpul lor de arrive. Simulatorul trebuie să anunțe utilizatorul în cazul unei erori ,care poate fi introducerea gresită a datelor , sau timpul minim să fie mai mare ca cel maxim</li><li>-Simulatorul trebuie să prezinte un buton de start ,prin care utilizatorul să pornească simularea ,precum și un buton de stop care va încheia execuția programului.</li></ul>
<b>Cerințe non-funcționale</b>	<ul style="list-style-type: none"><li>-Simulatorul de cozi trebuie să fie intuitiv și ușor de folosit</li><li>-Simulatorul de cozi trebuie să afișeze rezultatul într-un mod cât mai interactiv și ușor de citit și înțeles</li></ul>
<b>Constrângeri</b>	<ul style="list-style-type: none"><li>-Timpul maxim trebuie să fie mai mare decât cel minim.</li><li>-Numărul clienților trebuie sa fie mai mare ca 0, numărul coziilor mai mare decat 1</li></ul>

## Diagrama de use case

**Diagrama de use case prezintă o colecție de cazuri de utilizare și actori.**

**În cazul acestei teme :**

- **actorul este utilizatorul, cel care folosește simulatorul.**



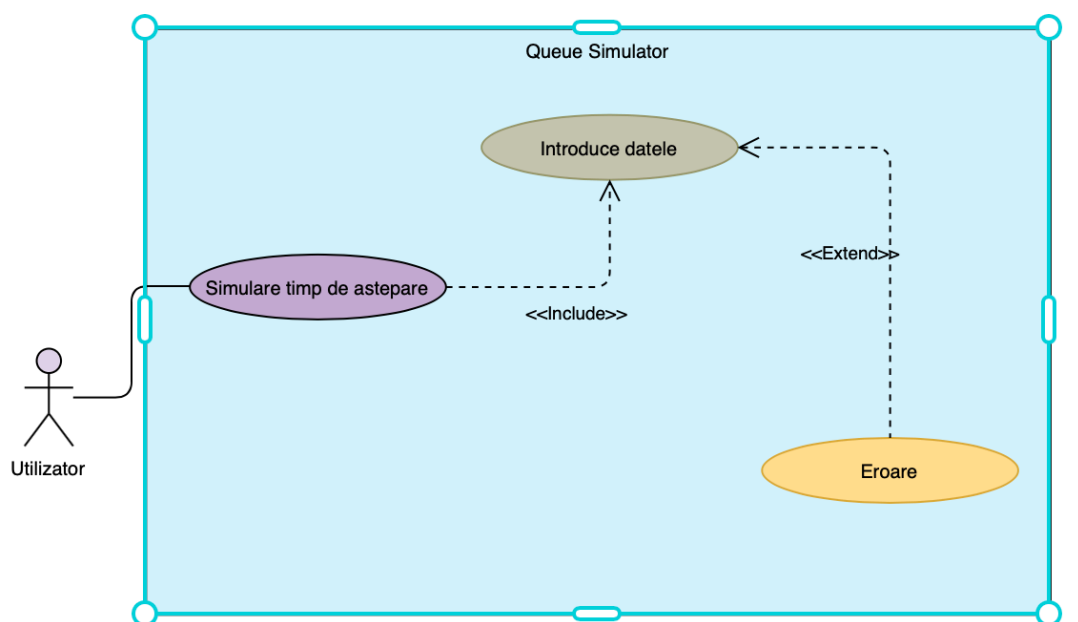
- **cazurile de utilizare în cazurile simulatorului sunt:**

- **În caz de succes:**

- utilizatorul introduce datele în interfața grafică.
- utilizatorul face click pe butonul **"start"**.
- simulatorul efectuează simularea unor clienți(task-uri) .Simularea va fi afisată în interfața grafică în timp real prin intermediul JTextArea
- aplicația salvează datele într-un fisier txt

- **În caz de nesucces:**

- utilizatorul introduce date invalide (nu respectă constrângerile amintite mai sus)
- simulatorul va genera o eroare
- se revine la primul pas din cele de succes



După cum se poate observa diagrama conține actorii sistemului, cazurile de utilizare și relațiile dintre ei.

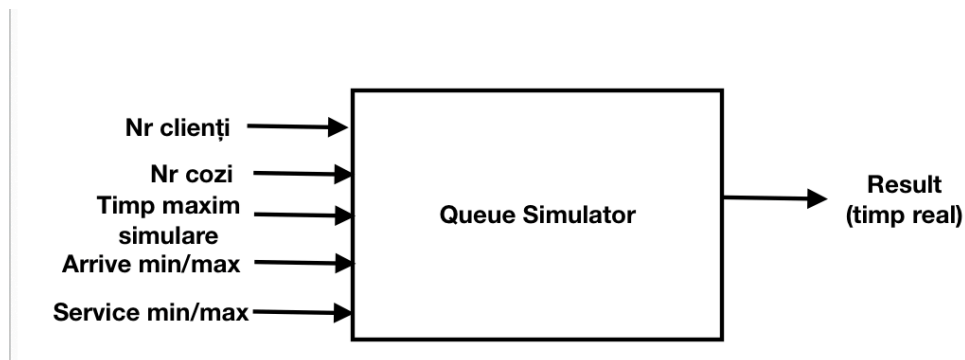
# 3.Proiectare

## 3.1.Etapa de design

*Prezintă mai multe nivele:*

- Design level 1

*În prima fază,am realizat o schemă bloc a sistemului de calcul,pentru a avea o privire de ansamblu asupra proiectului.*



*După cum se observă ,asupra cutiei negre se transmit date de intrare (introduse de către utilizator ),iar prin intermediul metodelor care se află în interiorul ei, se va genera în timp un rezultat.Această cutie neagră reprezintă o imagine globală a sistemului care urmează să fie implementat.*

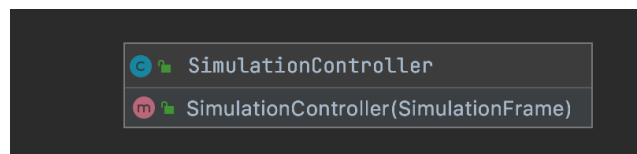
*În acest moment avem o viziune mai clară a ceea ce avem de construit, și anume un simulator care efectuează distribuirea unor clienți in cozile corespunzătoare , și care trebuie să genereze în timp real distrubuirea lor.*



- Design level 2

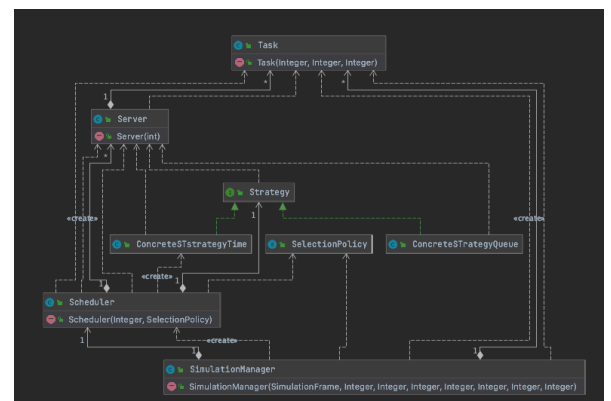
În a doua etapă ,am împărțit aplicația în 3 subpachete care reprezintă modelul arhitectural MVC(Model View Controller).Acesta ne permite să dezvoltăm,să implementăm și să testăm fiecare parte a programului independent ,menținând codul organizat. Însemnând un cod mai eficient și o modalitate mai bună de reutilizare a modelului.

*Pachetul controller conține clasa: SimulationControl*

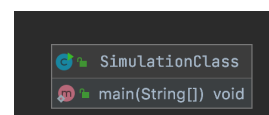


*Pachetul model*

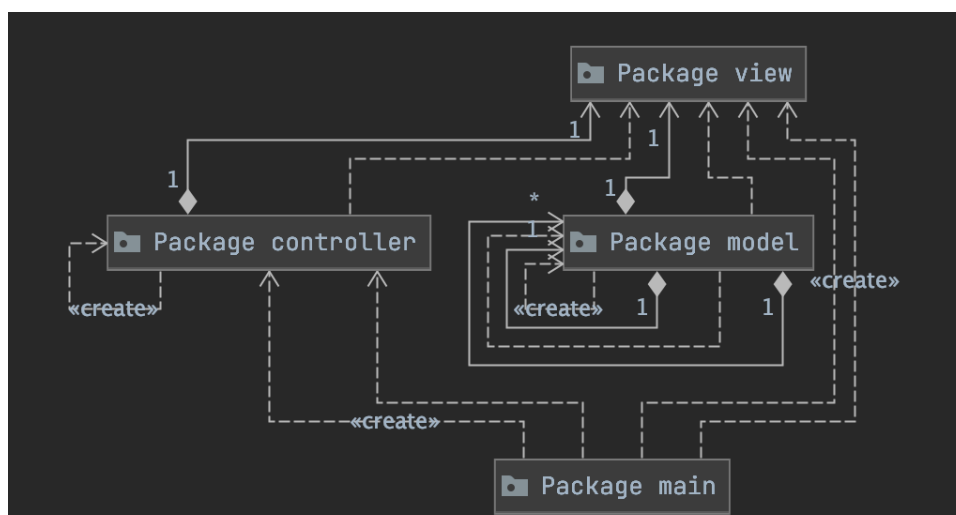
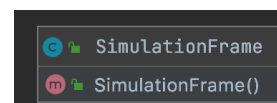
conține clasele: SimulationManager,  
ConcreteSTstrategyQueue  
ConcreteSTstrategyTime,  
Scheduler ,  
SelectionPolicy,  
Server,  
Strategy,  
Task



*Pachetul main conține clasa:SimulationClass*



*Pachetul view conține clasa:SimulationFrame*

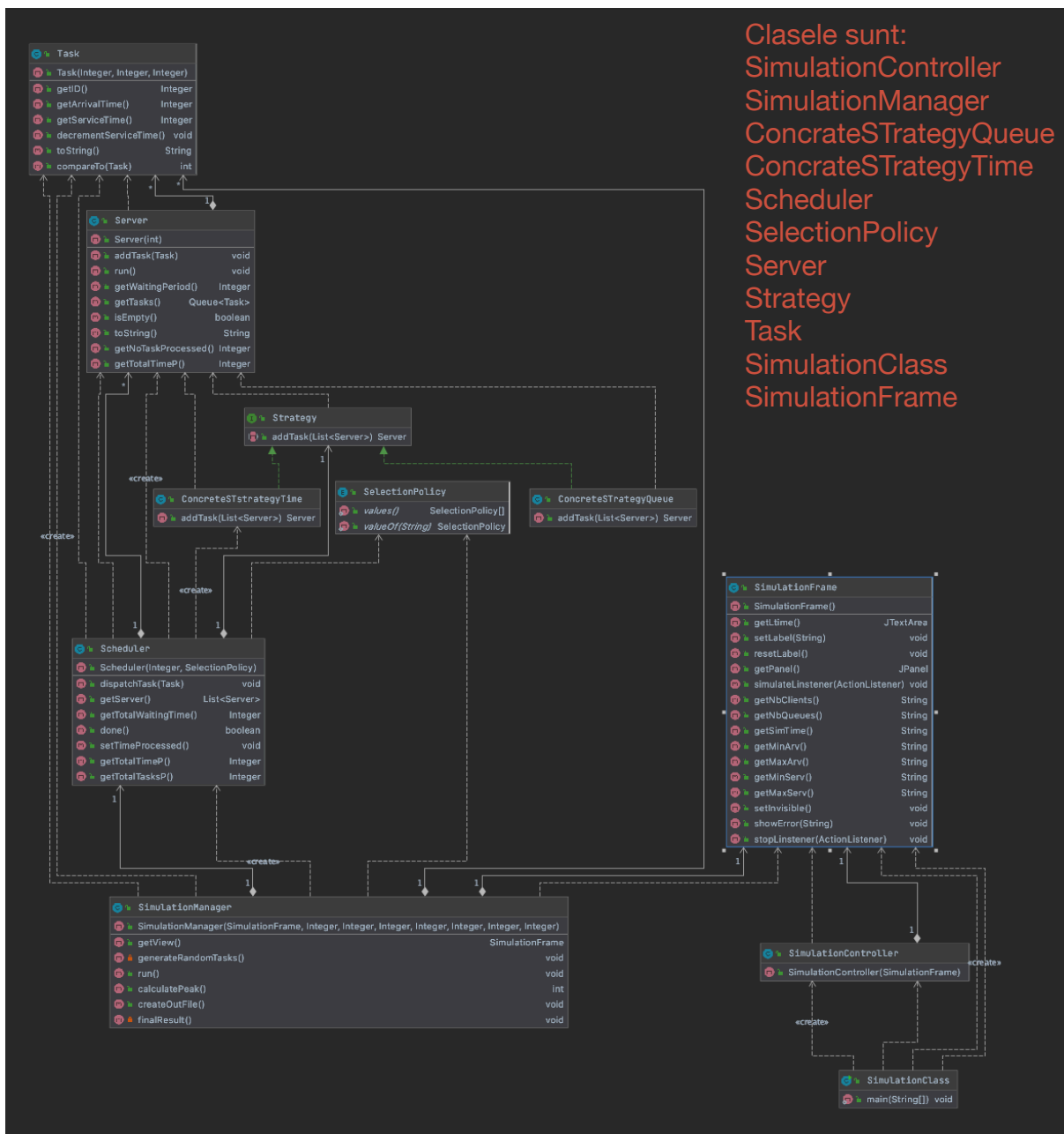


- Design level 3

In cea de a 3 etapă, se definesc clasele ,metodele și cum interacționează între ele  
Având toate aceste detalii am realizat diagrama UML.

**UML**-Unified Modeling Language.

**Reprezintă un set de clase,interfețe,colaborări și alte relații.**



## 3.2.Structuri de date

*Am decis ca în acest proiect să folosesc ca și structuri de date colecțiile.*

**Colecția : orice clasă care păstrează obiecte și implementează interfața Collection.**

- ArrayList pentru stocarea task-urilor în SimulationManager ,și pentru stocarea serverelor în Scheduler
- BlockingQueue-colecție concurentă și thread-safe- pentru stocarea task-urilor care sunt în coada de execuție a unui server.Aceste structuri sunt necesare datorita siguranței pe care o oferă in cazul thread-urilor, având în vedere faptul ca servere au fiecare un fir de lucru propriu
- Totodata folosesc și AtomicInteger pentru stocarea timpilor.

## 3.3.Algoritmi folosiți

Pe partea de algoritmică nu s-au folosit algoritmi fundamentali,decât cateva foreach-uri sau while-uri ,pentru parcurgerea servelor /task-urilor.

## 3.4.Interfața grafică (Graphical User Interface)

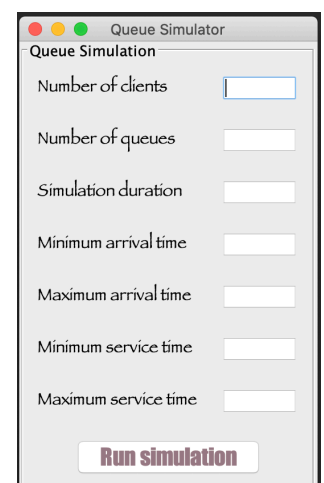
**Reprezintă un mecanism prietenos pentru interacțiunea utilizatorului cu programul.**

*Folosim o interfață User-Friendly pentru a permite utilizatorilor să se simta mai familiarizați cu programul chiar înainte de a-l fi utilizat. Printr-o interfață utilizatorul poate înțelege mai bine cum funcționează programul ,poate învăța mai repede modul de utilizare a acestui calculator.*

**Interfața grafică cuprinde următoarele componente:**

**Panou** - care cuprinde toate elementele și a cărui titlu este setat ca fiind "Queue Simulator".

**Buton** - "RUN SIMULATION",după cum sugerează și numele lui,acest buton este folosit pentru pornirea simulării.Este de tip JButton



**TextField-uri** -spațiile în care se pot introduce date de la tastatură sau se pot afișa rezultatele.

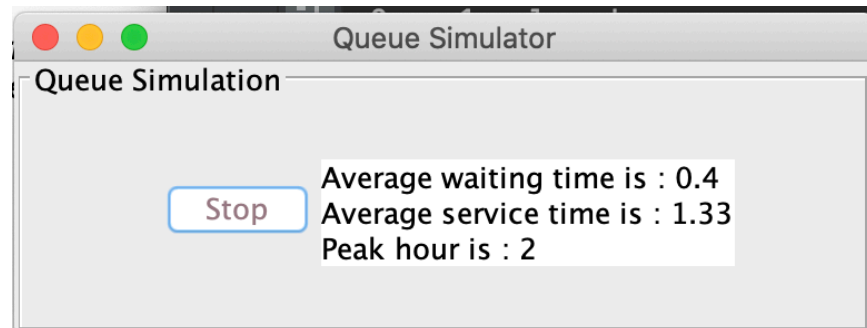
-sunt în număr de 7 :

- pentru Number of clients.
- pentru Number of queues.
- pentru Simulation duration.
- pentru Minimum arrival time.
- pentru Maximum arrival time.
- pentru Minimum service time.
- pentru Maximum service time.

**Label** - este o etichetă care conține numele datelor care trebuie introduse .  
Sunt în număr de 7.

După ce se apasă butonul “Run Simulation”, se începe simularea ,și se declanșează ceea de a doua parte a interfeței ,care poate fi observată în figura din dreapta. Aceasta interfață prezintă un buton de tip JButton, “stop” care are ca scop oprirea programului ,închiderea execuției acestuia.

Totodată aici este prezent și un JTextArea care are ca scop afișarea simulării în timp real, și la sfârșitul simulării afișează timpul mediu de așteptare,timpul mediu de servire,și ora de vârf.



Average waiting time — înainte să pun client-ul în coadă, am salvat valoarea waitTime-ului, după care am făcut media pentru toate task-urile care au ajuns înainte de expirarea timpului.

Average service time — se calculează ca fiind raportul dintre suma totală dintre service-Time-ul fiecărui client procesat , și numărul de clienți procesați.

Peak hour — reprezintă momentul de timp în care se găsesc cei mai mulți clienți în magazin. Adică suma maximă a tuturor task-urilor din toate cozile .

**!**În main se instantiază interfața grafică , deci de fiecare dată când este rulat main se formează o nouă interfață.

# 4.Implementare

**Clasele folosite sunt destul de ușor de înțeles și interpretat.**

## **1.Clasa SimulationClass**

*Este clasa în care spunem programului ce să execute.Sunt declarate 2 obiecte unul de tip SimulationFrame , ,și unul de tip Controller care are ca argumente un SimulationFrame .Care vor determina deschiderea interfeței grafice unde se întâmplă toate interacțiunile utilizatorului cu sistemul de simulare .*

```
public class SimulationClass {  
    public static void main(String[] args) {  
        SimulationFrame simulationFrame = new SimulationFrame();  
        SimulationController simulationController = new SimulationController(simulationFrame);  
        simulationFrame.setVisible(true);  
    }  
}
```

*simulationFrame.setVisible(true); — va face interfața vizibilă*

## **2.Clasa Task**

*—reprezintă munca care trebuie efectuată*

*Descriere task-ul/clientul (având ca atribut principal taskID ),care este pus în coadă, de către Scheduler la momentul arrivalTime , fiind procesat de serverul*

*unde va fi alocat .Timpul lui de procesare (adică durata de timp pe care o petrece în server) este dat de atributul serviceTime.Toate aceste attribute sunt de tipul Integer și sunt private.*

*Pentru toate aceste attribute avem metode de get, plus mai există o metodă denumită decrementServiceTime care are ca scop scăderea timpului de procesare a tasku-ului , și este apelată de către server la fiecare cuanta de timp.Aici se suprascrive și metoda toString ,care are ca scop formatarea informațiilor ,referitoare la task, pentru afisarea mai frumoasă a acestor informații.Clasa implementează și interfața Comparable ,pentru a sorta clienți în funcție de timpul de sosire ,iar dacă timpul lor de sosire este identic ,clienții se sortează după taskID.*

```
public Task(Integer taskID, Integer arrivalTime, Integer serviceTime) {  
    this.taskID = taskID;  
    this.arrivalTime = arrivalTime;  
    this.serviceTime = serviceTime;  
}
```

### 3. Clasa Server

—primește și execută un task

Descriere una din cozile la care se pot adauga task-urile. Astfel avem un atribut `tasks` de tip `BlockingQueue<Task>`, care stochează task-urile care mai trebuie executate, `waitingPeriod` de tipul `AtomicInteger` care reprezintă timpul pe care trebuie să-l petreacă un client în coadă până ajunge în capăt. Mai avem și `noTaskProcessed` și `totalTimeP`, care ajuta la calcularea timpului mediu de servire. `noTaskProcessed` crește atunci când un client a fost procesat complet, iar `totalTimeP` reprezintă timpul lui de service. Totodată mai prezintă și un atribut `serverID` de tip `int`, flagul boolean `running` care serveste la oprirea thread-ului atunci când simularea se sfârșeste.

Avem o metoda de `addTask`, care are ca argument un task, pe care il adaugă în task-uri, și actualizează `waitingPeriod`.

După cum se observă această clasă implementează interfața `Runnable`, deci există metoda `run`, care se va executa pe un thread separat, toate serverele procesând clienți în mod paralel. În această metoda verificăm dacă mai exista un task nou, dacă nu există se suspenda pentru o secundă. Dacă în această perioadă de timp a fost adaugat un task sau mai mulți, atunci se începe procesarea lor. Printr-un while parcurgem task-ul atâta timp cât timpul de procesare a lui este diferit de 0. În interiorul while-ului se decrementează timpul de procesare a task-ului și se decrementează `waitingPeriod`, după care așteptăm o secundă pentru a simula trecerea timpului. Când timpul lui de servire ajunge la 0, se elimină din tasks, se adugă timpul lui de service la timpul total de procesare și se incrementează nr de task-uri procesate. Se repetă acești pași până când thread-ul este intrerupt din exterior. Mai avem și metode de `get`, precum și o metodă care verifică dacă nu mai există task-uri, și care va fi apelată în `SimulationManager`. Este suprascrisă și metoda `toString`, care ajută la o afisare mai coerentă, mai eficientă.

```
public Server(int serverID) {
    this.serverID = serverID;
    tasks = new LinkedBlockingDeque<>();
    waitingPeriod = new AtomicInteger( initialValue: 0);
}
```

### 4. Clasa Schedule

—alege server-ul cel mai potrivit pentru task-ul dat

Este responsabilă pentru managementul serverilor și distribuirea task-urilor, și de asemenea reprezintă clasa prin care `SimulationManager` comunica cu toate serverele pe care le are în subordonanță. Prezintă ca attribute private o lista care stochează serverele, un `Integer` care prezintă numărul de servere care se vor forma, strategia aplicată, pentru a eficientiza procesul, care este de tipul `SHORTEST_TIME`, mai multe `AtomicInteger` care retin `totalWaitingTime`, `totalTimeP`, `totalTaskP`. În constructor se crează serverele, și se pornesc thread-urile. Aici se setează timpul de procesare, și numărul de task-uri procesate. Mai întâlnim și metoda `dispatchTask`, care adauga un task intr-un server, acest server fiind ales printr-o strategie, fie ori în funcție de numărul de task-

```
public Scheduler(Integer noOfServers, SelectionPolicy strategy)
{
    if (strategy == SelectionPolicy.SHORTEST_TIME)
        this.strategy = new ConcreteStrategyTime();
    servers = new ArrayList<>();

    this.noOfServers = noOfServers;
    for (int i = 0; i < noOfServers; i++) {
        Server server = new Server( serverID: i + 1);
        servers.add(server);
        (new Thread(server)).start();
    }
    this.totalWaitingTime = new AtomicInteger( initialValue: 0);
    this.totalTimeP = new AtomicInteger( initialValue: 0);
    this.totalTaskP = new AtomicInteger( initialValue: 0);
}
```



uri care se mai regasesc în el, fie în funcție de timpul de așteptare pe care l-ar avea task-ul dacă ar fi adăugat, astfel prin strategia aplicată, se va alege cel mai bun server unde să fie introdus task-ul. Metoda `done` are ca scop să verifice dacă s-au închis toate serverele, nu mai există thread-uri. Se întâlnesc și alte metode de `get`.

## 5. Clasa *SimulationManager* —simulatorul aplicației

Este clasa responsabilă cu realizarea simulării ca întreg. Atunci când se apelează constructorul se creează o instanță de tipul *SimulationFrame*, și până la apăsarea butonului de *Run Simulation* din interfață, această clasă nu face absolut nimic. După ce se apasă butonul se formează o instanță de tip *scheduler*, se generează task-urile random cu ajutorul metodei *generateRandomTasks*, care creează câte task-uri a definit utilizatorul, și setează timpul de procesare ca fiind un număr random între timpul minim de procesare și cel maxim introdus de utilizator, și setează timpul de arrive ca fiind un număr random între timpul minim de arrive și cel maxim introdus de utilizator.

```
public SimulationManager(SimulationFrame simulationFrame, Integer numberOfTask, Integer maxArrivalTime, Integer minProcessingTime, Integer timeLimit) {  
  
    this.numberOfTask = numberOfTask;  
    this.numberOfServers = numberOfServers;  
    this.minArrivalTime = minArrivalTime;  
    this.maxArrivalTime = maxArrivalTime;  
    this.minProcessingTime = minProcessingTime;  
    this.maxProcessingTime = maxProcessingTime;  
    this.timeLimit = timeLimit;  
    this.scheduler = new Scheduler(numberOfServers, SelectionPolicy.SHORTEST_TIME);  
    generateRandomTasks();  
    logger = new Logger();  
    this.simulationFrame = simulationFrame;  
  
}
```

După care sortează task-urile prin metoda implementată în clasa *Task*.

Se creează o instanță de tip *Logger* care ne va ajuta în scrierea în fișier, în interfață și respectiv în consolă. Aici se calculează și *PeakHour*, a cărui metoda de calcul am descris-o în Capitolul 3. Există și o metodă de *createOutFile*, care creează fișierul de ieșire, dându-i numele ca fiind *Result+numarul de secunde din prezent*. Metoda *finalResult* calculează timpul mediu de servire și așteptare, calculul lor a fost descris de asemenea în Capitolul 3, am folosit metoda *math.round* pentru a rotunji numărul, astfel încât dacă restul împărțirii va fi 1,66668, *math.round* îmi va da ca fiind 1,67, totodată această metodă creează string-ul, și apelează cu acesta metoda *logWriteResFinal* din *logger*.

După cum se poate observa această metodă implementează interfața *Runnable*, deci există o metodă de *run()*. În metodă avem un *while* care permite executarea metodei atâta timp cât timpul simulării actuale este mai mic decât timpul maxim introdus de noi, când se depășește timpul introdus de noi aplicația se oprește, sau se mai oprește la timpul în care nu mai există clienți care așteaptă să fie introdusi în cozi, deci serverele sunt închise. Se verifică dacă există task-uri, dacă există, *currentTask* ia primul task, după care atâta timp cât *currentTask* are timpul de arrive egal cu timpul curent de simulare, le trimitem la *scheduler* pentru a le găsi coada optimă în care să fie adăugate, și se calculează un *peakTime* intermediar, după ce acest task a fost procesat se elimină din lista de task-uri, și *currentTask* ia următorul task prezent în listă, dacă nu mai există task-uri în listă se iese din *while*-ul în-

tern , si se actualizează informațiile prin intermediul Logger-ului,după care se incrementează timpul de simulare , acum asteptam o secunda să se genereze un alt task,sau mai multe task-uri ,dacă nu se genereaza alte task-uri și scheduler și-a terminat execuția se încheie execuția programului.

## 6.Clasa **SimulationFrame** —extinde class **JFrame**

Este folosită pentru implementarea interfeței grafice,ceea ce vedem noi utilizatori.Prezintă 2 butoanele pentru cele două operații de Run si Stop.Mai întâlnim aici și numeroase textField-uri care ne permit scrierea datelor ,precum și etichete JLabel care ne îndrumă unde ar trebui scrise fiecare date .Totodată folosim și un JTextArea unde se vor scrie în timp real datele.

```
public SimulationFrame() {  
  
    this.setSize( width: 300, height: 300);  
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
  
    c.anchor = GridBagConstraints.WEST;  
    c.insets = new Insets( top: 10, left: 10, bottom: 10, right: 10);  
  
}
```

Am folosit și GridBagConstraints care mă ajută la aranjarea mai eficientă a componentelor în interfața grafică .Aici se gasesc diferite metode care fac legătura între butoane și controller,precum și metode de afisare ale erorilor , de setare a textArea,de setare a vizibilitați unor componente la anumite perioade de timp,si metode de get, pentru preloarea datelor din TextField-uri.

## 7.Clasa **SimulationController**

Are un atribut un SimulationFrame.

Traduce interacțiunile utilizatorului cu SimulationFrame, în acțiuni pe care le va efectua SimulationManager

Fiecare metodă întâlnită aici corespunde fiecarui buton prevăzut în interfață ,și descrie comportamentul programului în urma apăsării unuia dintre butoane .Astfel aici se găsesc metode pentru fiecare buton .În fiecare metodă se preiau datele de intrare ca

și niste string-uri și se convertesc în Integer prin intermediul metodei Integer.parseInt .Dacă datele citite nu respectă constrângerile impuse ,se vor genera erori și programul nu va mai funcționa. În cazul în care nu se întâlnesc erori se formează o instanță de tip SimulationFrame,după care se crează fisierul în care se vor scrie rezultatele , se creaza thread-ul principal ,după care se porneste simularea .

```
public class SimulationController {  
    private SimulationFrame simulationFrame;  
  
    public SimulationController(SimulationFrame simulationFrame) {  
        this.simulationFrame = simulationFrame;  
        simulationFrame.simulateListener(new SimulateListener());  
        simulationFrame.stopListener(new StopListener());  
    }  
}
```



## **8. Interfața Strategy**

*Această clasă obliga orice clasă care o implementează să implementeze metoda `addTask`, care este utilizată în `dispatchTask`. Metoda `addTask` returnează cel mai eficient server din punct de vedere a unui algoritm pe care îl voi prezenta mai jos, urmând ca în acest server generat să se adauge un task.*

## **9. Clasa ConcreteSTstrategyTime**

*Implementează interfața Strategy, deci prezintă metoda `addTask()`. Aici, această metoda returnează serverul care are cel mai mic timp de așteptare. Prima dată avem un atribut, de tip `server`, `optimalServer` care ia primul server din lista de servere, apoi un atribut de tip `int`, `minimumWaitingPeriod`, care ia timpul de așteptare a `optimalServer`, cu un `for` parcurgem toate serverele din listă, și le comparăm timpul de așteptare cu timpul minim declarat mai sus, dacă se găsește un server care are un timp de așteptare mai mic, se schimbă serverul optim cu acel server.*

## **10. Clasa ConcreteSTstrategyTime**

*Implementează interfața Strategy, deci prezintă metoda `addTask()`. Aici, această metoda returnează serverul care are cele mai puține task-uri. Prima dată avem un atribut, de tip `server`, `optimalServer` care ia primul server din lista de servere, apoi un atribut de tip `int`, `minimumServer`, care ia dimensiunea lui `optimalServer`, cu un `for` parcurgem toate serverele din listă, și le comparăm numărul de task-uri cu numărul declarat mai sus, dacă se găsește un server care are un număr de task-uri mai mic, se schimbă serverul optim cu acel server.*

## **11. TesteUnitare()**

*Aici se vor efectua testele de verificare ale sistemului de*

*Prin intermediul interfeței grafice utilizatorul poate introduce parametrii pe care dorește să execute simularea, cu ajutorul tastaturii. După care apasă pe `Run Simulation`, și astfel are loc execuția programului, și schimbarea aspectului interfeței. Interfața va fi actualizată la fiecare secundă de timp. După ce simularea sa încheiat, informațiile afișate sunt salvate într-un fișier text cu numele `Result` concatenat cu secunda curentă. Dacă se dorește efectuarea altei simulări, cu date diferite, se apasă butonul `stop`, după care trebuie apasat butonul `RUN` din clasa `SimulationClass`*

# 5.Rezultate

*Pentru a testa corectitudinea programului am utilizat două metode:*

*1.Prima metodă a fost introducerea unor date de la tastatură ,în repetate rânduri pentru a acoperi o varietate cât mai mare de cazuri.  
Am scris pe foaie ,coziile/coada la fiecare moment de timp , am calculat timpul mediu de servire,și de asteptare ,și totodată am analizat și care ar putea fi ora de vârf.După care am comparat rezultatul de pe foaia mea ,cu ce este scris în consolă.  
Am tot efectuat această metodă până am fost convinsă că programul funcționează corect.*

*2.A doua metodă a fost testarea cu JUnit.*

*În această metodă am folosit Assertions , mai exact **assertTrue**.*

*Ținând cont de faptul ca simulăm o aplicație care generează random niste numere,nu prea putem face o testare riguroasă a aplicației , având și thread-uri pot apărea și probleme de sincronizare.*

*Totuși am încercat să fac o așa zisă testare, bazându-mă pe proprietatea thread-urilor de a genera erori,astfel am apelat thread-ul cu join , și în cazul în care nu este prinsă o excepție, de tipul InterruptedException, să returneze true.*

*În varianta finală a proiectului nu au mai aparut erori sau greseli de calcule.Am rezolvat toate erorile găsite și întâmpinate de a lungul proiectului.*

Ca date de intrare am introdus datele din laborator.

Pentru primul caz :

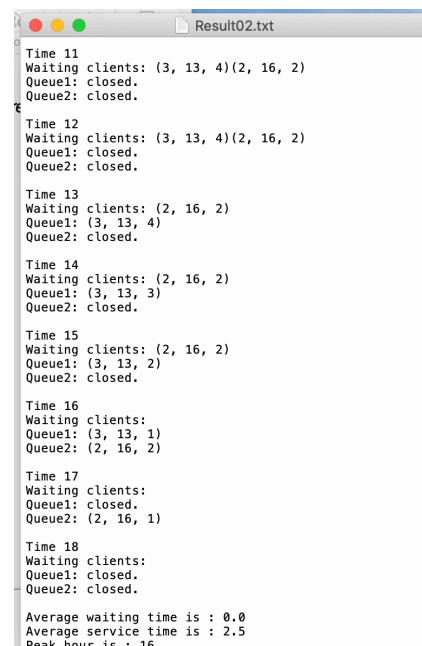
N=4,

Q=2,

Tmax=60,

[TAmin,TAmx]=[2,30],

[TSmin,TSmax]=[2,4]



```
Result02.txt
Time 11
Waiting clients: (3, 13, 4)(2, 16, 2)
Queue1: closed.
Queue2: closed.

Time 12
Waiting clients: (3, 13, 4)(2, 16, 2)
Queue1: closed.
Queue2: closed.

Time 13
Waiting clients: (2, 16, 2)
Queue1: (3, 13, 4)
Queue2: closed.

Time 14
Waiting clients: (2, 16, 2)
Queue1: (3, 13, 3)
Queue2: closed.

Time 15
Waiting clients: (2, 16, 2)
Queue1: (3, 13, 2)
Queue2: closed.

Time 16
Waiting clients:
Queue1: (3, 13, 1)
Queue2: (2, 16, 2)

Time 17
Waiting clients:
Queue1: closed.
Queue2: (2, 16, 1)

Time 18
Waiting clients:
Queue1: closed.
Queue2: closed.

Average waiting time is : 0.0
Average service time is : 2.5
Peak hour is : 16
```

### Pentru al doilea caz :

N=50,  
Q=5,  
Tmax=60,  
[TAmin,TAmax]=[2,40],  
[TSmin,TSmax]=[1,7]

```
Result21.txt

Waiting clients:
Queue1: closed.
Queue2: closed.
Queue3: (9, 40, 5)
Queue4: closed.
Queue5: (32, 40, 2)

Time 47
Waiting clients:
Queue1: closed.
Queue2: closed.
Queue3: (9, 40, 4)
Queue4: closed.
Queue5: (32, 40, 1)

Time 48
Waiting clients:
Queue1: closed.
Queue2: closed.
Queue3: (9, 40, 3)
Queue4: closed.
Queue5: closed.

Time 49
Waiting clients:
Queue1: closed.
Queue2: closed.
Queue3: (9, 40, 2)
Queue4: closed.
Queue5: closed.

Time 50
Waiting clients:
Queue1: closed.
Queue2: closed.
Queue3: (9, 40, 1)
Queue4: closed.
Queue5: closed.

Time 51
Waiting clients:
Queue1: closed.
Queue2: closed.
Queue3: closed.
Queue4: closed.
Queue5: closed.

Average waiting time is : 3.68
Average service time is : 4.24
Peak hour is : 27
```

### Pentru al treilea caz :

N=1000,  
Q=20,  
Tmax=200,  
[TAmin,TAmax]=[10,100],  
[TSmin,TSmax]=[3,9]

```
Result14.txt

87, 5)(842, 89, 8)(913, 91, 5)(640, 93, 4)(221, 95, 6)(918, 96, 8)(183, 99, 9)
Queue6: (636, 68, 5)(475, 69, 6)(301, 71, 9)(599, 73, 5)(750, 74, 5)(923, 75, 6)(530, 77, 4)(644, 78, 8)(618, 80, 3)(536, 81, 7)(357, 85, 6)(998, 87, 5)(868, 89, 4)(494, 90, 5)(698, 92, 9)(680, 95, 4)(619, 96, 4)(798, 97, 4)(834, 98, 8)
Queue7: (661, 67, 3)(164, 69, 4)(253, 70, 6)(757, 71, 8)(236, 74, 9)(932, 75, 4)(864, 76, 3)(860, 77, 8)(881, 79, 6)(671, 81, 6)(297, 85, 4)(871, 86, 4)(474, 88, 4)(880, 89, 4)(587, 90, 4)(2, 92, 6)(900, 93, 5)(731, 95, 8)(982, 97, 8)(460, 100, 5)
Queue8: (282, 68, 4)(364, 69, 4)(343, 70, 9)(482, 72, 9)(46, 75, 9)(216, 77, 6)(847, 78, 4)(25, 80, 7)(331, 82, 7)(198, 86, 7)(953, 88, 6)(144, 90, 9)(548, 93, 9)(315, 96, 4)(311, 97, 5)(929, 98, 6)
Queue9: (420, 67, 2)(128, 69, 4)(928, 69, 4)(34, 71, 4)(858, 71, 5)(173, 73, 4)(323, 74, 9)(414, 76, 5)(21, 78, 5)(972, 78, 5)(77, 80, 9)(780, 84, 3)(208, 86, 5)(223, 88, 8)(150, 90, 4)(630, 91, 9)(23, 95, 8)(157, 97, 4)(292, 98, 4)(366, 99, 3)(528, 100, 4)
Queue10: (531, 68, 6)(27, 70, 4)(162, 71, 4)(916, 71, 6)(645, 73, 9)(846, 75, 3)(436, 76, 7)(377, 78, 5)(585, 79, 9)(389, 83, 6)(256, 86, 6)(478, 88, 9)(777, 90, 9)(917, 93, 3)(226, 95, 7)(170, 97, 8)(426, 99, 8)
Queue11: (651, 68, 5)(477, 69, 9)(52, 72, 3)(500, 72, 3)(712, 73, 8)(545, 75, 3)(140, 76, 9)(660, 78, 7)(261, 80, 5)(344, 82, 6)(509, 85, 8)(408, 89, 9)(307, 91, 5)(28, 93, 6)(238, 95, 4)(491, 96, 7)(538, 98, 5)(967, 99, 5)
Queue12: (792, 68, 6)(43, 70, 7)(759, 71, 4)(766, 72, 6)(382, 74, 5)(579, 75, 7)(237, 77, 5)(845, 78, 9)(29, 81, 9)(646, 85, 6)(355, 88, 9)(684, 90, 6)(774, 92, 3)(789, 93, 4)(328, 95, 9)(866, 97, 8)(139, 100, 5)
Queue13: (556, 68, 8)(505, 70, 7)(285, 72, 9)(574, 74, 4)(589, 75, 9)(135, 78, 7)(688, 79, 3)(502, 80, 6)(600, 83, 9)(32, 87, 4)(483, 89, 9)(582, 91, 5)(217, 93, 5)(116, 95, 8)(171, 97, 8)(523, 99, 3)(618, 100, 3)
Queue14: (37, 68, 2)(152, 69, 5)(316, 70, 9)(348, 72, 3)(276, 73, 5)(744, 74, 7)(206, 76, 5)(565, 77, 6)(113, 79, 6)(738, 80, 4)(584, 82, 9)(875, 86, 4)(526, 88, 3)(784, 89, 7)(593, 91, 7)(872, 93, 4)(421, 95, 7)(176, 97, 6)(990, 98, 3)(9, 100, 8)
Queue15: (577, 68, 5)(662, 69, 5)(222, 71, 9)(568, 73, 8)(318, 75, 6)(713, 76, 5)(354, 78, 5)(305, 79, 8)(767, 81, 3)(6, 84, 5)(394, 86, 6)(658, 88, 5)(99, 90, 5)(617, 91, 8)(968, 93, 5)(758, 95, 6)(583, 97, 9)(204, 100, 9)
Queue16: (471, 68, 4)(418, 69, 3)(326, 70, 8)(339, 72, 5)(933, 73, 3)(397, 74, 9)(469, 76, 3)(413, 77, 6)(899, 78, 3)(729, 79, 7)(266, 82, 3)(78, 84, 9)(111, 87, 3)(677, 89, 6)(202, 90, 4)(695, 91, 9)(189, 95, 6)(756, 96, 8)(147, 99, 6)
Queue17: (510, 68, 1)(35, 69, 8)(678, 70, 7)(412, 72, 6)(295, 74, 8)(3, 76, 5)(467, 77, 4)(456, 78, 9)(24, 81, 8)(895, 84, 5)(934, 86, 4)(716, 88, 8)(702, 90, 5)(725, 92, 6)(796, 94, 8)(49, 97, 5)(753, 98, 7)(971, 100, 7)
Queue18: (299, 68, 1)(121, 69, 8)(741, 70, 3)(596, 71, 5)(769, 72, 8)(942, 74, 3)(637, 75, 4)(524, 76, 9)(965, 78, 9)(342, 81, 5)(143, 84, 8)(175, 87, 7)(104, 90, 4)(919, 90, 5)(797, 92, 7)(571, 95, 4)(496, 96, 9)(148, 99, 4)(333, 100, 3)
Queue19: (321, 68, 2)(163, 69, 5)(332, 70, 7)(243, 72, 4)(801, 72, 9)(346, 75, 7)(187, 77, 3)(225, 78, 7)(829, 79, 8)(649, 82, 7)(539, 86, 4)(310, 87, 6)(893, 89, 8)(294, 92, 6)(4, 94, 6)(242, 96, 5)(631, 97, 6)(214, 99, 6)
Queue20: (522, 68, 3)(201, 69, 6)(863, 70, 5)(280, 72, 6)(938, 73, 3)(553, 74, 8)(302, 76, 5)(606, 77, 6)(127, 79, 4)(60, 80, 7)(666, 83, 3)(92, 85, 9)(828, 88, 9)(250, 91, 9)(468, 94, 4)(686, 95, 7)(765, 97, 6)(220, 99, 6)

Average waiting time is : 101.69
Average service time is : 6.02
Peak hour is : 100
```

## 6.Concluzie

În urma realizării acestui proiect , am dobândit cunoștințe noi legate de thread-uri și de modul lor de utilizare ,precum am descoperit cât de bogate sunt pachetele în structuri de date de tip thread safe și cum se utilizează design pattern-ul de tip Strategy.Am acumulat și cunoștințe noi despre scrierea în fișiere și despre crearea fișierelor

*Ca și dezvoltare ulterioară, s-ar putea îmbunătăți afișarea în interfața grafică astfel încât să fie mult mai dinamică, s-ar putea îmbunătăți funcționalitatea butonului stop, astfel încât aplicația să se oprească când se apasă butonul și să afișeze pentru acel timp la care s-a apăsător ,datele de ieșire.*

## 7.Bibliografie

1.<https://ro.wikipedia.org>

2.<https://stackoverflow.com>

3.<https://www.tutorialspoint.com/index.htm>

4.<https://www.youtube.com>