

DOCUMENTAȚIE

TEMA2

-Gestionarea cozilor-

Nume student: Buda Andreea Rodica

Grupa: 30221

CUPRINS

1. Obiectivul temei	3
2. Analiza problemei, modelare, scenarii, cazuri de utilizare.....	3
3. Proiectare	5
4. Implementare	7
5. Rezultate	17
6. Concluzii.....	18
7. Bibliografie.....	18

1. OBIECTIVUL TEMEI

Obiectivul principal al temei este de a proiecta și implementa o aplicație care să permită analizarea sistemelor bazate pe cozi prin simularea unui număr N de clienți care sosesc, așteaptă în cozi, primesc servicii și apoi părăsesc coada, și prin calcularea timpului mediu de așteptare, a timpului mediu de servire și a orei de vârf, în care se află cei mai mulți clienți. Pentru a atinge obiectivul principal al proiectului, se vor urmări obiective secundare pentru a dezvolta o aplicație funcțională și eficientă.

Obiectivele secundare ale temei sunt următoarele:

- Analiza problemei, modelare, scenarii, cazuri de utilizare:

Prezentarea cadrului de cerințe funcționale și non-funcționale și cazurile de utilizare sub forma de diagrame și descrieri de use-case. (Capitolul 2)

- Proiectarea:

Proiectarea orientată pe obiecte a aplicației, prin definirea diagramei UML de clase și de pachete, a structurilor de date folosite, a interfețelor definite și a algoritmilor folosiți. (Capitolul 3)

- Implementarea:

Descrierea fiecărei clase cu câmpuri și metode importante, și a implementării interfeței utilizator. (Capitolul 4)

- Testarea:

Prezentarea scenariilor de testare, aplicația va fi supusă testelor pentru a verifica funcționalitatea și eficiența sa. (Capitolul 5)

- Concluzii:

Prezentarea concluziilor trase în urma implementării aplicației, a ceea ce s-a învățat din acest proces și a posibilelor dezvoltări ulterioare. (Capitolul 6)

- Bibliografie:

Adăugarea referințelor consultate pe parcursul implementării temei. (Capitolul 7)

2. ANALIZA PROBLEMEI, MODELARE, SCENARII, CAZURI DE UTILIZARE

a. Analiza problemei:

Problema: Dezvoltarea unei aplicații de simulare a unui sistem de coadă cu mai multe cozi și clienți. Utilizatorul trebuie să poată introduce parametrii de intrare și să pornească simularea.

Cerințe funcționale:

- Aplicația de simulare ar trebui să permită utilizatorului să introducă parametrii de intrare, cum ar fi numărul de clienți, numărul de cozi, intervalul de simulare, timpul minim și maxim de sosire și timpul minim și maxim de serviciu.
- Aplicația de simulare ar trebui să permită utilizatorului să înceapă simularea după validarea datelor de intrare.
- Aplicația de simulare ar trebui să afișeze evoluția cozilor în timp real.

Cerinte non-funcționale:

- Aplicația de simulare trebuie să fie intuitivă și ușor de folosit pentru utilizator.
- Aplicația de simulare ar trebui să ofere un timp de răspuns rapid și o performanță bună, astfel încât să poată fi folosită pentru simulări de sistem complexe.
- Aplicația de simulare ar trebui să fie stabilă și să nu genereze erori în timpul utilizării.

b. Modelare:

Diagrama de pachete:

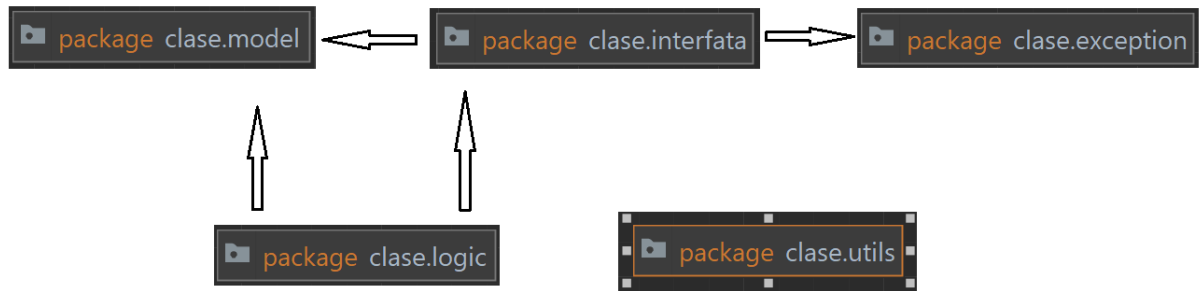
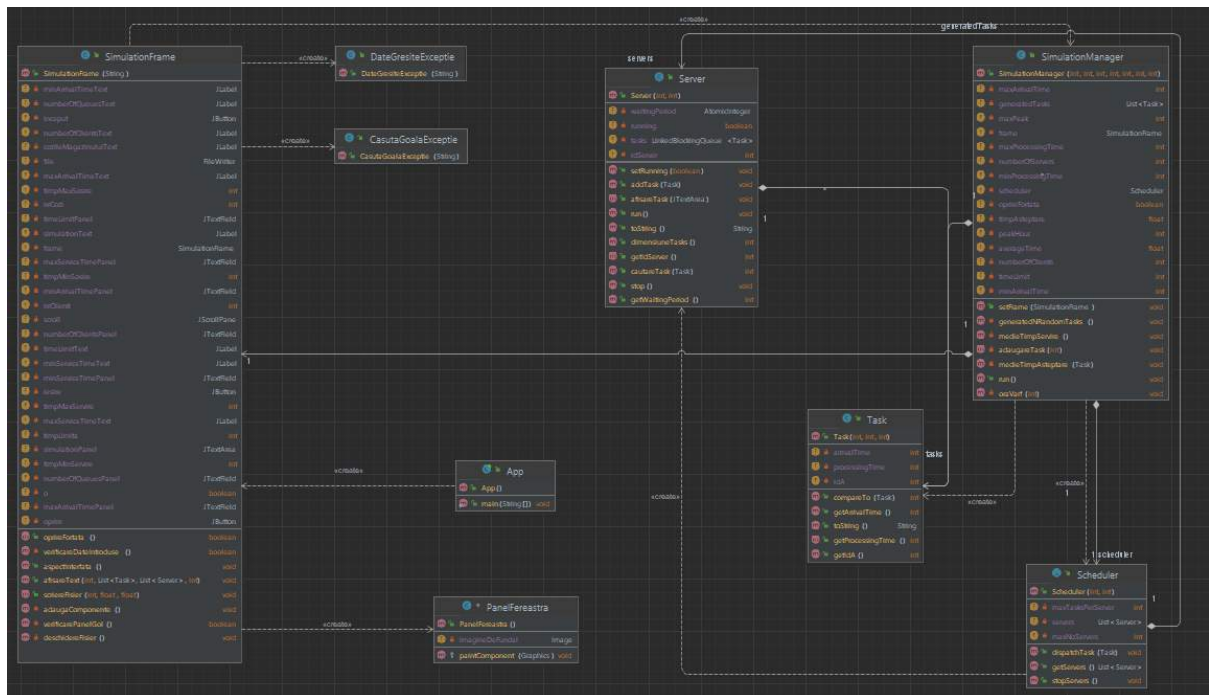


Diagrama de clase:



Unified Modeling Language (UML) este un limbaj standard de modelare utilizat în ingineria software pentru a reprezenta grafic și comunica modele conceptuale ale sistemelor software. UML oferă un set de diagrame grafice standardizate pentru a descrie diferite aspecte ale sistemelor software, inclusiv diagrame de clase, diagrame de secvențe, diagrame de activități și diagrame de stări. Aceste diagrame sunt concepute pentru a fi ușor de înțeles atât de către dezvoltatorii software cât și de către persoanele non-tehnice.

UML este foarte util pentru a detecta probleme și a identifica riscuri în timpul fazei de proiectare a sistemului. Prin vizualizarea modelelor UML, putem identifica posibile probleme în design sau interacțiuni defectuoase între componentele sistemului. Aceste probleme pot fi apoi corectate înainte ca sistemul să fie construit și implementat, reducând astfel costurile și timpul de dezvoltare.

c. Cazuri de utilizare și scenarii:

Cazuri de utilizare:

- ❖ Use Case: Setup simulare
- ❖ Actor Principal: Utilizator

Date de intrare:

- limita de timp
- timp minim servire
- timp maxim servire
- timp minim sosire
- timp maxim sosire
- numarul de clienti
- numarul de cozi

Date de iesire:

- afișarea evoluției cozilor în timp real

Scenariul principal de succes:

- Utilizatorul introduce valorile pentru: numărul de clienți, numărul de cozi, intervalul de simulare, timpul minim și maxim de sosire și timpul minim și maxim de serviciu.
- Utilizatorul apasă butonul de start care va face și validarea datelor de intrare.
- Se va afișa în timp real evoluția cozilor și a clienților.

Scenariu alternativ:

- Utilizatorul introduce valori invalide pentru parametrii de configurare ai aplicației.
- Aplicația afișează un mesaj de eroare și solicită utilizatorului să introducă valori valide.
- Scenariul revine la pasul 1.

3. PROIECTARE


Proiectarea este una dintre cele mai importante etape în dezvoltarea oricărui proiect software, inclusiv în dezvoltarea aplicației Java descrisă în tema dată. Aceasta implică stabilirea structurii generale a aplicației, a modului în care componentele sale interacționează între ele și a modului în care datele sunt stocate și procesate.

Am implementat această aplicație cu ajutorul a noua clase și am folosit următoarele structuri de date: `LinkedBlockingQueue`, `AtomicInteger` și `List`.

Clasele sunt organizate în următoarele pachete:

- exception - conține clasa `CasutaGoalaExceptie` si `DateGresiteExceptie`
- interfata - conține clasa `SimulationFrame` si `PanelFereastră`
- model - conține clasa `Server` si `Task`
- utils - conține clasa `App`
- logic - conține clasa `Scheduler` si `SimulationManager`

Descrierea claselor:

 Clasa "Server":

- reprezintă un server în cadrul unui sistem de cozi. Are un identificator, o coadă de sarcini (reprezentate de obiecte de tipul "Task"), o perioadă de așteptare (măsurată în secunde), și poate să ruleze pe un fir de execuție

propriu, prelucrând sarcinile din coadă. Metodele sale includ adăugarea de sarcini în coadă, afișarea sarcinilor din coadă, oprirea firului de execuție, obținerea sarcinilor din coadă, obținerea dimensiunii coadei și a perioadei de așteptare.

✚ Clasa "Task":

- reprezintă o sarcină în cadrul unui sistem de cozi. Are un moment de sosire, un timp de procesare (măsurat în secunde) și un identificator. Implementează interfața "Comparable" pentru a putea fi sortată după momentul de sosire.

✚ Clasa "Scheduler":

- este responsabilă de gestionarea mai multor servere și de distribuirea sarcinilor către acestea. Aceasta are o listă de servere și variabile care stochează numărul maxim de servere și sarcinile maxime per server. Construcția clasei creează și inițializează fiecare server și pornește firul de execuție al fiecărui server. Clasa Scheduler are o metodă dispatchTask care adaugă o sarcină la serverul cu cel mai mic timp de așteptare și o altă metodă de obținere a listei de taskuri.

✚ Clasa "SimulationManager":

- este responsabilă de gestionarea simulării în ansamblu. Aceasta primește datele de intrare de la interfața utilizatorului și creează obiecte Task aleatorii pe baza acestor date. De asemenea, creează și inițializează obiectul Scheduler și începe simularea prin intermediul metodei sale run(). Clasa SimulationManager are, de asemenea, metode auxiliare pentru calcularea orei de vârf, a mediei timpului de așteptare și a mediei timpului de servire. Această clasă afișează, de asemenea, datele de simulare curente pe interfața utilizatorului și scrie rezultatele finale într-un fișier.

✚ Clasa "SimulationFrame":

- este o implementare a unei interfețe grafice pentru un program de simulare. Ea conține diverse componente, cum ar fi etichete, câmpuri de text, butoane și un panou de text derulant, toate aranjate într-un layout specific. Clasa este responsabilă pentru crearea și afișarea interfeței grafice, precum și pentru gestionarea intrării utilizatorului și a rulării simulării.

✚ Clasa "PanelFereastra":

- este utilizată pentru a crea și afișa un panou cu o imagine de fundal. Mai precis, clasa definește un panou personalizat care are o imagine de fundal setată și suprascrive metoda paintComponent pentru a desena imaginea pe panou.

✚ Clasa "CasutaGoalaExceptie":

- reprezintă o excepție care este aruncată atunci când o casetă din panoul de interfață este goală și se încearcă accesarea conținutului acesteia. Constructorul clasei primește un mesaj de eroare sub formă de șir de caractere și îl transmite la constructorul clasei de bază folosind cuvântul cheie "super".

✚ Clasa "DateGresiteExceptie":

- reprezintă o excepție definită de utilizator care este aruncată atunci când datele introduse în program nu sunt valide sau sunt gresite. Constructorul clasei primește un mesaj de eroare sub formă de șir de caractere și îl transmite la constructorul clasei de bază folosind cuvântul cheie "super".

✚ Clasa "App":

- este o clasă de pornire a aplicației și conține metoda principală "main". Aceasta creează un obiect SimulationFrame, care este clasa care reprezintă fereastra aplicației și îi oferă un titlu.

* Utilizarea excepțiilor personalizate ajută la creșterea robusteții și fiabilității codului, permitând gestionarea și semnalarea erorilor specifice în mod corespunzător, în loc de a trata toate erorile într-un mod general.

4. IMPLEMENTARE

a. Aspect interfață:



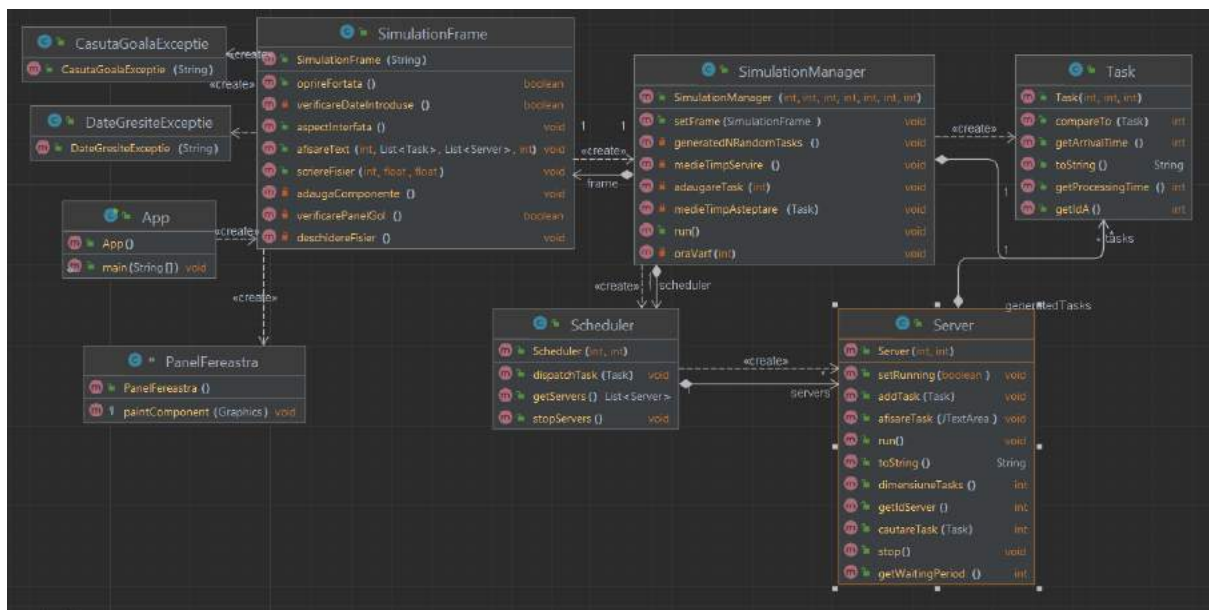
The screenshot shows a Windows-style application window titled 'Gestionare CLIENTI-COZI'. The main title 'Cozi la magazin' is centered at the top. Below it, the word 'Simularea:' is displayed. On the left side, there are seven input fields with corresponding labels: 'Limita de timp:', 'Timp minim servire:', 'Timp maxim servire:', 'Timp minim sosire:', 'Timp maxim sosire:', 'Numar de clienti:', and 'Numar de cozi:'. To the right of these fields is a large, empty rectangular box for simulation results. At the bottom of the window, there are three buttons: 'START', 'STOP', and 'EXIT'.

Interfața grafică constă din mai multe etichete, fiecare afișând un mesaj text, și câmpuri de text corespunzătoare în care utilizatorul poate introduce date, cum ar fi limita de timp, timpul minim și maxim de serviciu și numărul de clienți și cozi. Rezultatele simulării sunt afișate într-un panou de text, care este cuprins într-un panou derulant pentru a permite vizualizarea ușoară.

Clasa are și trei butoane: "START", "STOP" și "EXIT". Butonul "START" declanșează începerea rulării simulării, în timp ce butonul "STOP" oprește forțat simularea. Butonul "EXIT" închide interfața grafică și termină programul. Clasa folosește și alte clase, inclusiv Scheduler, SimulationManager, Server, Task și diverse clase de excepție. Aceste clase gestionează logica efectivă a simulării și orice erori care pot apărea în timpul simulării.

În ansamblu, clasa SimulationFrame servește drept punct principal de intrare pentru programul de simulare și oferă o interfață grafică convenabilă pentru utilizatori pentru a interacționa cu programul.

b. Implementare metodelor claselor:



Acest program implementeaza o simulare a unui sistem de cozi cu mai multe servere, unde sarcinile (clientii) ajung la momente aleatoare și cu timpi de procesare aleatorii, și sunt asigurate serverelor pentru a fi procesate.

A. App

```
public class App {  
    public static void main(String[] args) {  
        SimulationFrame f=new SimulationFrame( title: "Gestionare CLIENTI-COZI");  
    }  
}
```

În această clasă, atunci când programul este pornit, metoda main() instantiază un obiect SimulationFrame cu numele "Gestionare CLIENTI-COZI" și îl afișează pe ecran.

B. Server

- Componente: tasks(o lista, *LinkedBlockingQueue* <>), waitingPeriod(număr atomic, *AtomicInteger*), idServer(variabilă de tipul *int* care reprezintă numărul serverului, toate fiind distincte), running(variabilă de tip *boolean* care stabilește dacă firul de execuție rulează sau nu).

-Metode:

- Constructorul

```
public Server(int maxTasksPerServer, int idServer) {  
    tasks = new LinkedBlockingQueue<>();  
    waitingPeriod = new AtomicInteger( initialValue: 0);  
    running = true;  
    this.idServer=idServer;  
}
```


- cautareTask

```
public int cautareTask(Task task){
    for (Task t:tasks) {
        if(t.getIdA()==task.getIdA()){
            return 1;
        }
    }
    return 0;
}
```

Metoda "cautareTask" verifică dacă un anumit obiect "Task" este deja în coada de așteptare. Dacă obiectul "Task" se găsește deja în coadă, returnează valoarea 1, altfel returnează valoarea 0.

- dimensiuneTasks

```
public int dimensiuneTasks(){
    return tasks.size();
}
```

Metoda "dimensiuneTasks" returnează dimensiunea listei obiectelor de tip "Task".

- afisareTask

```
public void afisareTask(JTextArea text){
    for (Task ts:tasks) {
        text.append(ts.toString() + " ");
    }
    text.append("\n");
}
```

Metoda "afisareTask" afișează informațiile despre toate obiectele "Task" din coadă pe o componentă JTextArea dată ca parametru.

- addTask

```
public void addTask(Task newTask) {
    tasks.add(newTask);
    waitingPeriod.set(waitingPeriod.get()+newTask.getProcessingTime());
}
```

Metoda "addTask" adaugă un obiect "Task" la coada de așteptare și actualizează perioada de așteptare în funcție de timpul de procesare al obiectului "Task".

- run

```
public void run(){
    //process task
    while(running) {
        try {
            if(tasks.size()!=0){
                Task currentTask = tasks.element();
                Thread.sleep( millis: currentTask.getProcessingTime()*1000L);
                waitingPeriod.set(waitingPeriod.get()-currentTask.getProcessingTime());
                tasks.remove();
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Metoda "run" reprezintă comportamentul thread-ului serverului. Aceasta procesează obiectele de tipul "Task" din coada de așteptare în ordinea în care acestea au fost adăugate, folosind metoda "Thread.sleep" pentru a

simula timpul de procesare al fiecărui obiect de tipul "Task". După procesarea obiectului, acesta este eliminat din coadă și perioada de așteptare a serverului este actualizată.

- stop

```
public void stop() { running = false; }
```

Metoda "stop" oprește execuția thread-ului serverului prin setarea valorii variabilei "running" la "false".

- alte metode ajutatoare gettere si settere: getWaitingPeriod(), getIdServer(), setRunning(boolean running), toString().

Metoda "getWaitingPeriod" returnează perioada totală de așteptare a serverului, care este calculată prin adunarea timpului de procesare al tuturor obiectelor "Task" din coada de așteptare.

Metoda "toString" convertește coada de așteptare a obiectelor "Task" la un șir de caractere.

Metoda "getIdServer" returnează id-ul serverului.

Metoda "setRunning" actualizează starea variabilei "running".

C. Task

- Componente: arrivalTime(variabilă de tipul *int* care reprezintă timpul de sosire al clientului), processingTime(variabilă de tipul *int* care reprezintă timpul de servire al clientului), idA(variabilă de tipul *int* care reprezintă id-ul unic al clientului).

-Metode:

- compareTo

```
@Override
public int compareTo(Task otherTask){
    if(this.arrivalTime<otherTask.arrivalTime){
        return -1;
    }else if(this.arrivalTime==otherTask.arrivalTime){
        return 0;
    }
    else {
        return 1;
    }
}
```

Metoda "compareTo" este o metoda implementată din interfața Comparable și folosită pentru a sorta clientii în ordinea sosirii lor. Aceasta compară timpul de sosire al clientului curent cu timpul de sosire al altui client primit ca parametru și returnează -1, 0 sau 1 în funcție de relația dintre acestea (mai devreme, același timp, mai târziu).

- alte metode ajutatoare cum ar fi gettere si settere: getProcessingTime(), getArrivalTime(), toString(), getIdA().

Metoda "getProcessingTime" returnează timpul necesar procesării clientului.

Metoda "getArrivalTime" returnează timpul de sosire al clientului.

Metoda "toString" returnează o reprezentare sub formă de șir de caractere a clientului, ce conține ID-ul, timpul de sosire și timpul necesar servirii.

Metoda "getIdA" returnează ID-ul clientului.

D. Scheduler

- Componente: servers(o lista *List<>* care reprezintă cozile), maxNoServers(variabilă de tipul *int* care reprezintă numărul de cozi).

-Metode:

- Constructorul:

```
public Scheduler(int maxNoServers, int maxTasksPerServer){  
  
    this.maxNoServers=maxNoServers;  
    this.maxTasksPerServer=maxTasksPerServer;  
    this.servers=new ArrayList<>();  
    for(int i=0;i<maxNoServers;i++){  
        Server server=new Server(maxTasksPerServer,i);  
        Thread thread =new Thread(server);  
        thread.start();  
        servers.add(server);  
    }  
}
```

Scheduler(int maxNoServers, int maxTasksPerServer): constructorul clasei, care creează maxNoServers obiecte de tip Server, fiecare având capacitatea maximă de procesare de maxTasksPerServer sarcini. Aceste obiecte sunt pornite în thread-uri separate.

- dispatchTask:

```
public void dispatchTask(Task t){  
    int min=1000000;  
    int pozMin=0;  
    for(int i=0;i<maxNoServers;i++){  
        if (servers.get(i).getWaitingPeriod()<min) {  
            min = servers.get(i).getWaitingPeriod();  
            pozMin = i;  
        }  
    }  
    servers.get(pozMin).addTask(t);  
}
```

Metoda "dispatchTask " atribuie un client unei cozi disponibile cu cel mai scurt timp de așteptare. Mai întâi se determină coada cu cel mai scurt timp de așteptare prin parcurgerea listei de servere și determinarea serverului cu timpul minim de așteptare. Apoi clientul este adăugat în coada.

- alte metode cum ar fi gettere: getServers().

Metoda "getServers " returnează lista de obiecte de tip Server create în constructor.

E. SimulationManager

- Componente: generatedTasks(o lista *List<>* care reprezintă clientii), timeLimit(variabilă de tipul *int* care reprezintă timpul de executie), maxProcessingTime(variabilă de tipul *int* care reprezintă limita de timp maxima in care se serveste clientul), minProcessingTime(variabilă de tipul *int* care reprezintă limita de timp minima in care se serveste clientul), maxArrivalTime(variabilă de tipul *int* care reprezintă limita de timp maxima la carea junge clientul), minArrivalTime(variabilă de tipul *int* care reprezintă limita de timp minima la care ajunge clientul), numberOfServers(variabilă de tipul *int* care reprezintă numarul de cozi), numberOfClients(variabilă de tipul *int* care reprezintă numarul de clienti), scheduler(obiect de tip *Scheduler* prin care facem legatura cu clasa), frame(obiect de tip *SimulationFrame* prin care facem legatura cu clasa), oprireFortata(variabila *boolean* pe care o folosim in functia run ca sa facem legatura cu butonul STOP din interfata), maxPeak(variabila de tipul *int* care ne ajuta sa gasim cea mai mare ora pe care o sa atribuim orei de varf), peakHour(variabila de tipul *int* care reprezinta ora de varf), averageTime(variabila de tipul *float* care reprezinta media timpului de servire), timpAsteptare(variabila de tipul *float* care reprezinta media timpului de asteptare).

-Metode:

- Constructorul:

```
public SimulationManager(int timeLimit, int maxProcessingTime, int minProcessingTime, int maxArrivalTime, int minArrivalTime, int numberOfServers, int numberOfClients) {
    this.timeLimit = timeLimit;
    this.maxProcessingTime = maxProcessingTime;
    this.minProcessingTime = minProcessingTime;
    this.maxArrivalTime = maxArrivalTime;
    this.minArrivalTime = minArrivalTime;
    this.numberOfServers = numberOfServers;
    this.numberOfClients = numberOfClients;

    scheduler = new Scheduler(numberOfServers, maxTasksPerServer 100);
    generatedTasks = new ArrayList<>();
    generatedNRandomTasks();
}
```

Metoda “SimulationManager- primește parametrii necesari pentru inițializarea simulării. Acesta initializează obiectul Scheduler cu numărul de cozi și numărul maxim de clienti pe cozi, generează numberOfClients clienti aleatori și îi sortează în funcție de momentul de sosire.

- generatedNRandomTasks:

```
private void generatedNRandomTasks(){
    for(int i=0;i<numberOfClients;i++){
        int arrivalTime=(int)(Math.random()*(maxArrivalTime-minArrivalTime)+minArrivalTime);
        int processingTime=(int)(Math.random()*(maxProcessingTime-minProcessingTime)+minProcessingTime);

        Task task=new Task(arrivalTime,processingTime,i);
        generatedTasks.add(task);
    }
    Collections.sort(generatedTasks);
}
```

Metoda “generatedNRandomTasks” - generează clienti aleatori pentru simulare în funcție de parametrii specificați în constructor.

- oraVarf:

```
private void oraVarf(int currentTime) {
    int suma = 0;
    for (Server s : scheduler.getServers()) {
        suma = suma + s.dimensiuneTasks();
    }
    if (suma > maxPeak) {
        maxPeak = suma;
        peakHour = currentTime;
    }
}
```

Metoda “oraVarf” - calculează ora de vârf în care se află cel mai mare număr de clienți în așteptare, și actualizează valoarea maximă a acestui număr și ora corespunzătoare.

- medieTimpServire:

```
private void medieTimpServire(){
    for (Task t:generatedTasks) {
        averageTime=averageTime+t.getProcessingTime();
    }
    averageTime=averageTime/numberOfClients;
}
```

Metoda “medieTimpServire” - calculează media timpilor de procesare ai clientilor generati și o stochează în variabila averageTime.

- medieTimpAsteptare:

```
private void medieTimpAsteptare(Task t){
    for (Server s:scheduler.getServers()) {
        if(s.cautareTask(t)==1){
            timpAsteptare=timpAsteptare+s.getWaitingPeriod();
            timpAsteptare=timpAsteptare-t.getProcessingTime();
        }
    }
}
```

Metoda “medieTimpAsteptare” - calculează media timpului de așteptare pentru un client t, în funcție de cozile care in care se afla.

- adaugareTask:

```
private void adaugareTask(int timpCurent){
    for(int i=0;i<generatedTasks.size();i++){
        Task task = generatedTasks.get(i);
        if(task.getArrivalTime()==timpCurent) {
            scheduler.dispatchTask(task);
            medieTimpAsteptare(task);
            generatedTasks.remove(i);
            i--;
        }
    }
}
```

Metoda “adaugareTask”- adaugă clientii care au ajuns la momentul de sosire curent timpCurent în coada serverelor.

- run:

```
@Override
public void run() {
    int currentTime = 0;
    medieTimpServire();
    while(currentTime <= timeLimit && !oprireFortata){
        adaugareTask(currentTime);
        oraVarf(currentTime);
        frame.afisareText(currentTime, generatedTasks, scheduler.getServers(), numberOfServers);
        try{
            Thread.sleep( millis: 1000);
        }catch (InterruptedException ex){
            ex.printStackTrace();
        }
        currentTime++;
        oprireFortata = frame.oprireFortata();
    }
    for (Server s:scheduler.getServers()) {
        s.setRunning(false);
    }
    timpAsteptare=timpAsteptare/numberOfClients;
    frame.scriereFisier(peakHour, averageTime, timpAsteptare);
}
```

Metoda “run” - reprezintă firul de execuție al simulării. Aceasta actualizează timpul curent, adaugă clientii la coada serverelor, calculează ora de vârf, afișează starea curentă a sistemului în fereastra de simulare, și actualizează variabila oprireFortata în funcție de comanda utilizatorului din fereastra de simulare. După terminarea simulării, această metodă oprește serverele și scrie rezultatele într-un fișier.

- alte metode ajutatoare, cum ar fi settere: setFrame;

Metoda “setFrame” - setează obiectul frame ce reprezintă fereastra de simulare.

F. PanelFereastra

- Componente: imagineDeFundal(de tipul *Image* in care am stocat imaginea de fundal a Interfetei grafice)

-Metode:

- Constructor

```
public PanelFereastra() {
    ImageIcon img = new ImageIcon( filename: "Purple.jpg");
    imagineDeFundal = img.getImage();
}
```

Metoda constructor "PanelFereastra()" - creează o instanță a clasei și setează imaginea de fundal a panoului. Aceasta utilizează clasa ImageIcon pentru a încărca imaginea dintr-un fișier local "Purple.jpg" și o stochează în variabila imagineDeFundal.

- paintComponent

```
protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.drawImage(imagineDeFundal, x: 0, y: 0, getWidth(), getHeight(), observer: this);
}
```

Metoda "paintComponent" - este suprascrisă din clasa JPanel și este responsabilă de desenarea componentelor panoului. Metoda primește ca parametru un obiect de tip Graphics care este utilizat pentru a desena imaginea de fundal pe panou. În această metodă, se apelează metoda "super.paintComponent(g)" pentru a desena componentele de bază ale panoului, iar apoi se utilizează metoda "g.drawImage()" pentru a desena imaginea de fundal în întregul panou, cu dimensiunile date de "getWidth()" și "getHeight()".

G. SimulationFrame

- Componente: 9 JLabel, 7 JTextField, 1 JTextArea, 3 JButton, frame(de tipul *SimulationFrame* care reprezinta referinta catre interfata), 1 JScrollPane, un fisier in care scriem datele, o variabila boolean o cu care stabilim daca oprim executia fortat, si 7 variabile int care reprezinta datele pe care le vom introduce in interfata si care vor fi afisate in fisier.

-Metode:

- aspectInterfata
- Constructorul
- adaugaComponente
- verificarePanelGol

```
private boolean verificarePanelGol() throws CasutaGoalaExceptie {
    if ((timeLimitPanel.getText().isEmpty() || (minArrivalTimePanel.getText().isEmpty() || (maxArrivalTimePanel.getText().isEmpty() || 
        throw new CasutaGoalaExceptie("Completati panelul care este gol!");
    }
    return false;
}
```

- verificareDateIntroduce

```
private boolean verificareDateIntroduce() throws DateGresiteExceptie {
    try {
        if (Integer.parseInt(timeLimitPanel.getText()) < 0 ||
            Integer.parseInt(minServiceTimePanel.getText()) < 0 ||
            Integer.parseInt(maxServiceTimePanel.getText()) < 0 ||
            Integer.parseInt(minArrivalTimePanel.getText()) < 0 ||
            Integer.parseInt(maxArrivalTimePanel.getText()) < 0 ||
            Integer.parseInt(numberOfClientsPanel.getText()) < 0 ||
            Integer.parseInt(numberOfQueuesPanel.getText()) < 0 {
            throw new DateGresiteExceptie("Nu ati introdus datele corect in panel!");
        }
    } catch (NumberFormatException ex1) {
        throw new DateGresiteExceptie("Nu puteti introduce litere!");
    }
    return false;
}
```

- o afisareText

```
public void afisareText(int timp, List<Task> taskuri, List<Server> servere, int nrServere) {
    simulationPanel.append("Timp: "+timp + "\n");
    simulationPanel.append("In asteptare:");
    for (Task t:taskuri) {
        simulationPanel.append(t.toString() + " ");
    }
    simulationPanel.append("\n");

    for(int i=0; i<nrServere; i++){
        simulationPanel.append("Server[" + (i+1) + "]:");
        for (Server s:servere) {
            if(s.getIdServer()==i){
                s.afisareTask(simulationPanel);
            }
        }
    }
    simulationPanel.append("\n");
}
```

- o scriereFisier

```
public void scriereFisier(int peakHour, float timpServire, float timpAsteptare){
    simulationPanel.append("Ora de varf: " + peakHour + "\n");
    simulationPanel.append("Timp mediu servire: " + timpServire + "\n");
    simulationPanel.append("Timp mediu asteptare: " + timpAsteptare);
    try {
        file.write(simulationPanel.getText());
        file.close();
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}
```

- o deschidereFisier

```
private void deschidereFisier() {
    try{
        file=new FileWriter( fileName: "FisierLog.txt");
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}
```

- o oprireFortata

```
public boolean oprireFortata(){
    return o;
}
```

H. CasutaGoalaExceptie

-Metode:

- o Constructorul

```
public CasutaGoalaExceptie(String message) { super(message); }
```

Constructorul clasei primește un mesaj de eroare care va fi afișat atunci când această excepție este aruncată. Acest mesaj poate fi personalizat și transmis utilizatorului pentru a-l ajuta să înțeleagă ce s-a întâmplat și ce trebuie făcut pentru a remedia problema.

I. DateGresiteExceptie

-Metode:

- Constructorul

```
public DateGresiteExceptie(String message) { super(message); }
```

Constructorul clasei primește un mesaj de eroare care va fi afișat atunci când această excepție este aruncată. Acest mesaj poate fi personalizat și transmis utilizatorului pentru a-l ajuta să înțeleagă ce s-a întâmplat și ce trebuie făcut pentru a remedia problema.

5. REZULTATE

Pentru testarea aplicației am folosit datele de intrare din tabelul de mai jos:

Test 1	Test 2	Test 3
N = 4 Q = 2 $t_{simulation}^{MAX} = 60$ seconds $[t_{arrival}^{MIN}, t_{arrival}^{MAX}] = [2, 30]$ $[t_{service}^{MIN}, t_{service}^{MAX}] = [2, 4]$	N = 50 Q = 5 $t_{simulation}^{MAX} = 60$ seconds $[t_{arrival}^{MIN}, t_{arrival}^{MAX}] = [2, 40]$ $[t_{service}^{MIN}, t_{service}^{MAX}] = [1, 7]$	N = 1000 Q = 20 $t_{simulation}^{MAX} = 200$ seconds $[t_{arrival}^{MIN}, t_{arrival}^{MAX}] = [10, 100]$ $[t_{service}^{MIN}, t_{service}^{MAX}] = [3, 9]$

Asfel am introdus trei seturi de teste, iar rezultatele lor le-am Salvat pe fiecare cate intr-un fisier log text, numite FisierLog1.txt, FisierLog2.txt si FisierLog3.txt.

Datele din aceste fisiere sunt urmatoarele:

FisierLog1.txt:

```
Ora de varf:5
Timp mediu servire:2.5
Timp mediu asteptare:0.0
```

FisierLog2.txt:

```
Ora de varf:38
Timp mediu servire:3.16
Timp mediu asteptare:1.02
```

FisierLog3.txt:

```
Ora de varf:99
Timp mediu servire:5.408
Timp mediu asteptare:89.67
```

6. CONCLUZII

În concluzie, aplicația de management a cozilor a adresat cu succes problema minimizării timpului de așteptare pentru clienți în sistemele bazate pe cozi. Prin simularea sosirii și serviciului a N clienți în Q cozi, aplicația a identificat timpul minim de așteptare pentru fiecare client și l-a asignat cozii cu cel mai scurt timp de așteptare. De asemenea, aplicația a urmărit timpul total petrecut de fiecare client în cozi și a calculat timpul mediu de așteptare.

Din acest proiect, am învățat că optimizarea managementului cozilor necesită un echilibru între numărul de servere (cozi) și numărul de task-uri (clienți). Adăugarea mai multor servere poate reduce timpul de așteptare. Pe de altă parte, reducerea numărului de servere poate duce la timpi mai lungi de așteptare și o calitate mai scăzută a serviciului.

Dezvoltările viitoare posibile pentru aplicația de management al cozilor includ adăugarea de mai multe funcții, cum ar fi capacitatea de a alege strategia de selecție a cozii (de exemplu, prima coadă disponibilă sau coada cu cel mai mic timp de așteptare), stabilirea unui prag de timp de așteptare pentru clienți (de exemplu, dacă timpul de așteptare depășește pragul, clientul va părăsi sistemul) și stabilirea limitelor pentru capacitatea cozilor (de exemplu, dacă capacitatea cozii este atinsă, clienții vor fi redirecționați către alte cozi).

7. BIBLIOGRAFIE

https://dsrl.eu/courses/pt/materials/PT2023_A2.pdf

https://dsrl.eu/courses/pt/materials/PT2023_A2_S1.pdf

https://dsrl.eu/courses/pt/materials/PT2023_A2_S2.pdf

<https://docs.oracle.com/javase/tutorial/uiswing/index.html>

<https://www.geeksforgeeks.org/queue-data-structure/>

<https://www.geeksforgeeks.org/atomic-variables-in-java-with-examples/>

<https://uncoded.ro/tratarea-exceptiilor-in-limbajul-java/>