DOCUMENTATIE

TEMA 2

NUME STUDENT: Rujac Roxana
GRUPA: 30224

Cuprins:

1.Obiectivul temei

1.1 Obiectivul principal

Obiectivul principal al acestei aplicații în Java este de a proiecta și implementa un sistem de gestiune a cozilor folosind tehnici de multithreading. Această aplicație va simula un sistem de cozi în care N clienți sosesc pentru serviciu, intră în Q cozi, așteaptă, sunt serviți și în cele din urmă părăsesc cozile. Obiectivul principal implică, de asemenea, calculul timpului mediu de așteptare, timpului mediu de serviciu și a orei de vârf.

1.2 Obiective secundare

- proiectarea interfetei utilizator (UI)
- lucrul cu fire de executie
- testarea aplicatiei

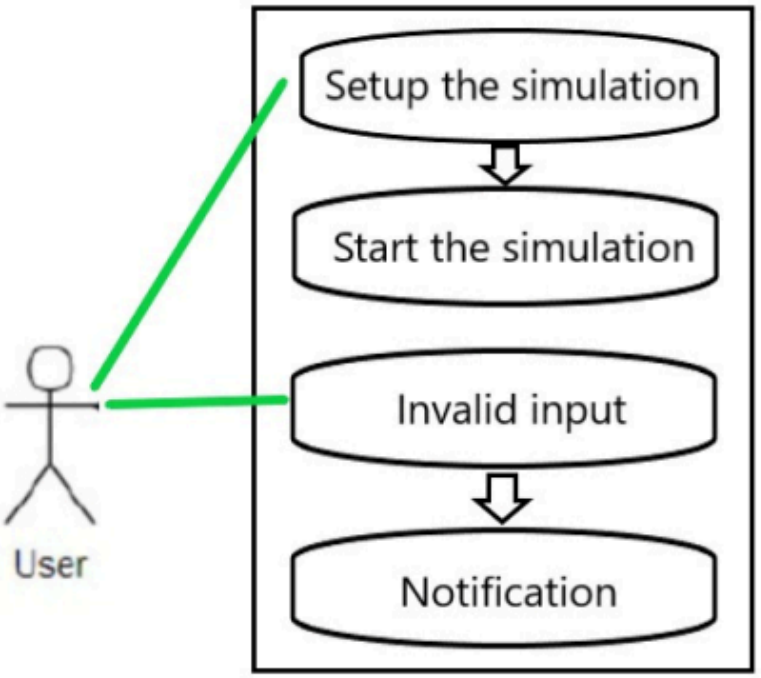
2.Analiza problemei, modelare, scenarii, cazuri de utilizare

2.1 Cerinte functionale

2.1.1 Clasificarea MoSCoW

- Must have – implementarea unui sistem de simulare a coziilor care să permită sosirea şi servirea clienţilor în mod concurent folosind multithreading
- Should have – verificari in caz de eroare (ex:introducrea unui string ce nu reprezinta un intreg, lasarea de field-uri goale, intervale de timp invalide)
- Could have - implementarea unei funcționalități de export a rezultatelor simulării format de fișiere .txt
- Won't have - implementarea unui sistem de autentificare şi autorizare pentru utilizatori

2.1.2 Diagrama use-case



2.1.3 Scenarii de use-case

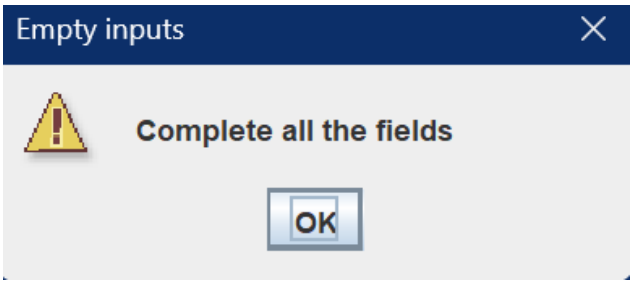
- Scenariu de utilizare: Configurare simulare
- Actor principal: Utilizatorul

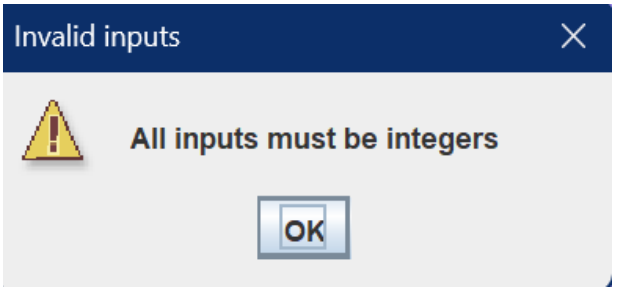
Scenariu principal de succes:

➢ 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 servire.
➢ Utilizatorul apasă pe butonul de validare a datelor de intrare.
➢ Aplicația validează datele și afișează un mesaj informativ utilizatorului pentru a începe simularea.

Secvență alternativă: Valori invalide pentru parametrii de configurare

➢ 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.





2.2 Cerinte non-functionale

- Programul este usor de utilizat si intuitiv
- Design simpu si compact
- Ofera un timp de raspuns rapid datorita operatiilor efectuate prin metoda multithreading
- Functioneaza fara a genera erori sau situatii exceptionale

3. Proiectare

3.1 Proiectarea OOP

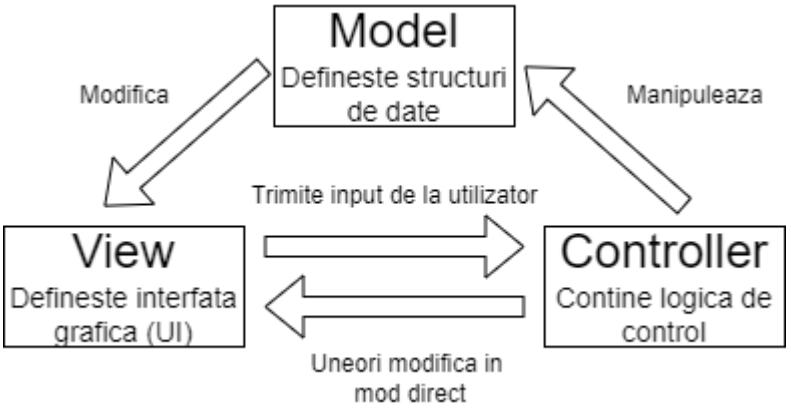
Codul respecta principiile OOP:

- incapsularea
- abstractizare
- mostenire
- polimorfism

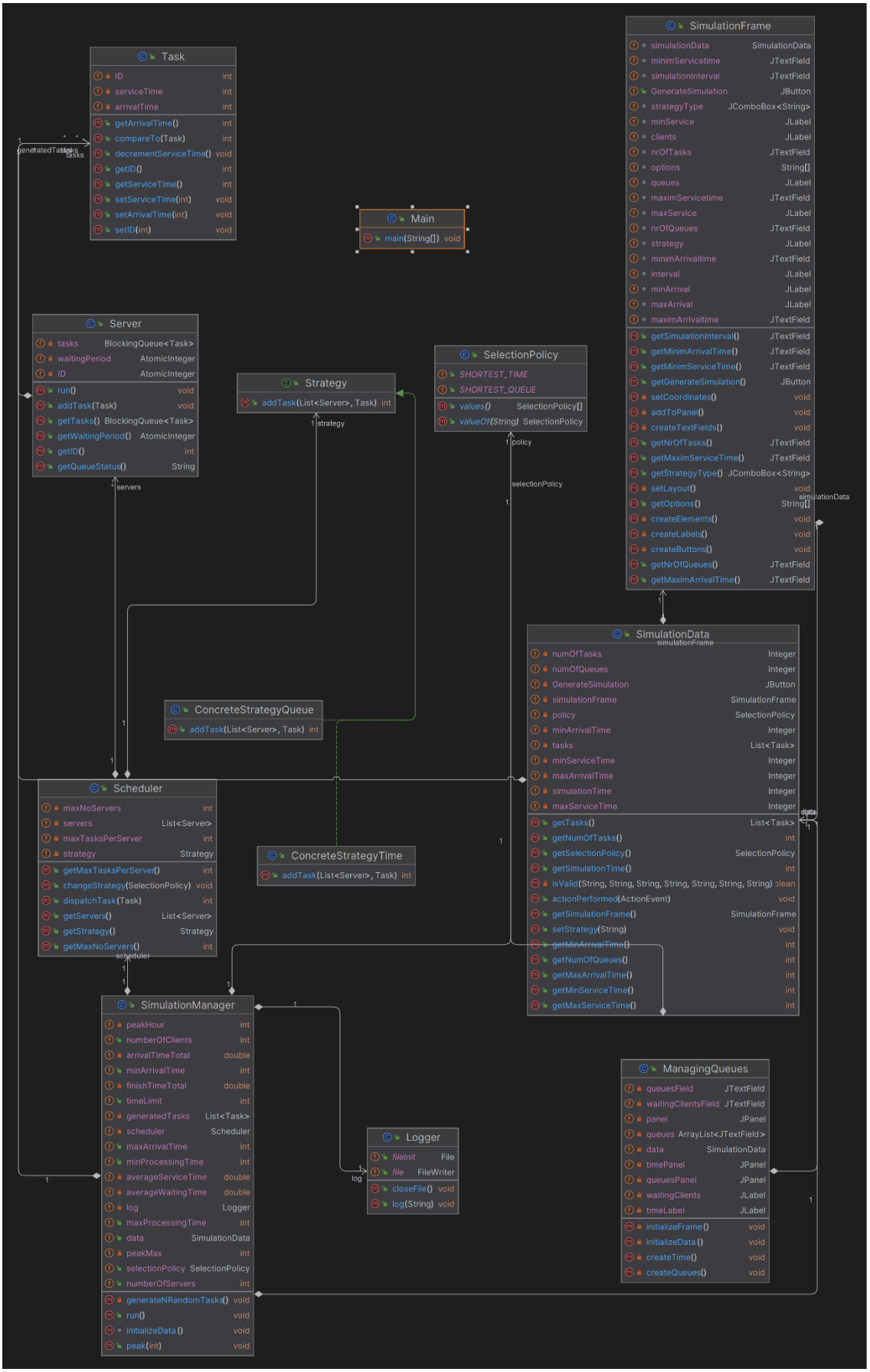
3.2 Diagrame UML

3.2.1 Diagrama UML a pachetelor

Am folosit o arhitectura de tip MVC (Model-View-Controller) pentru pachetele programului.



3.2.2 Digrama UML a claselor



3.3 Structuri de date folosite

- BlockingQueue<Task> pentru impementarea unei cozi (Server)
- List<Task> pentru clienții generați
- List<Task> pentru clientii adăugați în cozi
- Enum pentru selectarea strategiei (shortest time, shortest queue)

4. Implementare

4.1 Implementarea claselor

Clasele sunt organizate in 3 pachete, conform arhitecturii MVC. Pachetul “Model” contine clasele “‘Task” si “Server”. Pachetul “View” contine clasele “SimulationFrame”, “SimulationData”, iar pachetul “Controller” contine clasele “”.

Modelul conține clasele care definesc structurile de date care vor fi utilizate alături de unele dintre metodele lor. Aici sunt implementate Cozile, Clienții si clasa Logger care ajuta la scrierea in fisiere a rezultatelor simularii.. Singura logică în această parte este adăugarea clienților într-o coadă.

Vizualizarea conține clasele care definesc ceea ce este afișat: meniul de configurare și fereastra de simulare. De asemenea, reprezentarea pentru clienți este implementată în acest pachet.

Controlerul este "lipiciul" între Model și Vizualizare. Aici se află cea mai mare parte a logicii și controlează când, ce și dacă se fac modificări în interfața grafică și efectuează alte operații pentru a menține corectitudinea simulării.

1. Pachetul de clase Model

   Clasa Task conține date despre fiecare client individual și metode pentru a obține atributele private ale clasei, cum ar fi ID-ul, timpul de sosire, timpul de servire și timpul rămas.

   Clasa Server implementează interfața Runnable, astfel încât să aibă o metodă run(). În această metodă, se fac operații pe primul client din coadă. Clienții sunt stocați într-un ArrayBlockingQueue pentru a asigura siguranța în utilizarea thread-urilor. Primul client este eliminat dacă sarcina sa este completată. În caz contrar, se va decrementa timpul rămas al acestuia în coadă. De asemenea, din această clasă, aceste acțiuni sunt scrise în fișierul de jurnal (log file) cu ajutorul clasei Logger.

```java
public class Server implements Runnable{
    7 usages
    private BlockingQueue<Task> tasks;
    3 usages
    private AtomicInteger waitingPeriod;
    2 usages
    private static AtomicInteger ID = new AtomicInteger( initialValue: 0);

    1 usage
    public Server()
    {
        this.tasks = new LinkedBlockingQueue<>();
        this.waitingPeriod = new AtomicInteger();
        this.ID.incrementAndGet();
    }

    2 usages
    public void addTask(Task newTask) {
        tasks.add(newTask);
        // add service time for each new task in waiting period
        this.waitingPeriod.addAndGet(newTask.getServiceTime());
    }
```

2. Pachetul de clase Controller

Clasa SimulationManager: -Generează task-urile pentru simulare, inițializează politica de selecție pentru alegerea cozii optime, calculează ora de vârf a activității sistemului, implementează logica de simulare a sistemului de cozi, calculează timpul mediu de așteptare și înregistrează evenimentele finale, scriind intr-un fisier de tip text..

Clasa Scheduler: Inițializează lista de cozi, schimbă strategia pentru distribuirea sarcinilor, preia un task către o coada folosind strategia curentă si il adăugă la una dintreacestea,

Interfața Strategy: Selectează strategia dată de utilizator.

Clasa ConcreteStrategyTime: returneaza coada cu cel mai mic timp de așteptare.

Clasa ConcreteStrategyQueue: reyrneaza coada cu cea mai scurta lista de task-uri.

Clasa SelectionPolicy: Selectează strategia dorită de utilizator.

```
private void generateNRandomTasks() {
    generatedTasks = new ArrayList<>();
    for (int i = 0; i < numberOfClients; i++) {
        //generate a random client with the id = i
        Random random = new Random();
        int arrivalTime = random.nextInt(minArrivalTime, maxArrivalTime);
        int serviceTime = random.nextInt(minProcessingTime, maxProcessingTime);
        Task newTask = new Task(i, arrivalTime, serviceTime);

        //add it to the list of tasks
        generatedTasks.add(newTask);
        //add the service time for each client
        averageServiceTime += serviceTime;
        //add the arrival time for each client
        arrivalTimeTotal += arrivalTime;
    }

    //sort them by arrival time
    generatedTasks.sort(Comparator.comparingInt(Task::getArrivalTime));
    averageServiceTime /= numberOfClients;
}
```

3. Pachetul View

SimulationFrame: Această clasă afișează primul cadru al aplicației, adică prima vizualizare. Aici utilizatorul trebuie să introducă datele necesare pentru a începe simularea. Aceste date constau în: numărul de clienți, numărul de cozi, timpul maxim de simulare, timpul minim de sosire, timpul maxim de sosire, timpul minim de servire și timpul maxim de servire si strategia aleasa. Aceste date sunt transmise clasei SimulationData pentru validare.

SimulationData: Aceasta clasa preia datele introduse de utilizator in SimulationFrame, le valideaza, afiseaza mesare de eroare in cazul in care acestea nu sunt corespunzatoare si trimite mai departe datele spre pachetul Controller unde se intampla logica.

ManagingQueues: Al doilea cadru (vizualizare) al aplicației. Afișează evoluția în timp real a tuturor cozilor.

```
private void createLabels() {
    Font font = new Font( name: "Arial", Font.BOLD,  size: 12);

    JLabel titleLabel = new JLabel( text: "Simulation input:");
    titleLabel.setFont(new Font( name: "Arial", Font.BOLD,  size: 25));
    titleLabel.setForeground(new Color( r: 42,  g: 42,  b: 49));
    add(titleLabel);
    titleLabel.setBounds( x: 180,  y: 20,  width: 300,  height: 30);

    queues = new JLabel( text: "number of queues  = ");
    queues.setFont(font);
    queues.setForeground(new Color( rgb: 0x2A2A31));

    clients = new JLabel( text: "number of clients  = ");
    queues.setFont(font);
    queues.setForeground(new Color( rgb: 0x2A2A31));

    interval = new JLabel( text: "simulation interval  = ");
    queues.setFont(font);
    queues.setForeground(new Color( rgb: 0x2A2A31));
```

4. Clasa Main

   Acesta este fisierul Main al aplicatiei, care se ocupa de pornirea și afisarea interfetei utilizator pentru sistemul de simulare a cozilor.

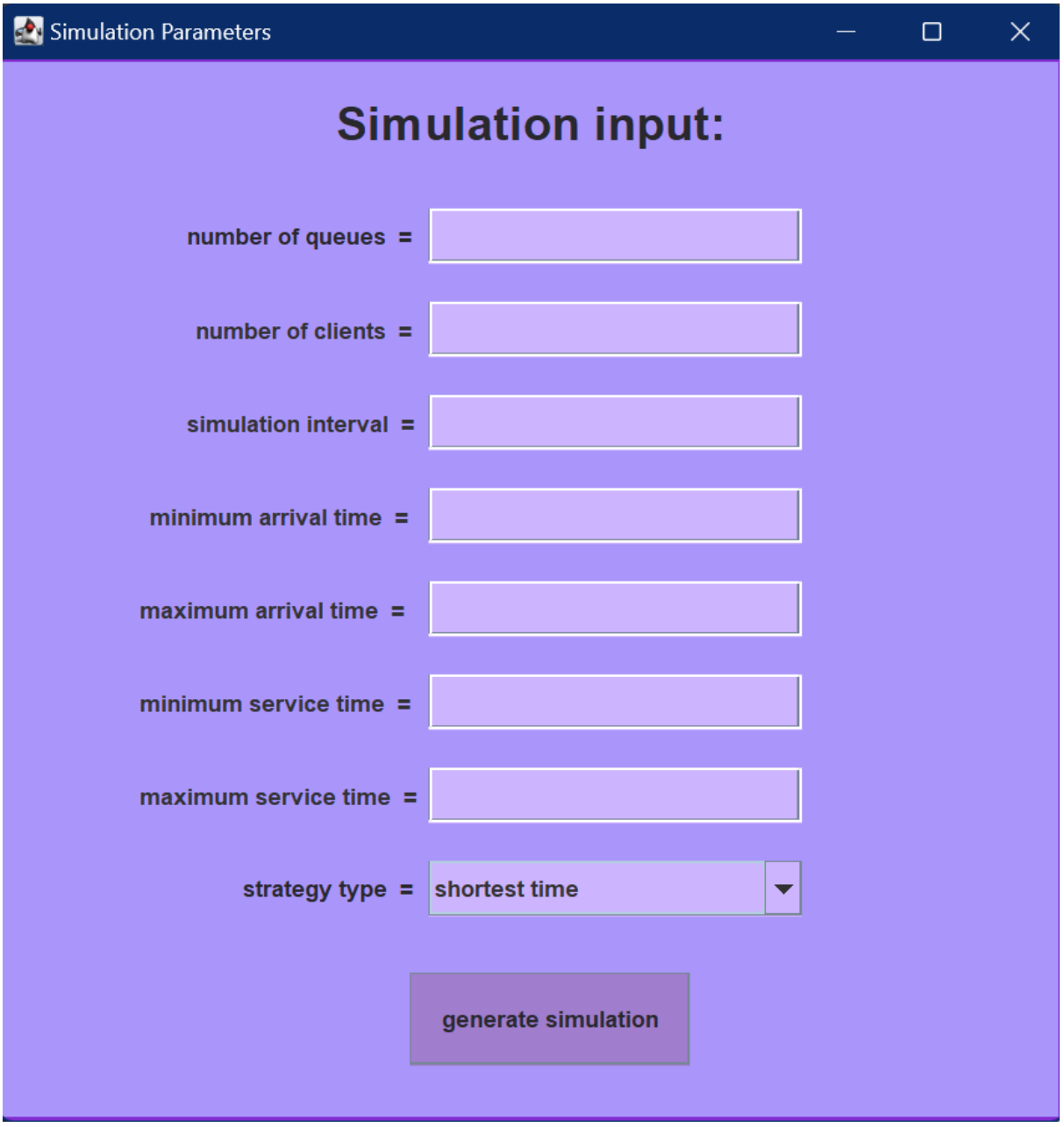
```
public class Main {
    public static void main(String[] args) {
        JFrame frame = new JFrame( title: "Simulation Parameters");
        frame.setContentPane(new SimulationFrame());
        frame.pack();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);

    }
}
```

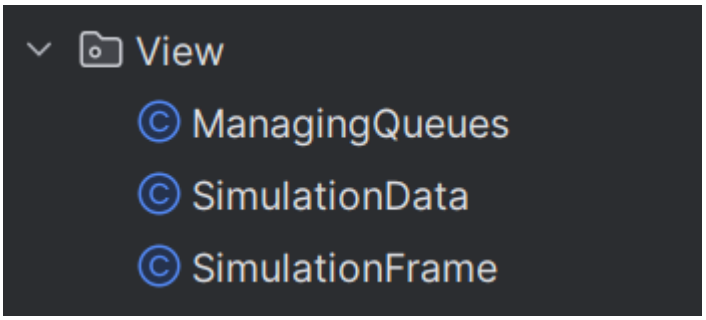
4.2 Implementarea Interfetei-Utilizator (GUI)

4.2.1 Prezentare generala

Interfata grafice este una simpla si intuitiva. In cele 7 casete text utizatorul poate introduce datele corespunzatoare, cat si un meniu din care poate selecta strategia de implementare dorita pentru a asigna clienti la cozi. De asemenea, exista si un buton de pornire a simulatii, butonul “generate simulation” din partea inferioara a ferestrei, la apasarea caruia se valideaza datele introduse si se porneste simularea.



4.2.2 Organizarea in pachetul “View” si implementarea propriu-zisa



Pachetul “View” contine clasele “SimulationFrame”, “SimulationData” si ManagingQueues.

Clasa “SimulationFrame” este responsabila pentru definirea și gestionarea interfetei grafice a aplicației.. Aceasta utilizeaza componente grafice din Java Swing pentru a crea o interfața utilizator intuitiva.

Clasa “SimulationData” este responsabila pentru gestionarea logica a aplicației, inclusiv validarea datelor introduse de utilizator și trasmiterea lor catre partea de logica a aplicatiei.

Clasa “ManagingQueues” contine o simulare in timp real a modului in care se creaza si gestioneaza cozile.

5. Rezultate

N = numărul de clienți generați
Q = numărul de cozi generate
tMAXsimualtion = intervalul de simulare (de la 1 până la tMAXsimulation)
tMINarrival = timpul minim de sosire
tMAXarrival = timpul maxim de sosire
tMINservice = timpul minim de servire
tMAXservice = timpul maxim de servire

Test 1
$N = 4$ $Q = 2$ $t_{simulation}^{MAX} = 60$ seconds $[t_{arrival}^{MIN}, t_{arrival}^{MAX}] = [2, 30]$ $[t_{service}^{MIN}, t_{service}^{MAX}] = [2, 4]$

```
Average service time : 2.25
Average waiting time : 2.25
Peek hour: 9
```

Test 2
N = 50 Q = 5 $t_{simulation}^{MAX}$ = 60 seconds $[t_{arrival}^{MIN}, t_{arrival}^{MAX}] = [2, 40]$ $[t_{service}^{MIN}, t_{service}^{MAX}] = [1, 7]$

```
Average service time : 4.4
Average waiting time : 9.32
Peek hour: 34
```

Test 3
N = 1000 Q = 20 $t_{simulation}^{MAX}$ = 200 seconds $[t_{arrival}^{MIN}, t_{arrival}^{MAX}] = [10, 100]$ $[t_{service}^{MIN}, t_{service}^{MAX}] = [3, 9]$

```
Average service time : 5.484
Average waiting time : 18.693
Peek hour: 99
```

6. Concluzii

Acest proiect m-a ajutat să descopăr noi funcționalități în limbajul de programare Java, cum ar fi abilitatea de a citi din fișiere și utilizarea firelor de execuție. Acestea pot fi extrem de utile pentru executarea sarcinilor complexe în fundal, fără a perturba funcționarea programului principal, și sunt esențiale pentru îmbunătățirea performanței aplicației.

Pentru viitoare imbunatatiri s-ar putea adauga imbunătățirea interfeței grafice, gestionarea clientilor in functie de prioritate sau folosirea unor algoritmi mai eficienti pentru a determina coada optima.

7. Bibliografie

1. https://www.geeksforgeeks.org/java-threads/
2. https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html
3. *https://www.javatpoint.com/java-swing*
4. *https://www.tutorialspoint.com/mvc_framework/mvc_framework_introduction.htm*