DOCUMENTAȚIE

TEMA2

Nume Student: Pode Dana Ioana

Grupa: 30222

Cuprins

1. Obiectivul temei

Scopul temei este de a implementa o aplicație de gestionare a clientilor într- o coadă în așa fel încăt timpul de așteptare este minimizat.

Coziile sunt folosite în mod obișnuit pentru a modela domenii din lumea reală. Obiectivul principal al unei cozi este de a oferi un loc pentru un "client" să aștepte înainte de a primi un "serviciu". Managementul sistemelor bazate pe cozi este interesat să minimizeze timpul în care "clienții" lor așteaptă în cozi înainte de a fi serviți. O modalitate de a minimiza timpul de așteptare este de a adăuga mai mulți servere, adică mai multe cozi în sistem (fiecare coadă este considerată ca având un procesor asociat), dar această abordare crește costurile furnizorului de servicii.

Aplicația de management al cozilor simulează (prin definirea unui timp de simulare $t\ simulation$) o serie de N clienți care sosesc pentru a fi serviți, intră în Q cozi, așteaptă, sunt serviți și în cele din urmă părăsesc cozile. Toți clienții sunt generați când simularea este pornită și sunt caracterizați de trei parametri: ID (un număr între 1 și N), $t\ arrival$ (timpul de simulare când sunt pregătiți să intre în coadă) și $t\ service$ (intervalul de timp sau durata necesară pentru a servi clientul; adică timpul de așteptare când clientul este în fața cozii). Aplicația urmărește timpul total petrecut de fiecare client în cozi și calculează timpul mediu de așteptare. Fiecare client este adăugat la coadă cu timpul minim de așteptare când timpul său $t\ arrival$ este mai mare sau egal cu timpul de simulare ($t\ arrival \geq t\ simulation$).

Obiectivele secundare:

i. Înțelegerea cerinței pentru întocmirea claselor si sublaselor necesare, structura acestora
ii. Structura codului într-o arhitectura tip MVC(Model-View-Controller)
iii. Crearea claselor de SimulationManager(reprezinta main-ul pentru simulare, care genereaza un numar de N clienti aleatori si simuleaza timpul de ajungere si servire la cele Q cozi) si Server(clasa care se ocupa cu gestionarea clientului in coada) ca și parte a Model ului *(cap 4)*
iv. Crearea unei interfeței grafice care sa permită interacțiunea cu utilizatorul, ușor de implementat și folosit
v. Interceptarea tuturor erorilor pentru a evita o situație în care programul se află într-o stare necunoscută(ca de exemplu, corectitudinea input-ului introdus de la tastatură prin pop-up-uri)
vi. Utilizarea unui thread separat pentru fiecare coada

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

Cerințele funcționale: Cerințele funcționale ale aplicației sunt de a genera clienți, de a ii aloca cozii cu timpul minim de așteptare, de a ii prioritiza pe baza timpului de sosire și de a calcula timpul mediu de așteptare. Utilizatorul trebuie să poată specifica datele de intrare necesare și să optimizeze alocarea serverelor pentru a minimiza timpul de așteptare și costurile pentru furnizorul de servicii.

La analiza cerinței identificăm datele de intrare / ieșire astfel:

Date de intrare:

- numarul clientilor: N

-numarul de cozi: Q

-intervalul de simulare (tmax simulation)

-timpul de sosire minim si maxim

-timpul de servire minim si maxim

-alegerea politicii de gestionare a cozii

Date de ieșire: - un text log.txt in care apare evolutia cozii si a clientilor(generati aleator) in functie de politica aleasa

Cerințe non-funcționale: aplicație accesibilă la fiecare nouă inserare(consistența inserarilor) , dispunere mesajelor de eroare in cazul unui input greșit, transparență

Scenariul principal de utilizare a aplicatie de gestionare a clientilor unei cozi: utilizatorul introduce de la tastatura datele necesare de intrare prin intermediul interfetei grafice. Acesta apasa butonul de startSimulation si daca datele sunt corecte si valide, incepe simularea. Rezultatul simularii este vizibil in textul ``log.txt``.

Datele introduse sunt parsate in clasa SimulationManager care joacă rolul de controller ca mai apoi, folosindu se clasele Client si Server cu scopul de Model a aplicației pentru realizarea simularii, să se afișeze evolutia dorita in fisierul text. Partea de View este gestionata de clasa SimulationFrame care realizeaza interfata grafica.

Scenariu secundar:

- utilizatorul nu adaugă datele de intrare necesare, caz în care primeste mesajul de eroare ,,Content Missing"

-utilizatorul adaugătoate datele necesare dar acestea nu respectă tipulul de data prestabilită, se afișează mesajul ,,Wrong Input"

3. Proiectare

Arhitectura: Model-View-Controller. . View-ul cuprinde elementele din interfata, acea parte din program cu care utilizatorul intra in contact direct. Controller-ul este cel care face legătura intre

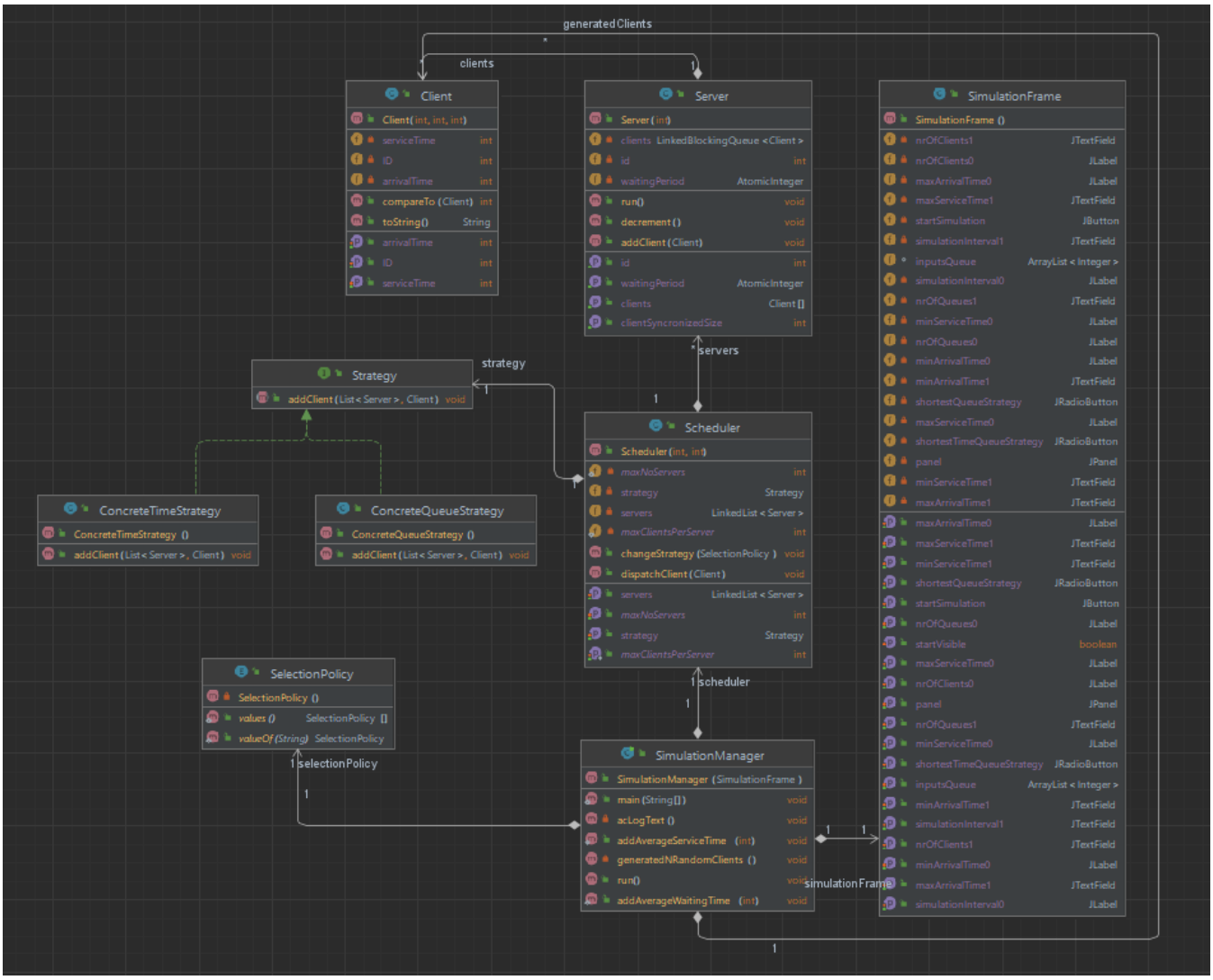
View și Model. Acesta este cel care ia date de intrare, le prelucrează pentru a corespunde modelului de date folosit, apelează operațiile din model și preia rezultatul din model, pe care-l transmite în interfață. Modelul cuprinde toate structurile de date folosite si operatiile care se pot realiza cu acestea.

Pentru a descrie modelul aplicatiei am folosit doua clase:Client (atributele si metodele specifice unui client) si Server (atributele si metodele specifice unei coze).

Pentru a descrie view-ul am folosit o clasa: SimulationFrame

Pentru a descrie controller-ul am folosit sase clase printre care cele mai importante: Simulation Manager (cu rol de main si de incepere a simularii) si Scheduler (Această clasă este responsabilă pentru gestionarea alocării clienților către servere în aplicație)

Diagrama UML este următoarea:



4. Implementare

Clasa Client

Clasa care stochează informații despre clienți. Fiecare client are un ID unic, un timp de sosire și un timp de serviciu. Această clasă implementează interfața "Comparable" pentru a putea compara clienții după timpul de sosire. De asemenea, clasa conține metode pentru a seta și a obține informațiile despre clienți.

Această clasa conține și o metodă de toString() aferentă care afișează un client.

```
private int ID;
private int arrivalTime;
private int serviceTime;

public Client(int ID, int arrivalTime, int serviceTime) {
    this.ID = ID;
    this.arrivalTime=arrivalTime;
    this.serviceTime=serviceTime;
}

public int getID() {
    return ID;
}
```

Clasa Server

Clasa care este responsabilă de gestionarea clienților alocându-i la cozile disponibile implementand interfața Runnable. Serverul are o coadă de clienți (clients) și un AtomicInteger waitingPeriod care reprezintă perioada de așteptare a clienților în coadă. Fiecare server are un identificator unic (id).

Metoda addClient este folosită pentru adăugarea unui client la coada serverului. Metoda verifică dacă clientul poate fi adăugat în coadă și apoi adaugă timpul de servire al clientului la waitingPeriod:

```
public void addClient(Client client)
{
    try{
        clients.add(client);
        waitingPeriod.addAndGet(client.getServiceTime());
    }catch (Exception e){
        JOptionPane.showMessageDialog(null,"Wrong Input");
    }

}
```

Metoda getClientSyncronizedSize() returnează numărul de clienți din coada serverului și este sincronizată pentru a preveni probleme de concurență.

```
public synchronized int getClientSyncronizedSize(){
    return clients.size();
}
```

Metoda run() reprezintă ciclul de viață al serverului. Se verifică dacă există un client disponibil pentru servire și dacă da, se scade timpul de servire al clientului cu o unitate și se scade waitingPeriod cu o unitate. Dacă timpul de servire al clientului devine zero, clientul este eliminat din coadă

```
@Override
public void run()
{

    while(true){
        Client nextClient=clients.peek();

        if(nextClient!=null){
            nextClient.setServiceTime(nextClient.getServiceTime()-1);
            decrement();
            if(nextClient.getServiceTime()==0){
                try {
                    clients.take();
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
            }
        }
        try{
            Thread.sleep(1000);
        }catch(InterruptedException e){
            JOptionPane.showMessageDialog(null,"Wrong Input");
        }
    }
}
```

Clasa SimulationManager

Această clasă este managerul de simulare și are rolul de a coordona simularea unui sistemului de cozi.

Metoda generatedNRandomClients() generează clienți aleatoriu pe baza numărului de clienți nrOfClients, timpului de sosire minim și maxim minArrivalTime și maxArrivalTime, și timpul de servire minim și maxim minServiceTime și maxServiceTime. Acești clienți sunt sortați în ordinea timpului de sosire, iar lista lor este stocată în variabila generatedClients:

```
private void generatedNRandomClients() {
    generatedClients = Collections.synchronizedList(new ArrayList<>());
    Random random = new Random();
    for (int i = 0; i < nrOfClients; i++) {
```

```
        Client client = new Client(i, random.nextInt(minArrivalTime, maxArrivalTime),
random.nextInt(minServiceTime, maxServiceTime));
        generatedClients.add(client);
    }
    generatedClients.sort(new Comparator<Client>() {
        @Override
        public int compare(Client o1, Client o2) {
            return o1.getArrivalTime() - o2.getArrivalTime();
        }
    });

}
```

Metoda acLogText() adaugă text la variabila logText, care conține jurnalul evenimentelor din simulare:

```
private void acLogText() {
    logText += "Time: " + currentTime + "\n";
    logText += "Waiting Clients: " + "";
    for (Client client : generatedClients) {
        logText += "(" + client.getID() + ", " + client.getArrivalTime() + ", " +
client.getServiceTime() + ");";
    }
    logText += "\n";
    for (Server server : scheduler.getServers()) {
        logText += "Queue " + server.getId() + ": ";
        if (server.getClientSyncronizedSize() == 0) {
            logText += "closed";
        } else {
            for (Client client : server.getClients()) {
                logText += "(" + client.getID() + ", " + client.getArrivalTime() + ", " +
client.getServiceTime() + ");";
            }
        }
        logText += "\n";
    }
    logText += "\n";
    System.out.println(currentTime);
}
```

Metoda run() rulează simularea, bucla simulează timpul în intervalul 0 la simulationInterval. Aceasta face verificări periodice ale listei generatedClients pentru a vedea dacă un client a ajuns, îi atribuie o coadă și îl elimină din lista de clienți generată. Altfel, așteaptă un timp scurt pentru a simula trecerea timpului:

```
@Override
public void run() {
    while (currentTime < simulationInterval) {
        while (true) {
            if (generatedClients.size() > 0) {
                Client client = generatedClients.get(0);
```

```
                if (client.getArrivalTime() == currentTime) {
                    scheduler.dispatchClient(client);//pune clientul in coada care
trebuie
                    addAverageWaitingTime(client.getArrivalTime());
                    addAverageWaitingTime(client.getServiceTime());
                    generatedClients.remove(client);
                } else {
                    if (client.getArrivalTime() > currentTime) {//clienti serviti
                        break;
                    }
                }
            }
            if (generatedClients.size() == 0) {
                break;
            }
        }
        acLogText();
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        try {
            FileWriter writer = new FileWriter("log.txt");
            logText+="\naverage wating time:"+sum1/nrOfClients;
            logText+="\naverage service time:"+sum2/nrOfClients;

            writer.write(logText);

            writer.close();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
        currentTime++;
    }
}
```

Metoda main() pornește simularea atunci când butonul de pornire este apăsat. Acesta preia valorile de la componentele interfeței grafice de utilizator pentru a inițializa variabilele statice ale clasei SimulationManager.

Clasa Scheduler

Acesta are două proprietăți statice pentru numărul maxim de cozișinumărul maxim de clienți per cozi. Clasa este utilizată într-un sistem de simulare a unei cozi de așteptare pentru servicii.

Constructorul clasei inițializează o listă de servere și o buclă for adaugă în mod repetat noi obiecte Server la această listă și inițiază un fir de execuție separat pentru fiecare server. În plus, constructorul stabilește strategia implicită a obiectului de tip Scheduler ca "SHORTEST_TIME":

```
public Scheduler(int maxNoServers, int maxClientsPerService) {
    servers = new LinkedList<>();
    this.maxNoServers=maxNoServers;
    this.maxClientsPerServer = maxClientsPerServer;
    for (int size = 0; size < maxNoServers; size++) {
        Server server=new Server(size+1);
        servers.add( server);
        Thread thread = new Thread(server);
        thread.start();
    }

    changeStrategy(SelectionPolicy.SHORTEST_TIME);
}
```

Metoda changeStrategy() este utilizată pentru a schimba strategia de selecție a coziiatunci când este necesara:

```
public void changeStrategy(SelectionPolicy policy)
{
    if(policy==SelectionPolicy.SHORTEST_QUEUE){
        strategy= (Strategy) new ConcreteQueueStrategy();
    }
    else {
        strategy= (Strategy) new ConcreteTimeStrategy();
    }
}
```

Metoda dispatchClient() este folosită pentru a trimite un client către un server în funcție de strategia curentă:

```
public void changeStrategy(SelectionPolicy policy)
{
    if(policy==SelectionPolicy.SHORTEST_QUEUE){
        strategy= (Strategy) new ConcreteQueueStrategy();
    }
    else {
        strategy= (Strategy) new ConcreteTimeStrategy();
    }
}
```

Clasa ConcreteQueueStrategy

Clasa ConcreteQueueStrategy implementează interfața Strategy și oferă o strategie de selecție a serverului bazată pe dimensiunea minimă a cozii de așteptare. Aceasta alege cea mai mică dimensiune de coadă și adaugă clientul la aceasta.

```
@Override
public void addClient(List<Server> servers, Client client) {
    if(Scheduler.getMaxClientsPerServer()>0){
        int min=servers.get(0).getClientSyncronizedSize();
```

```
        Server nextServer=servers.get(0);
        int i=1;
        while(i!=Scheduler.getMaxClientsPerServer()){
            if(servers.get(i).getClientSyncronizedSize()<min){
                min=servers.get(i).getClientSyncronizedSize();
                nextServer=servers.get(i);
            }
            i++;
        }
        nextServer.addClient(client);
    }
}
```

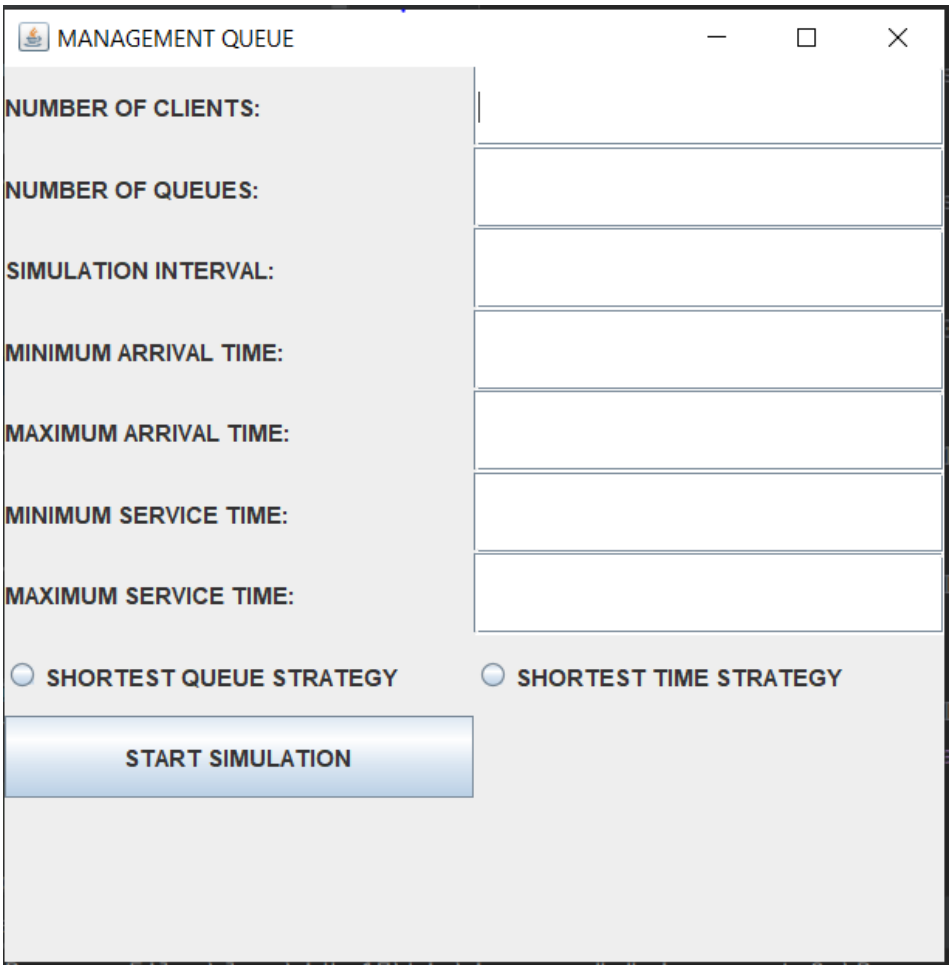
Clasa ConcreteTimeStrategy

Clasa ConcreteTimeStrategy alege coada cu cel mai putin timp de asteptare și adaugă clientul la aceasta.

```
@Override
public void addClient(List<Server> servers, Client client) {
    //the smallest waiting period for a server gets a client
    Server nextServer=servers.get(0);
    for(Server server:servers){
        if(server.getWaitingPeriod().get()<nextServer.getWaitingPeriod().get()){
            nextServer=server;
        }
    }
    nextServer.addClient(client);
}
```

Clasa SimulationFrame

Acesta utilizează biblioteca Swing pentru a crea interfața grafică cu utilizatorul. De asemenea, definește o serie de metode get și set pentru a permite accesul la valorile introduse în cadrul de simulare.



5. Rezultate

Pentru a testa aplicatia am folosit testele stabilite pentru acest assignment. Am realizat 3 teste, test1.txt, test2.txt si test3.txt.

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 $[t_{arrival}^{MIN}, t_{arrival}^{MAX}] = [10, 100]$ $[t_{service}^{MIN}, t_{service}^{MAX}] = [3, 9]$

6. Concluzii

Prin această temă consider că am reusit sa aprofundez cunostinte de POO învățate și am aprofundat lucrul cu thread uri, depanarea codului.

Posibile dezvoltări viitoare:

-interfata grafica mai complexa

-interfata grafica pentru evolutia simularii care sa termita vizualizarea in timp a fiecarului client in asteptare sau selectat sa fie servit

-istoric de simulari

-precizarea timpului mediu de servire si asteptare a clientilor

-integrarea unui meniu de exit, help sau readme

7. Bibliografie

https://www.w3schools.com/java/java_threads.asp

https://www.digitalocean.com/community/tutorials/atomicinteger-java

https://www.geeksforgeeks.org/java-threads/

https://dsrl.eu/courses/pt/