Documentatie

Tema numarul 2

Costea Delia

Grupa: 30222

CUPRINS

1.Obiectivul temei

Obiectivul temei numarul doi este de a implementa si proiecta un sistem de gestionare al cozilor, generand N clienti care asteapta la randul celor Q cozi, intr-un mod cat mai eficient, folosind thread-uri si cozi in Java.

Obiective secundare:

- Analiza problemei si identificarea cerintelor temei
- Proiectarea simularii cozilor
- Implementarea simularii
- Testarea simularii

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

Pentru a implementa simulatorul de cozi, este necesar in primul rand un generator de clienti si sortarea acestora in ordine crescatoare in functie de timpul de sosire. Apoi cu ajutorul unui scheduler se impart clientii in cozi, in functie de timp sau de cea mai scurta coada.

Aplicatia interactioneaza cu utilizatorul printr-o interfata grafica usor de utilizat

care ofera o metoda de introducere a datelor si apasarea unui buton pentru pornirea simularii. Datele rezultate in urma simularii se vor salva intr un fisier, pe care utilizatorul il poate deschide pentru a observa rezultatul.

Pasii de utilizare ai aplicatiei sunt:

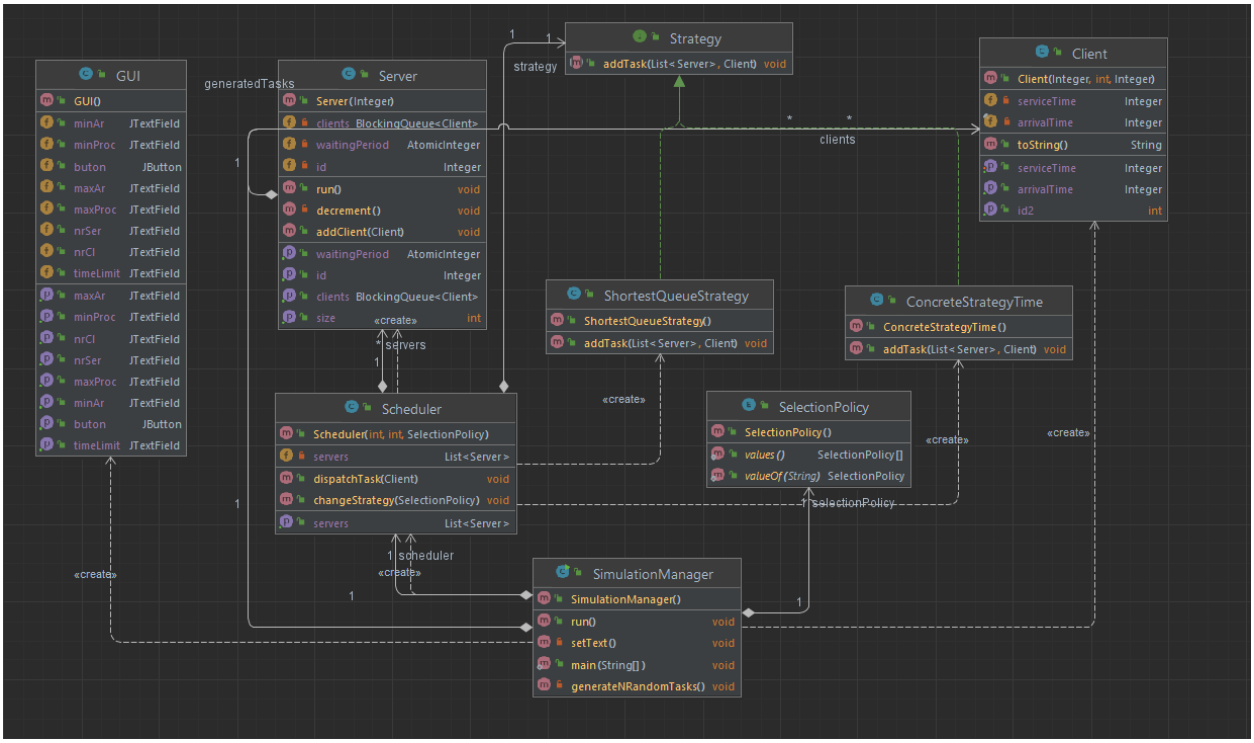
- Introducerea datelor pentru a simula
- Apasarea pe buton pentru pornirea simularii
- Deschiderea fisierului pentru a vedea rezultatul

3. Proiectare

Aplicatia este impartita in trei pachete:

- Model, care contine clasele: Client si Server
- BusinessLogic, care contine clasele: Strategy, ConcreteStrategyTime, Scheduler, SelectionPolicy, ShortestQueueStrategy si Simulation Manager
- GUI: care contine clasa interfetei grafice, GUI

Diagrama UML :



4. Implementare

Clasa Client:

Aceasta clasa reprezina un obiect de tip client pentru un sistem de simulare a unei cozi de asteptare, unde clientii asteapta sa fie serviti. Ea are cateva atribute, cum ar fi timpul de sosire, timpul de servire și un identificator unic pentru fiecare client.

Clasa extinde clasa "Thread", sugerand faptul ca aceasta poate fi utilizată pentru a crea obiecte client care pot fi executate în paralel cu alte obiecte din program.

Cod:

```
package Model;

public class Client extends Thread {
    private final Integer arrivalTime;
    private int id;
    private Integer serviceTime;
    public Client(Integer a, int id, Integer s)
    {
        this.arrivalTime=a;
        this.id=id;
        this.serviceTime=s;

    }

    public void setServiceTime(Integer serviceTime) {
        this.serviceTime = serviceTime;
    }

    public Integer getArrivalTime() {
        return arrivalTime;
    }

    public int getId2() {
        return id;
    }

    public Integer getServiceTime() {
        return serviceTime;
    }
    public String toString()
    {
        String s="";
        s="("+id+","+arrivalTime+","+serviceTime+" )";
        return s;

    }
}
```

Clasa Server: Aceasta clasa reprezinta un obiect de tip server pentru un sistem de simulare al unei cozi de asteptare. Este responsabila de preluarea clientilor din coada si procesarea lor. Clasa implementeaza interfata Runnable, pentru a putea crea obiecte server care pot fi executate in paralel cu alte obiecte din program. Metoda run este principala metoda a clasei, aceasta ruleaza intr-o bucla infinita si incearca sa preia primul client din coada si sa-l proceseze timp de o secunda. Apoi verifica daca timpul de servire a fost redus la 0, caz in care acest client va fi scos din coada.

Cod:

```
public void run()
{
    while(true)
    {

        try{
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        try
        {
            if(clients.size()>0) {

clients.element().setServiceTime(clients.element().getServiceTime() - 1);
                decrement();
                if(clients.element().getServiceTime()==0)
                    clients.take();
            }
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Clasa Strategy: Este o interfata, care defineste un comportament comun pentru strategiile de alocare a clientilor catre cozi. Este implementata pentru a crea diferite strategii de alocare a clientilor, urmand sa fie prezentate.

Cod:

```
package BusinessLogic;

import Model.Client;
import Model.Server;

import java.util.List;

public interface Strategy {
    public void addTask(List<Server> servers, Client t);

}
```

Clasa ShortestQueueStrategy: Aceasta clasa implementeaza interfata Strategy si suprascrie metoda “addTask”, care primeste o lista de servere si un client, dupa care sorteaza serverele in ordine crescatoare a dimensiunii cozii, adica in ordine crescatoare a numarului de client care asteapta sa fie serviti. Dupa sortare, se adauga clientul in coada primului server din lista, care ar fi serverul cu cea mai scurta coada.

Cod:

```
package BusinessLogic;

import Model.Client;
import Model.Server;

import java.util.List;

public class ShortestQueueStrategy implements Strategy {
    @Override
    public void addTask(List<Server> servers, Client t) {//verific care
server are cei mai putini clienti si la ala adaug clientul noi
        servers.sort((t1, t2) -> Integer.compare(t1.getSize(),
t2.getSize()));
        servers.get(0).addClient(t);
    }
}
```

Clasa ConcreteStrategyTime: Aceasta clasa, la fel ca si cea anterioara, implementeaza interfata Strategy si suprascrie metoda “addTask”. Scopul ei este de a gasii serverul cu timpul total minim de asteptare si de a adauga clientul in coada acestuia.

Cod:

```
package BusinessLogic;

import Model.Client;
import Model.Server;

import java.util.List;

public class ConcreteStrategyTime implements Strategy {

    @Override
    public void addTask(List<Server> servers, Client t)
    {
        int idServer = servers.get(0).getId();
        int totalTime = servers.get(0).getWaitingPeriod().get();
        for (Server a : servers) {
            if (totalTime > a.getWaitingPeriod().get()) {
                totalTime = a.getWaitingPeriod().get();
                idServer = a.getId();
            }
        }
        servers.get(idServer - 1).addClient(t);



    }
}
```

Clasa Scheduler: Aceasta clasa reprezinta un program care gestioneaza procesul de impartire a sarcinilor catre unul sau mai multe servere.

Cod:

```
public void changeStrategy(SelectionPolicy policy)
{

    if(policy == SelectionPolicy.SHORTEST_QUEUE)
    {
        strategy=new ShortestQueueStrategy();
    }
    if(policy==SelectionPolicy.SHORTEST_TIME)
    {
        strategy=new ConcreteStrategyTime();
    }
}
public void dispatchTask(Client t)
{
    //call the strategy addTask method
    strategy.addTask(servers,t);
}
public List<Server>getServers()
{
    return servers;
}
```

Clasa SelectionPolicy: este o enumerare care defineste doua politici posibile de selectie a unui server dintr-un sistem de cozi: coada cu cel mai mic numar de clienti deja in asteptare, sau coada cu cel mai mic timp de asteptare.

Cod:

```
package BusinessLogic;

public enum SelectionPolicy {
    SHORTEST_QUEUE, SHORTEST_TIME
}
```

Clasa SimulationManager: Reprezinta simularea unui sistem de cozi, implementeaza interfata “Runnable”, permitand instantierea si executarea clasei intr-un fir de executie separate. Aici se afla unele din metodele principale ale programului:

Metoda generateNRandomTasks: genereaza un numar de client cu timp se sosire si timp de procesare aleatorii si ii sorteaza in functie de timpul de sosire. Ei sunt stocati intr-o lista “generatedTasks”.

Metoda run: contine bucla principala a simularii. La fiecare iteratie, verific clientii din lista si adauga clientii care au ajuns in timpul current la coada prin apelarea metodei

“dispatckTasks” din Scheduler, apoi elimina clientii din lista. Timpul curent este incrementat cu 1 si se asteapta o secunda. La sfarsitul simularii, variabila text este scrisa intr un fisier.

Cod:

```
private void generateNRandomTasks()
{
    Random r=new Random();
    int processingTime;
    int arrivalTime;

    for(int i=0;i<numberOfClients;i++)
    {
        processingTime=r.nextInt(maxProcessingTime-
minProcessingTime)+minProcessingTime;
        arrivalTime=r.nextInt(maxArrivalTime-minArrivalTime)+minArrivalTime;
        Client c=new Client(arrivalTime,i+1,processingTime);
        generatedTasks.add(c);
    }
    generatedTasks.sort((t1, t2) -> Integer.compare(t1.getArrivalTime(),
t2.getArrivalTime()));

}
public void run()
{

    while(currentTime<timeLimit)
    {

        boolean k=true;
        while(k)
        {
            if(generatedTasks.size()>0)
            {
                Client c=generatedTasks.get(0);
                if(c.getArrivalTime()==currentTime)
                {
                    scheduler.dispatchTask(c);
                    generatedTasks.remove(c);
                }
                else
                {
                    if(c.getArrivalTime()>currentTime)
                        k=false;
                }
            }
            if(generatedTasks.size()==0)
                k=false;

        }

        setText();
        currentTime++;
        try
        {
            Thread.sleep(1000);
```

```
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        try
        {
            FileWriter fis =new FileWriter("Fisier.txt");
            fis.write(text);
            fis.close();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }

    }
```

Clasa GUI: Aceasta clasa creeaza o interfata grafica pentru a permite utilizatorului sa introduca valorile necesare pentru a rula un rogram de simulare. Interfata grafica contine o fereasta, un buton se start si cateva campuri si etichete pentru a specifica datele de intrare.

Cod:

```
public GUI()
{
    timeLimit1=new JLabel("Timp limita:");
    minAr1=new JLabel("Timp min de sosire:");
    maxAr1=new JLabel("Timp max de sosire:");
    minProc1=new JLabel("Timp min de service:");
    maxProc1=new JLabel("Timp max de service:");
    nrSer1=new JLabel("Numar de cozi:");
    nrCl1=new JLabel("Numar de clienti:");
    buton=new JButton("Start");

    timeLimit=new JTextField("60");
    minAr=new JTextField("2");
    maxAr=new JTextField("30");
    minProc=new JTextField("2");
    maxProc=new JTextField("4");
    nrSer=new JTextField("2");
    nrCl=new JTextField("4");
    f=new JFrame("Cozi");
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    f.setBounds(200,200,300,400);
    f.setResizable(false);

    timeLimit1.setBounds(20,30,130,20);
    nrSer1.setBounds(20,60,130,20);
    nrCl1.setBounds(20,90,130,20);
    minAr1.setBounds(20,120,130,20);
    maxAr1.setBounds(20,150,130,20);
    minProc1.setBounds(20,180,130,20);
    maxProc1.setBounds(20,210,130,20);

    timeLimit.setBounds(150,30,100,20);
    nrSer.setBounds(150,60,100,20);
```

```
    nrCl.setBounds(150,90,100,20);
    minAr.setBounds(150,120,100,20);
    maxAr.setBounds(150,150,100,20);
    minProc.setBounds(150,180,100,20);
    maxProc.setBounds(150,210,100,20);
    buton.setBounds(20,240,100,50);
```

5. Rezultate

Rezultatele simularii pentru urmatoarele date de intrare se vor gasii in fisierele Test1.txt, Test2.txt, 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

In concluzie, aceasta tema m a ajutat sa inteleg mai bine limbajul java, si sa imi imbunatatesc cunostintele. Simulatorul functioneaza correct, dar ar mai putea exista posibile imbunatatiri, cum ar fi realizarea unei interfete grafice care sa arate in timp real dispunerea clientilor la cozi, dar si afisarea timpului mediu pentru servire.

7. Bibliografie

- https://dsrl.eu/courses/pt/
- https://www.w3schools.com/
- https://www.geeksforgeeks.org/