DOCUMENTATIE TEMA 2

QUEUES MANAGEMENT APPLICATION USING THREADS AND SYNCHRONIZATION MECHANISMS

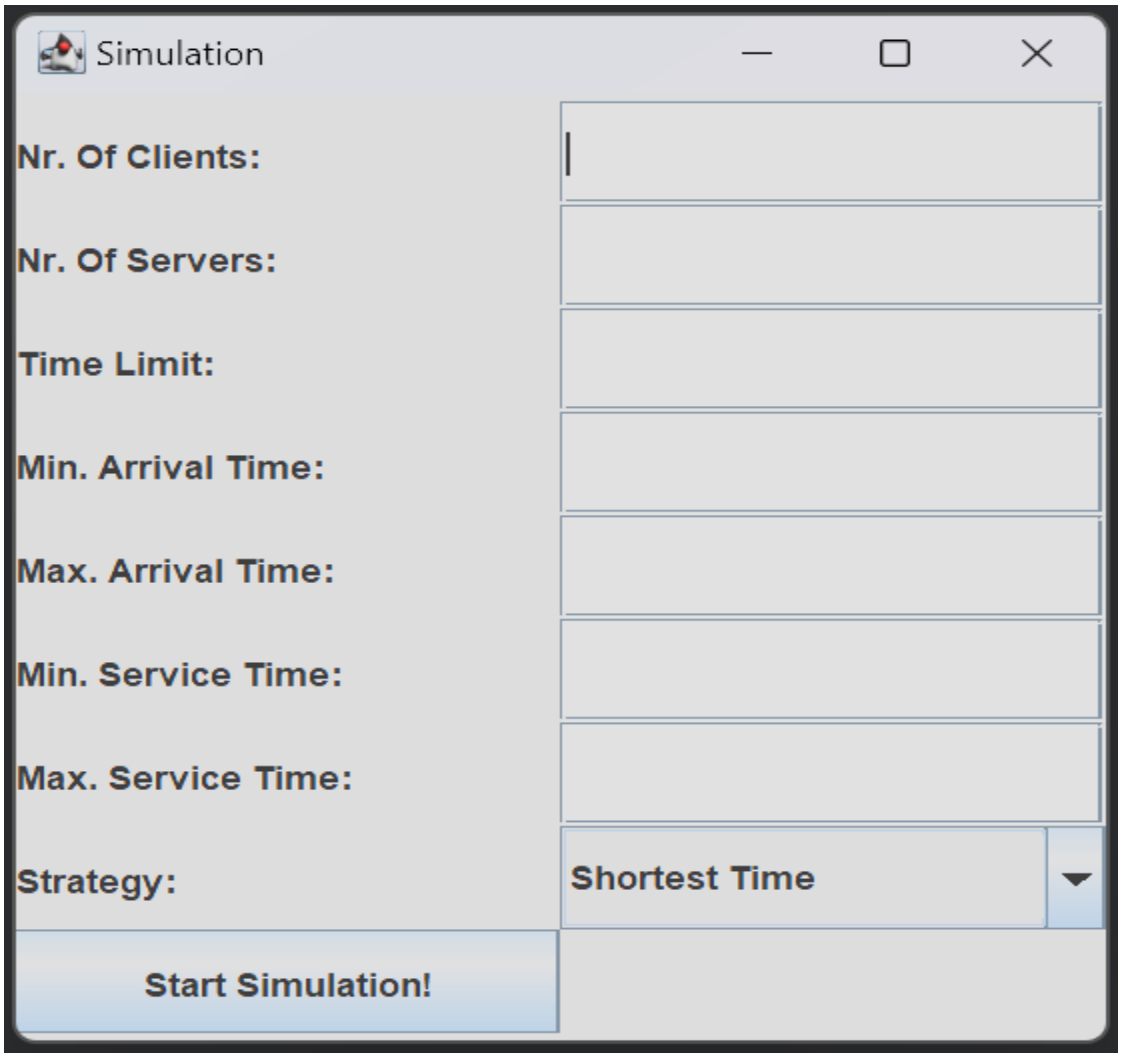
Sarkozi Stefan

Grupa: 30223

Cuprins

Cerinte Functionale

Proiectati si implementati o aplicatie de gestionare a cozilor care atribuie clientii la cozi, astfel incat timpul de asteptare este minimizat.



Obiective

Obiectiv Principal:

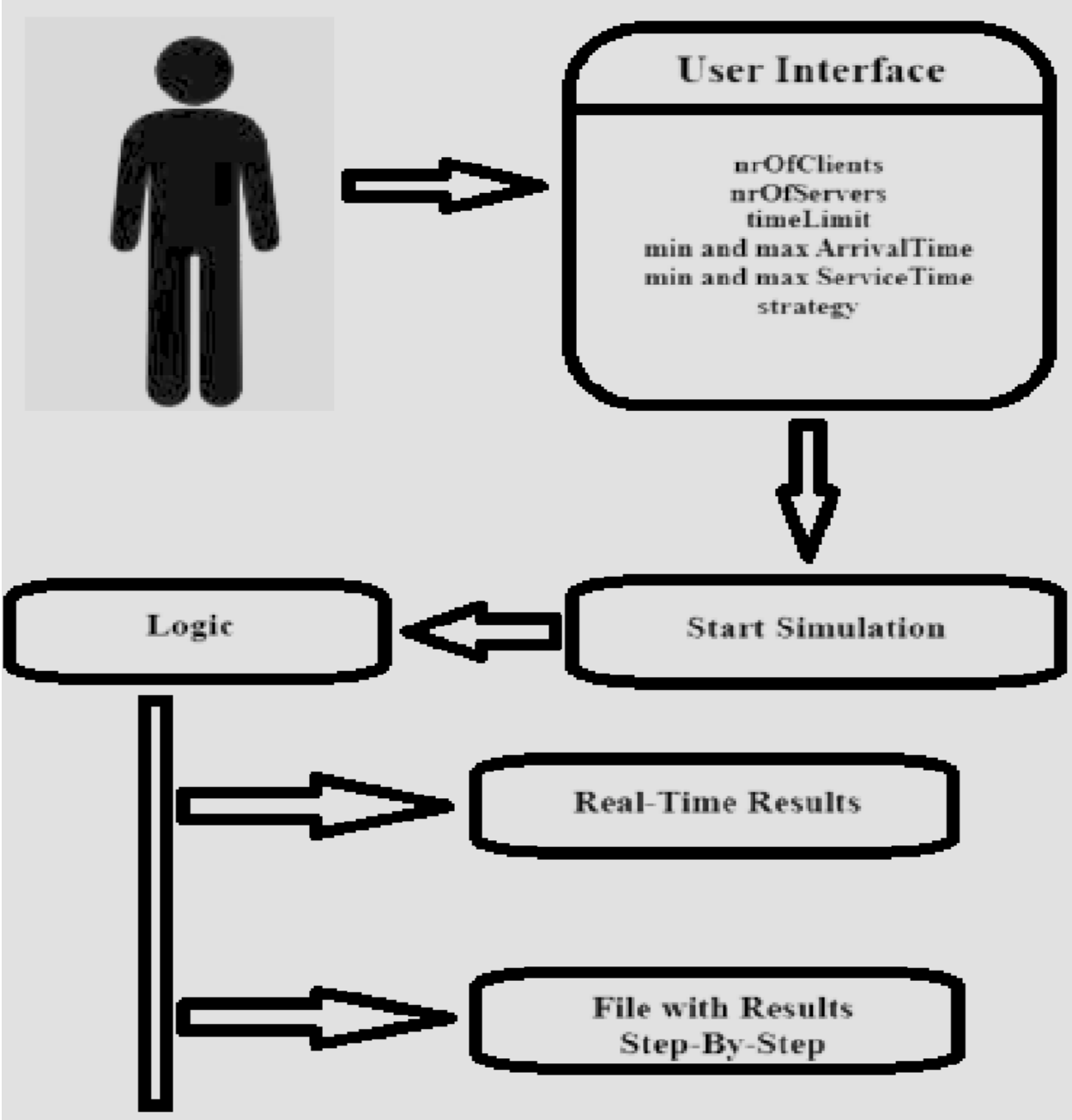
Cozile sunt utilizate in mod obisnuit pentru a modela domeniile lumii reale. Obiectivul principal al unei cozi este de a furniza un loc pentru un client, care asteapta sa fie servit. Gestionarea sistemelor bazate pe cozi, sunt interesate in minimizarea timpului pe care clientii il petrec in cozi inainte de a fi serviti.

Obiective Secundare:

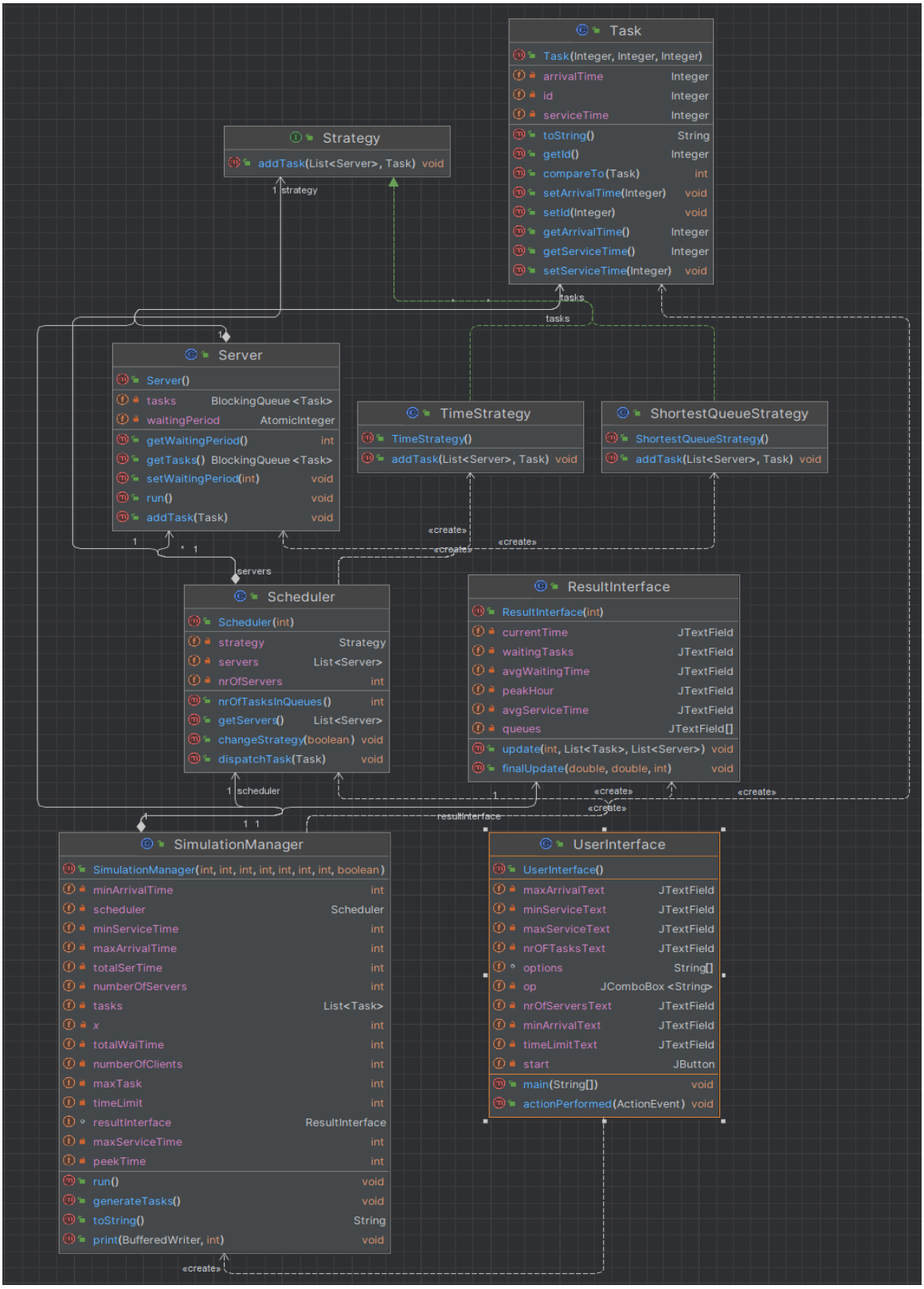
Aplicatia simuleaza sosirea a N clienti, care intra in Q cozi, asteptand sa fie serviti, ulterior iesind din cozi. Se monitorizeaza timpul total petrecut de fiecare client in coada, si se calculeaza timpul mediu de asteptare. Fiecare client este adaugat in coada cu cel mai mic timp de asteptare.

Analiza Problemei

Use Case-uri:



Proiectare


Task
Task(Integer, Integer, Integer)
arrivalTime Integer
id Integer
serviceTime Integer
toString() String
getId() Integer
compareTo(Task) int
setArrivalTime(Integer) void
setId(Integer) void
getArrivalTime() Integer
getServiceTime() Integer
setServiceTime(Integer) void
Strategy
addTask(List<Server>, Task) void
1 strategy
tasks
tasks
Server
Server()
tasks BlockingQueue <Task>
waitingPeriod AtomicInteger
getWaitingPeriod() int
getTasks() BlockingQueue <Task>
setWaitingPeriod(int) void
run() void
addTask(Task) void
TimeStrategy
TimeStrategy()
addTask(List<Server>, Task) void
ShortestQueueStrategy
ShortestQueueStrategy()
addTask(List<Server>, Task) void
«create»
«create»
«create»
servers
Scheduler
Scheduler(int)
strategy Strategy
servers List<Server>
nrOfServers int
nrOfTasksInQueues() int
getServers() List<Server>
changeStrategy(boolean) void
dispatchTask(Task) void
ResultInterface
ResultInterface(int)
currentTime JTextField
waitingTasks JTextField
avgWaitingTime JTextField
peakHour JTextField
avgServiceTime JTextField
queues JTextField[]
update(int, List<Task>, List<Server>) void
finalUpdate(double, double, int) void
1 scheduler
«create»
«create»
«create»
resultInterface
SimulationManager
SimulationManager(int, int, int, int, int, int, int, boolean)
minArrivalTime int
scheduler Scheduler
minServiceTime int
maxArrivalTime int
totalSerTime int
numberOfServers int
tasks List<Task>
x int
totalWaiTime int
numberOfClients int
maxTask int
timeLimit int
resultInterface ResultInterface
maxServiceTime int
peekTime int
run() void
generateTasks() void
toString() String
print(BufferedWriter, int) void
UserInterface
UserInterface()
maxArrivalText JTextField
minServiceText JTextField
maxServiceText JTextField
nrOFTasksText JTextField
options String[]
op JComboBox <String>
nrOfServersText JTextField
minArrivalText JTextField
timeLimitText JTextField
start JButton
main(String[]) void
actionPerformed(ActionEvent) void
«create»

Structuri de Date:

```
public class Server implements Runnable {
    7 usages
    private BlockingQueue<Task> tasks;
    5 usages
    private AtomicInteger waitingPeriod;

    1 usage   Your Name
    public Server() {
        tasks = new LinkedBlockingQueue<>();
        waitingPeriod = new AtomicInteger( initialValue: 0);
    }
```

Clasa Server are la baza un BlockingQueue, scopul acestuia fiind de a stoca clienti, care sunt formati din tuple, cum ar fi (id, arrivalTime, serviceTime). Variabila waitingPeriod este de tipul AtomicInteger, valoarea acestuia este actualizata de catre un fir de executie direct in memoria principala, ceea ce inseamna ca valoarea actualizata este vizibila de catre celelalte fire de executie.

Implementare

```
public void run() {
    while (true) {
        try {
            Task client = tasks.peek();
            if (client != null) {
                Thread.sleep( millis: 1000);
                tasks.peek().setServiceTime(client.getServiceTime() - 1);
                synchronized (waitingPeriod) {
                    this.setWaitingPeriod(this.getWaitingPeriod() - 1);
                }
                if (tasks.peek().getServiceTime() <= 0) {
                    tasks.poll();
                }
            } else {
                Thread.sleep( millis: 1000);
            }
        } catch (InterruptedException exception) {
            exception.printStackTrace();
        }
    }
}
```

Metoda run din cadrul clasei Server are rolul de a lua urmatorul client care asteapta la coada, pune firul de executie pe pauza, respectiv decrementeaza variabila waitingPeriod.

```
public void addTask(Task newTask) {
    synchronized (waitingPeriod) {
        try {
            tasks.put(newTask);
            this.setWaitingPeriod(this.getWaitingPeriod() + newTask.getServiceTime());
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
    }
}
```

Operatiile aplicate asupra cozii tasks sunt sincronizate in functie de waitingPeriod. Metoda addTask care adauga clienti in coada, poate sa fie accesata la un moment dat doar de un singur fir de executie, cel care detine lock-ul, restul firelor de executie care doresc sa acceseze metoda sunt nevoite sa astepte pana cand lock-ul este eliberat.

```
public interface Strategy {
    1 usage  2 implementations  Your Name
    void addTask(List<Server> servers, Task task);

}
```

Interfata Strategy este un contract, pe care clasele care il implementeaza trebuie sa il respecte. Cele doua clase care implementeaza aceasta interfata sunt: ShortestQueueStrategy si TimeStrategy.

```java
public class ShortestQueueStrategy implements Strategy {
    1 usage  Your Name
    @Override
    public void addTask(List<Server> servers, Task task) {
        int minLen = servers.get(0).getTasks().size();
        int indexOfMin = 0;
        for (Server index : servers) {
            if (index.getTasks().size() < minLen) {
                minLen = index.getTasks().size();
                indexOfMin = servers.indexOf(index);
            }
        }
        servers.get(indexOfMin).addTask(task);
    }

}
```

Metoda addTask din cadrul acestei clase are rolul de a adauga un client la cea mai scurta coada.

```java
public class TimeStrategy implements Strategy {
    1 usage  Your Name
    @Override
    public void addTask(List<Server> servers, Task task) {
        Server minTimeServ = servers.get(0);
        for (Server index : servers) {
            if (index.getWaitingPeriod() < minTimeServ.getWaitingPeriod()) {
                minTimeServ = index;
            }
        }
        servers.get(servers.indexOf(minTimeServ)).addTask(task);
    }

}
```

Spre deosebire de metoda addTask din clasa ShortestQueueStrategy, metoda din clasa TimeStrategy are rolul de a adauga un client la coada cu cel mai mic waitingTime.

Clasa Scheduler este responsabila de setarea strategiei selectate de catre utilizator in Interfata Grafica (UserInterface), respectiv in functie de alegerea facuta metoda dispatchTask adauga clientii in cozi.

Concluzii si Dezvoltari Ulterioare

In urma realizarii acestui proiect m-am familiarizat cu conceptul de Thread, cu blocurile sincronizate, cu variabilele volatile, care ajuta la scrierea unui cod Thread-Safe. Aplicatia rezultata fiind folositoare in cadrul vederii minimizarii timpilor de asteptare la cozi.

Din punctul meu de vedere, o dezvoltare ulterioara ar putea fi calcularea numarului optim de cozi pentru o multime de clienti, in cazul in care o coada are un pret de fabricatie si un cost cat timp aceasta este deschisa. Numarul optim fiind calculat in functie de waitingTime si de costul total.

Bibliografie

https://www.tutorialspoint.com/java/java_multithreading.htm

https://www.tutorialspoint.com/java/java_thread_synchronization.htm

https://www.geeksforgeeks.org/runnable-interface-in-java/?ref=lbp

https://docs.oracle.com/javase/tutorial/essential/concurrency/interrupt.html