DOCUMENTATIE

TEMA *2*

NUME STUDENT: GHERMAN COSMIN-NICOLAE
GRUPA: 30228

CUPRINS

1. Obiectivul temei

OBIECTIVUL PRINCIPAL este de a reazila proiectarea si implementarea unui manager de cozi (queue manager) pentru a analiza aplicatia prin :

- Generarea unui numar de N clienti (task-uri) care au ca scop sa fie serviti (rezolvate), intra in cozile Q si parcurg urmatorii pasi : isi asteapta randul, sunt serviti, parasesc coada la care au asteptat si au fost serviti.
- Calcularea timpilor de ora de varf, timpilor de asteptare cat si de serviciu

OBIECTIVELE SECUNTARE pot fi acestea:

- Analizarea problemei pentru care se va genera solutia si identificarea cerintelor ei (2)
- Proiectarea aplicatiei de simulare (3)
- Implementarea acesteia (4)
- Rezultatele tesatrii (5)

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

Se pot identifica doua cazuri de cerinte :

a) Cerinte functionale :

- Aplicatia ce va urma sa fie implementata trebuie sa ii permita utilizatorului urmatoarele lucruri : sa poata sa seteze/actualizeze datele simularii, permiterea pornirii simularii si dupa aceasta posibilitatea de vizualizare in timp real a evolutiei cozilor si in final rezultatele simularii plus realizarea unui fisier in care se vor trece rezultatele simularii

b) Cerintele non-functionale :

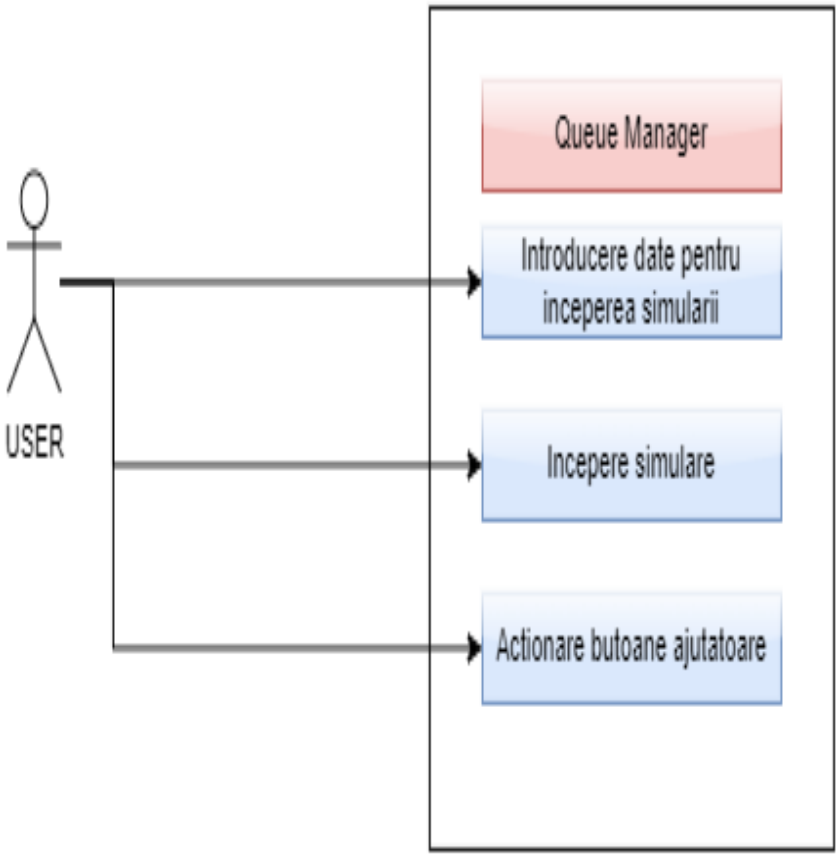
- Aplicatia trebuie sa fie cat mai usor de utilizat pentru user si sa vina cu anumite functionalitati in plus pentru a reduce efortul utilizatorului.

Use-case :

- Utilizatorul initializeaza textField-urile corespunzatoare numarului de clienti si cozi, timpului minim si maxim de sosire, timpului minim si maxim de servire si limita de timp in care trebuie sa se produca tot acest proces.
- Nu se va intampla nimic pana cand utilizatorul nu apasa butonul de Launch
- Se vor valida datele introduce in textField uri
- Pentru simplitate se poate apasa peu nul din cele trei butoane de Fill care introduc in textField-uri data prestabilite pentru anumite exemple, astfel pentru alte tipuri de cazuri trebuie introduse date de utilizator

- La actiunea butonului de Delete se vor sterge toate field-urile care au fost completate de utilizator sau simulare, pentru o posibila reincepere a unei noi simulari.

Diagrama use-case :



3. Proiectare

DIAGRAMA DE PACHETE:

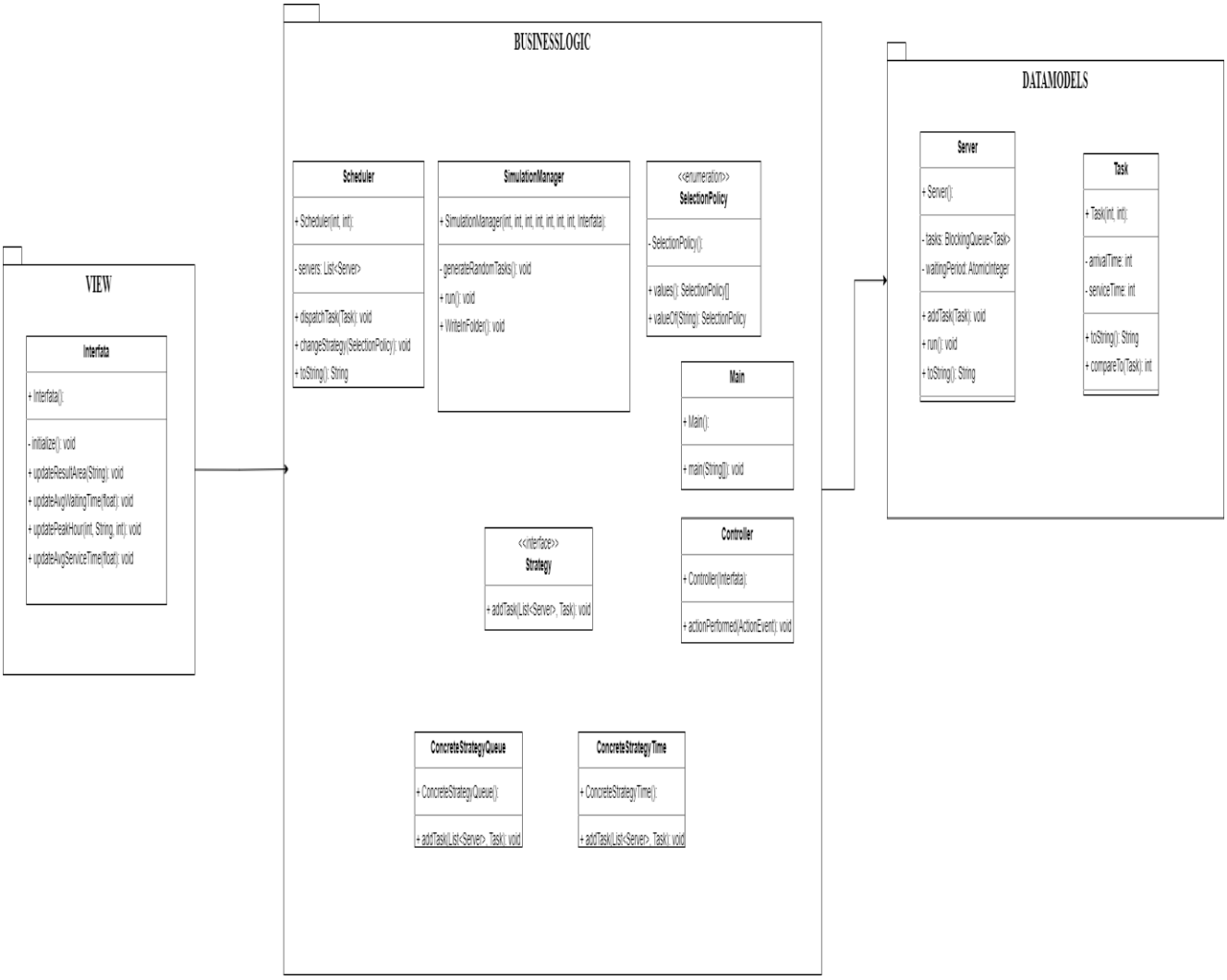
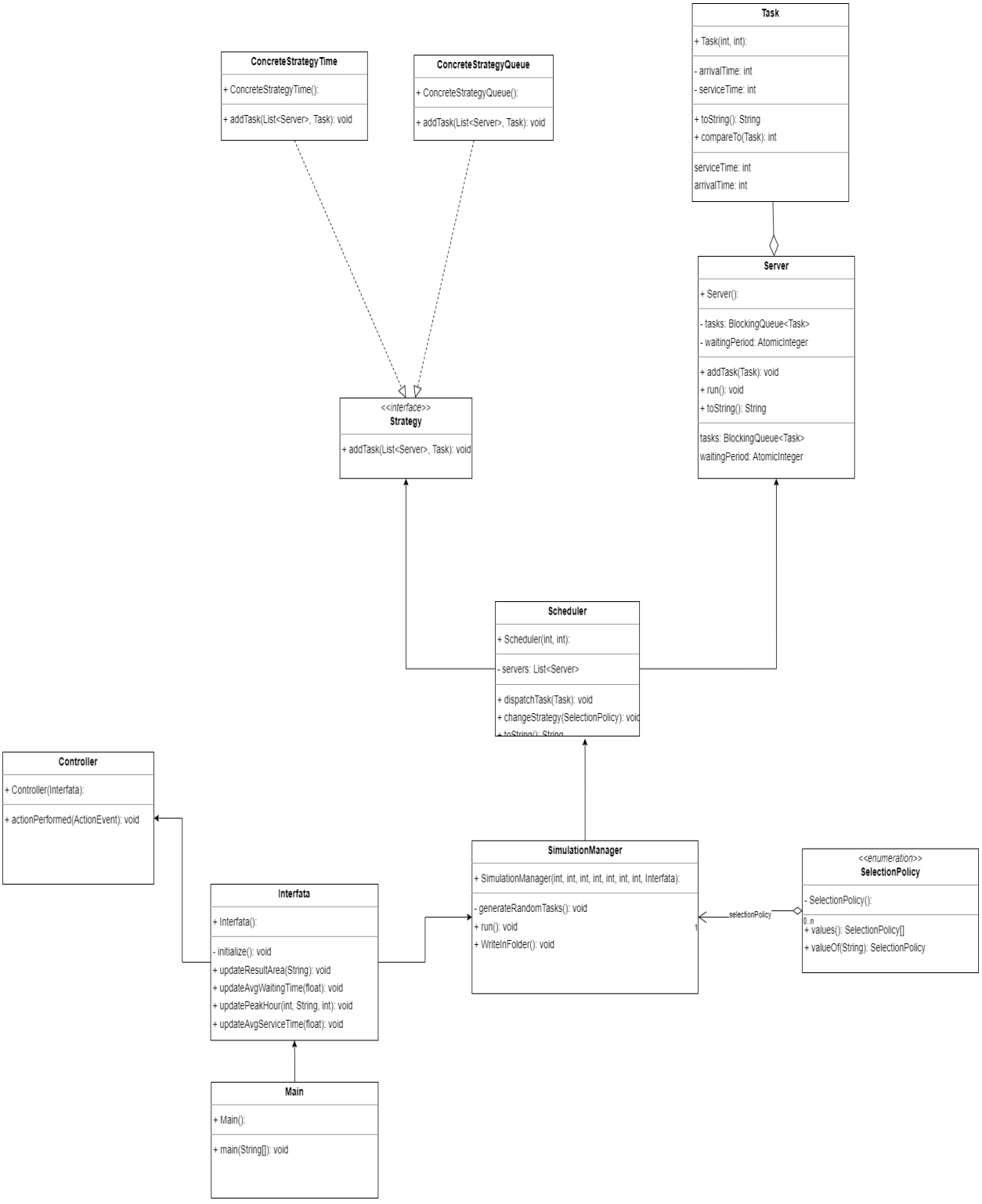


DIAGRAMA DE CLASE:



Structurile de date folosite sunt :

- BlockingQueue<Task> => a fost necerasa utilizarea acesteia pentru a tine evidenta task-urilor unui server, adica pentru a le stoca.
- ArrayList<Task> => folosit pentru a parcurge task-urile fiecarui server
- AtomicInteger => ajuta la memorarea timpului de asteptare la una dintre cozi
- List<Server> servers => stocarea datelor despre cozi

4. Implementare

1) Task

Atribute => arrivalTime, serviceTime sunt cele care stabilesc valibilitatea unui task

Metode => este vorba doar despre un constructor cu parametrii, gettere si settere, metoda toString suprascrisa si metoda compareTo in care se face comparatia intre doua task uri generate random, dupa timpul de arrival

Cod din clasa Task:

```
@Override
public String toString() { return "T{" + "AT=" + arrivalTime + ", ST=" + serviceTime + '}'; }

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

2) Server

Atribute => tasks, reprezinta cozile de Task-uri, si waitingPeriod-ul care dupa nume este este sugestiv si expune ca in el o sa se tina perioada de asteptare a unui task

Metode => tot constructor dar fara parametrii, gettere si settere, metoda toString care scrie stratutul unei cozi daca este inchisa sau are task-uri si metoda de addTask care adauga un task la o coada si seteaza waitingPeriod ul si metoda run, apartinand clasei Runnable, care executa sarcinile preluate de la client in mod continuu. Aceasta verifica daca exista un task la coada, se va obtine inceptul cozii prin peek, se va astepta pentru un timp egal cu durata de serviciu a task ului iar dupa se elemina din coada task ul prin remove iar daca nu este, se astepta pana la adaugarea unuia in coada

Cod pentru clasa Server :

```
public void addTask(Task newTask) {
    tasks.add(newTask);
    waitingPeriod.addAndGet(newTask.getServiceTime());
}

public void run() {
    while (true) {
        if (tasks.size() != 0) {
            Task aux = this.tasks.peek();
            try {
                Thread.sleep( millis: aux.getServiceTime() * 1000L);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            this.tasks.remove();
            this.waitingPeriod.set(waitingPeriod.get() - aux.getServiceTime());
        }

    }
}

public String toString() {
    if (tasks.size() == 0)
        return "Close";
    return "Tasks=" + tasks;
}
```

3) Strategy

Este scrisa doar o metoda ce va urma a fi implementata in alta clasa si aceasta este addTask(List<Servers> servers, Task t). Va avea scopul de a adauga un task t la o lista de serveri (cozile propriu zise).

4) ConcreteStrategy Queue/Time && SelectionPolicy

Aceste clase sunt oarecum identice. Au acelasi scop doar abordarea difera. Amandoua implementeaza metoda din Strategy de addTask dar in doua moduri diferite. In timp ce metoda de addTask din ConcreteStrategyQueue adauga task ul la coada de dimensiune mai mica, cealalta metoda de addTask din clasa ConcreteStrategyTime are ca scop adaugarea unui task la o coada cu timpul de asteptare total cel mai mic, indiferent de numarul de clienti din coada.

SelectionPolicy este doar un enum in care se afla tipul de simulare de vrem sa-l alegem, dupa timp ul cat mai scurt sau dupa marimea cozilor.

Cod al claselor ConcreteStrategy Queue/Time:

```
no usages
public class ConcreteStrategyQueue implements Strategy{

    1 usage
    public void addTask(List<Server> servers, Task t){
        int minQueue = Integer.MAX_VALUE;
        int index = 0;
        int indexS = 0;
        for(Server server: servers){
            if(server.getTasks().size() < minQueue){
                indexS = index;
                minQueue = server.getTasks().size();
            }
            index++;
        }
        servers.get(indexS).addTask(t);
    }
}
public class ConcreteStrategyTime implements  Strategy{
    1 usage
    @Override
    public void addTask(List<Server> servers, Task t) {
        int indexMin = 0;
        int index = 0;
        int minTime = Integer.MAX_VALUE;
        for(Server server: servers){
            if(server.getWaitingPeriod().get() < minTime){
                minTime = server.getWaitingPeriod().get();
                indexMin = index;
            }
            index++;
        }
        servers.get(indexMin).addTask(t);
    }
```

5) Scheduler

Atribute => List<Server>, maxNoServers, maxTaskPerServer, strategy, aceste obiecte Sunt bine denumite avand o clar scopul

Metode => costructorul cu parametri unde se initializeaza si array-ul de cozi si se pornesc thread-urile, gettere si settere, metoda de changeStrategy pentru a seta dupa ce mod ne dorim sa simulam cozile, toString si dispatchTask care apeleaza metoda unueia dintre clasele de ConcreteStrategy pentru a adauga task-ul.

Cod pentru clasa Scheduler :

```
public void changeStrategy(SelectionPolicy policy){
    if(policy == SelectionPolicy.SHORTEST_QUEUE){
        strategy = new ConcreteStrategyTime();
    }
    else if(policy == SelectionPolicy.SHORTEST_TIME){
        strategy = new ConcreteStrategyTime();
    }
}

1 usage
public void dispatchTask(Task task){
    strategy.addTask(servers, task);

}

1 usage
public List<Server> getServers(){
    return servers;
}

public String toString() {
    String rezultat = "";
    int index = 1;
    for (Server server : servers) {
        rezultat += "Queue " + index + ": " + server.toString() + "\n";
        index++;
    }
    return rezultat;
}
```

6) SimulationManager

Atribute => maxArrivalTime, minArrivalTime, maxProcessingTime, minProcessingTime, timeLimit, numberOfServers, numberOfClients, selectionPloicy, scheduler , generatedTask , view;

Metode => constructor in care se initializeaza fiecare obiect declarant in clasa, metoda run care va rula continuu si in functie de numarul de task-uri vor fi redirectionate catre servers (cozi). Aceasta metoda va calcula ora de varf, si timpul mediu de asteptare si de servire si va scrie in Interfata si in fisier. prin apelarea metodei WriteInFolder, rezultatele simularii. O alta metoda din aceasta clasa este generateRandomTasks care are ca scop generarea unor

task-uri noi, tinand cont de camurile de intrare, prin folosirea lui Math.random care ofera un numar random pentru ArrTime si SerTime Cuprins intre maxArrivalTime & minArrivalTime si maxProccessingTime & minProccessingTime date de utilizator.

Cod din aceste clasa:

```
private void generateRandomTasks() {

    for (int i = 0; i < numberOfClients; i++) {
        int randomArrival = (int) (Math.random() * (maxArrivalTime - minArrivalTime)) + minArrivalTime;
        int randomService = (int) (Math.random() * (maxProcessingTime - minProcessingTime)) + minProcessingTime;
        Task task = new Task(randomArrival, randomService);
        generatedTask.add(task);
        //task.toString();
    }

    Collections.sort(generatedTask);
}

public void run() {
    int currentTime = 1;
    float serviceTime = 0;
    float sumOfPeriods = 0;
    float averageWaitingTime = 0;
    int peakTime = 0;
    int peakTasks= 0;
    int sumOfTasks = 0;
    String result = "";
    try {
        for (Task task : generatedTask) {
            serviceTime += task.getServiceTime();
        }
        serviceTime /= (numberOfClients);
 serviceTime /= (numberOfClients);
 while (currentTime <= timeLimit) {
     if (!generatedTask.isEmpty()) {
         for (int i = 0; i < numberOfClients; i++) {
             if (generatedTask.size() > 0 && generatedTask.get(0).getArrivalTime()==currentTime) {
                 scheduler.changeStrategy(selectionPolicy);
                 scheduler.dispatchTask(generatedTask.get(0));
                 generatedTask.remove( index: 0);
             }
         }

     }
     //System.out.println("Current time: " + currentTime);
     String waitingTasks = "Current time: " + currentTime + "\nWaiting tasks: ";
     for (Task task : generatedTask) {
         waitingTasks += task.toString() + ", ";
     }
     result = waitingTasks + " \n" + scheduler.toString() + "\n";
     System.out.println(result);
     if (view != null) {
         view.updateResultArea(result);
     }
     sumOfTasks = 0;
     for(Server s: scheduler.getServers()) {
         sumOfTasks += s.getTasks().size();
         sumOfPeriods += s.getWaitingPeriod().get();
     }
     averageWaitingTime = sumOfPeriods / (numberOfServers * timeLimit);
     if(peakTasks < sumOfTasks){
         peakTasks = sumOfTasks;
         peakTime = currentTime;
```

```
            }
            currentTime++;
            try {
                Thread.sleep( millis: 1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

    }catch (Exception e){
        throw new RuntimeException(e);
    }
    if (view != null) {
        view.updateAvgServiceTime(serviceTime);
        view.updateAvgWaitingTime(averageWaitingTime);
        view.updatePeakHour(peakTime,  text2: " nr tasks: ", peakTasks);
    }
    WriteInFolder();

}
1 usage
public void WriteInFolder() {
    try {
        FileWriter fileWriter = new FileWriter( fileName: "simulari.txt");
        fileWriter.write(this.view.resultArea.getText());
        fileWriter.write(this.view.avgWT.getText());
        fileWriter.write(this.view.avgST.getText());
        fileWriter.write(this.view.peakH.getText());
        fileWriter.close();
    } catch (Exception e) {
        System.out.println(e);
    }
}
```

7) Interfata

Atribute => 4 butoane: Launch (are rolul de a porni simularea si automat de a face anumite verificari) Fill fields for1,2,3 (usureaza viata user-ului introducand direct datele in textField-uri pentru anumite simulari) si DeleteAll (scop de a sterge toate datele introduce si afisate ale simularii trecute)

=> 7 textField-uri asociate cu 7 label-uri pentru introducerea de date
=> 3 label-uri pentru afisarea datelor despre mediile de servicii, ora de varf si durata medie de astepatre.
=> 1 textArea folosit pentru afisarea evolutiei cozilor la care i-a fost atajat un JScrollPanel pentru a putea vizualiza toate secventele simularii.

Metode => 4 metode de updateResult pentru ultimele 2 componente despre care am vorbit la atribute, in care se mentine textul anterior si se adauga cel transimis din clasa SimulationManager.

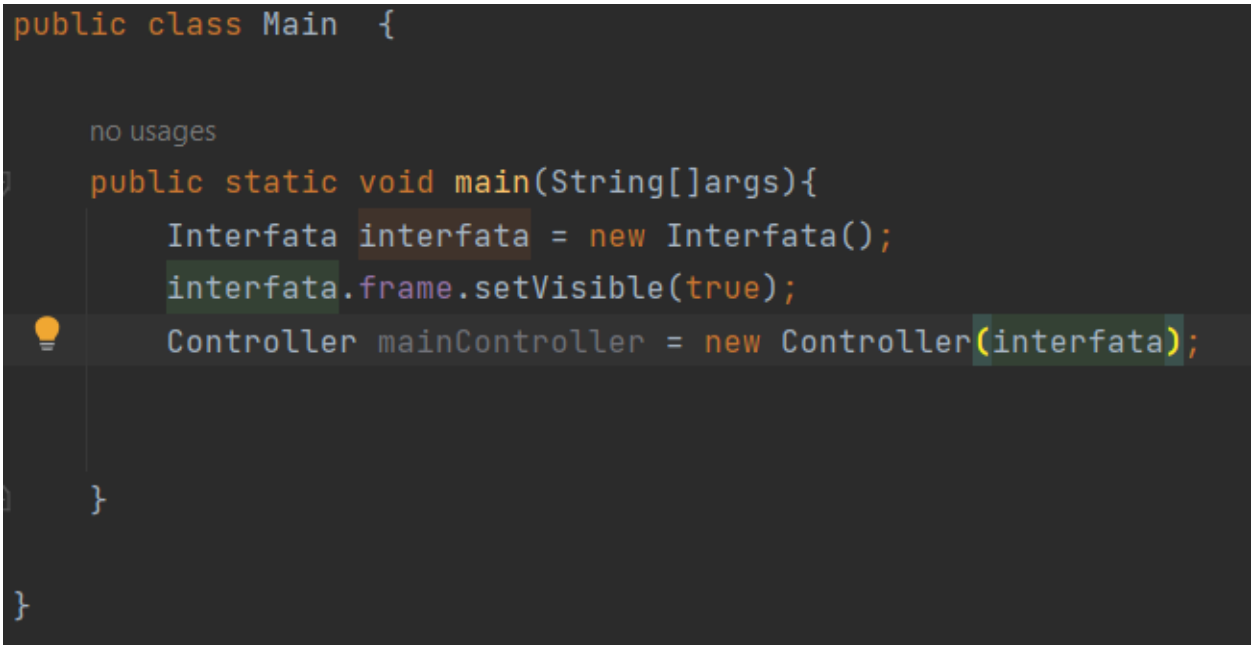
Cod din aceasta clasa :

```
1 usage
public void updateResultArea(String text) {
    resultArea.setText(resultArea.getText() + text);
}

1 usage
public void updateAvgServiceTime(float var) {
    avgST.setText(avgST.getText() + var + "\n");
}

1 usage
public void updateAvgWaitingTime(float var) { avgWT.setText(avgWT.getText() + var +"\n"); }

1 usage
public void updatePeakHour(int var, String text2, int var2) {
    peakH.setText(peakH.getText() + var + text2 + var2 + "\n");
}
```

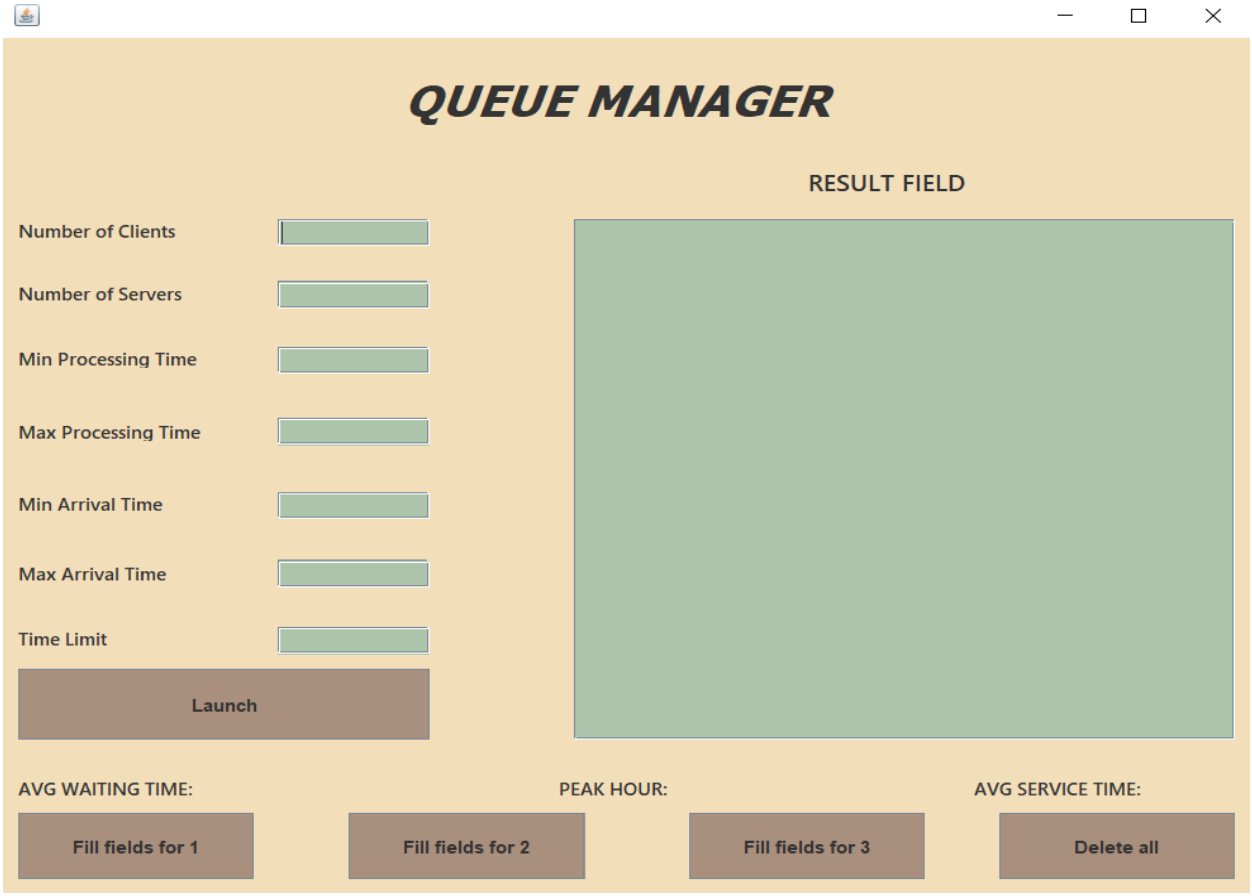
8) Main si Controller

In clasa Controller se realizeaza actionPerformed-ul butonului Launch si aici sunt verificate anumite conditii in care programul nici nu trebuie sa porneasca. Dupa tura de validari toate datele preluate din textField-uri sunt transmise catre SimulationManager unde se incepe executia procgrmului.

In Clasa Main se creaza o noua interfata si este transmisa catre controller care va decide daca incepe simularea dupa introducerea datelor sau nu.



9) Design



5. Rezultate

Testele principale efectuate au fost cele din indrumatorul pentru aceasta tema si rezultatele au fost stocate in fisierele respective fiecaruia test1, test2, test3:

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

Rezultate test1:

```
AVG WAITING TIME:0.175
AVG SERVICE TIME:2.25
PEAK HOUR:17 nr tasks: 2
```

Rezultate test 2:

```
AVG WAITING TIME:3.33
AVG SERVICE TIME:3.44
PEAK HOUR:21 nr tasks: 12
```

Rezultate test 3:

```
AVG WAITING TIME: 107.665
AVG SERVICE TIME: 5.39
PEAK HOUR: 99 nr tasks: 671
```

6. Concluzii

Din punctul meu de vedere tema a fost destul de dificil de inteles, probabil acest lucru a ingreunat procesul temei dar am invatat sa foloses thread-uri si variabile cincronizate amintindu-mi de cozi.

O posibila dezvoltare a temei de inaintare in coada a unui client cu serviceTime-ul foarte mic pentru a nu astepta dupa altii cu serviceTime-ul mai mare, chiard aca arrivalTime-ul pentru cel care cu serviceTime ul mai mic este mai mare decat al celuilalt cu serviceTime-ul mai mare.

7. Bibliografie

- *https://app.diagrams.net*
- *https://dsrl.eu*