Universitatea Tehnică din Cluj-Napoca
Facultatea de Automatică și Calculatoare

DOCUMENTAȚIE

TEMA NUMĂRUL 2

Nume student: Maria-Magdalena Creț

Grupa 30223

Cuprins

1 Obiectivul temei

Obiectivul principal a acestei teme este acela de a se proiecta și implementa o aplicație care are scopul de a gestiona cozilor și care atribuie clienții la cozi astfel încât timpul de așteptare să fie minimizat. Aplicația conține o interfață grafică realizată cu GUI care permite utilizatorului să introducă datele necesare pentru generarea cozilor în mod dinamic, respectiv să se selecteze strategia de punere la coadă. obiectul principal este ca studentul să se familiarizeze cu conceptele de threads, pattern strategy, atomicity, synchronized statement, synchronized methods, cât și cu realizarea aplicaților de tip GUI. Pentru implementarea interfeței grafice există posibilitatea alegerii între Java Swing sau JavaFX.

Cerința acestei teme este următoarea:

Proiectați și implementați o aplicație de gestionare a cozilor care atribuie clienții la cozi astfel încât timpul de așteptare să fie minimizat. Cozile 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 să adăugați 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 gestionare a cozilor ar trebui să simuleze (prin definirea unui timp de simulare $t_{simulation}$) o serie de N clienți care vin pentru serviciu, intră în Q cozi, așteaptă, sunt serviți și, în cele din urmă, părăsesc cozile. Toți clienții sunt generați când simularea începe și sunt caracterizați de trei parametri: ID (un număr între 1 și N), $t_{arrival}$ (timpul de simulare când sunt gata 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 atunci când timpul său de $t_{arrival}$ este mai mare sau egal cu timpul de simulare ($t_{arrival} \geq$t$_{simulation}$).

Următoarele date ar trebui considerate ca date de intrare pentru aplicație și ar trebui introduse de către utilizator în interfața de utilizare a aplicației:

- Numărul de clienți (N)
- Numărul de cozi (Q)
- Intervalul de simulare ($t_{\max}^{simulation}$)
- Timpul minim și maxim de sosire ($t_{\min}^{arrival} \leq t_{arrival} \leq t_{\max}^{arrival}$)
- Timpul minim și maxim de servire ($t_{\min}^{service} \leq t_{service} \leq t_{\max}^{service}$)

1.1 Obiectivele secundare ale acestei teme sunt:

1. Asigurarea unei interfețe grafice prietenoase și ușor de folosit pentru utilizator, care să permită introducerea și vizualizarea datelor de mai sus într-un mod ușor și intuitiv. Acest lucru realizându-se după modelul MVC (Model View Controller).

2. Implementarea logicii pentru așezarea clienților la cozi cât și a strategiilor utilizate pentru acest lucru într-un mod eficient astfel încât să se ofere o performanță bună a aplicației, indiferent de numărul de clienți sau de cozi ales.
3. Realizarea claselor necesare funcționării aplicației de gestionare a cozilor.
4. Realizarea unei bune organizări a codului, a claselor, pentru a putea face ca acest cod sa fie cât mai ușor și accesibil de reutilizat sau modificat în caz de nevoie.
5. Documentarea temei, redarea informațiilor corespunzătoarea realizării acesteia, pentru ca aplicație sa fie cât mai ușor de înteles și utilizat de orice persoană.
6. Folosirea strategy patterns, a metodelor și/sau a variabilelor sincronizate, atomice, înțelegerea funcționării thread-urilor.

2 Analiza problemei, modelare, scenarii, cazuri de utilizare

2.1 Analiza problemei

1. Cerințele funcționale:
   - utilizatorul poate să introducă parametrii pentru simulare;
   - parametrii de simulare sunt validați înainte ca simularea să înceap;
   - utilizatorul poate să înceapă simularea;
   - simularea afișează dinamic (în mod real) așezarea clienților la cozi;
   - se creează un fișier textse suprascriu datele simulării pentru fiecare moment de timp când are loc aceasta;
2. Cerințele non-funcționale:
   - aplicația trebuie să fie intuitivă și ușor de utilizat
   - afișarea evoluției cozilor în timp real să fie ușor de înțeles de utilizator

2.2 Modelarea problemei pentru scrierea codului Java

Așadar pentru a simplifica rezolvarea problemei și pentru a crea o structură cât mai ușor de analizat, unde se pot realiza într-un mod accesibil și rapid modificări, s-a ales organizarea codului Java în pachete(packages):

- business_logic: se integreaza clasele care conțin implementarea codului. Clasele sunt:
  - ConcreteStrategyTime , care conține implementarea strategiei în funcție de timp - Clientul se va așeza la coada unde restul clienților termină în cel mai scurt timp;

  - ShorthestQueueStrategy , care conține implementarea strategiei în funcție de numărul de clienți aflați la coadă;
  - SelectionPolicy, un enum pentru alegerea startegiei de către client;
  - SimulationManager, clasa în care s-a implementat logica pentru simularea cozilor și asezarea clienților la acestea;
  - Strategy, o interfață care implementează metoda de addTask = adăugarea unui client la o coadă;
- exception: se integreaza clasa ValidationException pentru afișarea mesajelor de eroare atunci când este cazul: extinde superclasa Throwable, pentru toate erorile și excepțiile limbajului Java. Această clasă conține un constructor super(), pentru a putea transmite un mesaj atunci când are loc excepția.
- model:
  - Client, această clasă reține informațiile despre clienți;
  - Queue , această clasă reține informațiile despre cozi;
- gui_interface: se consideră ca fiind un pachet pentru că acesta conține clasa GUI, unde s-a implementat codul pentru design-ul aplicației. Pentru interfața grafică s-a ales implementarea cu Java Swing.

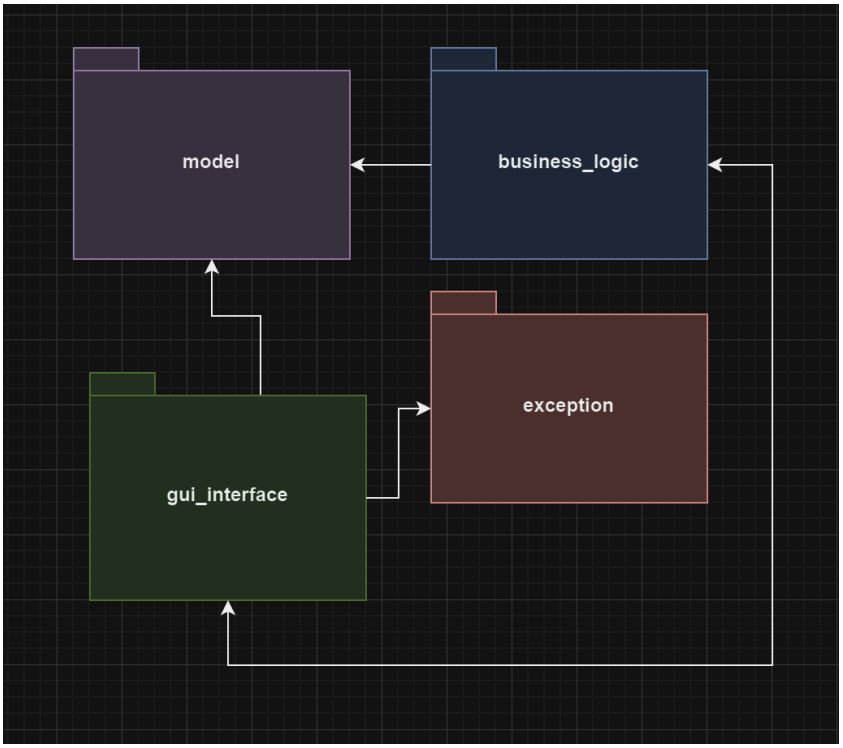

Figura 1: Diagrama Pachete

3 Proiectare

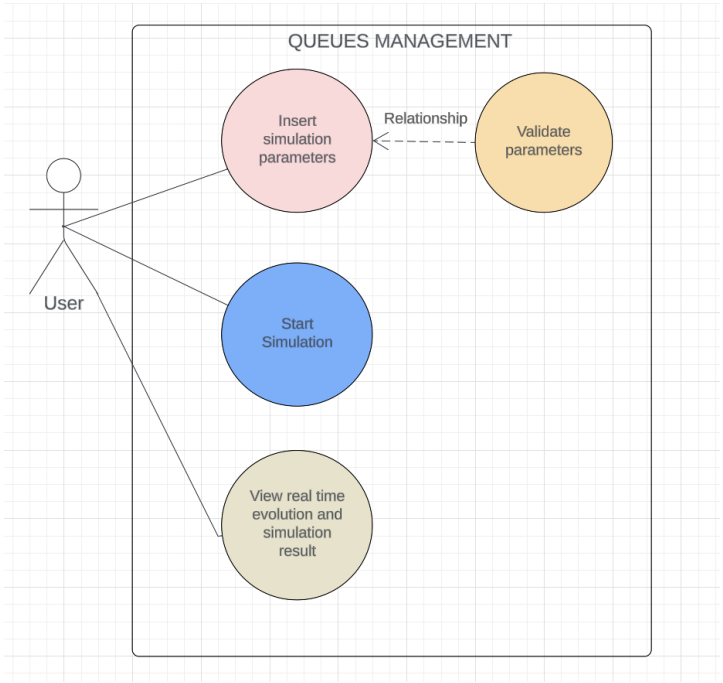

Figura 2: Diagrama Use-Case

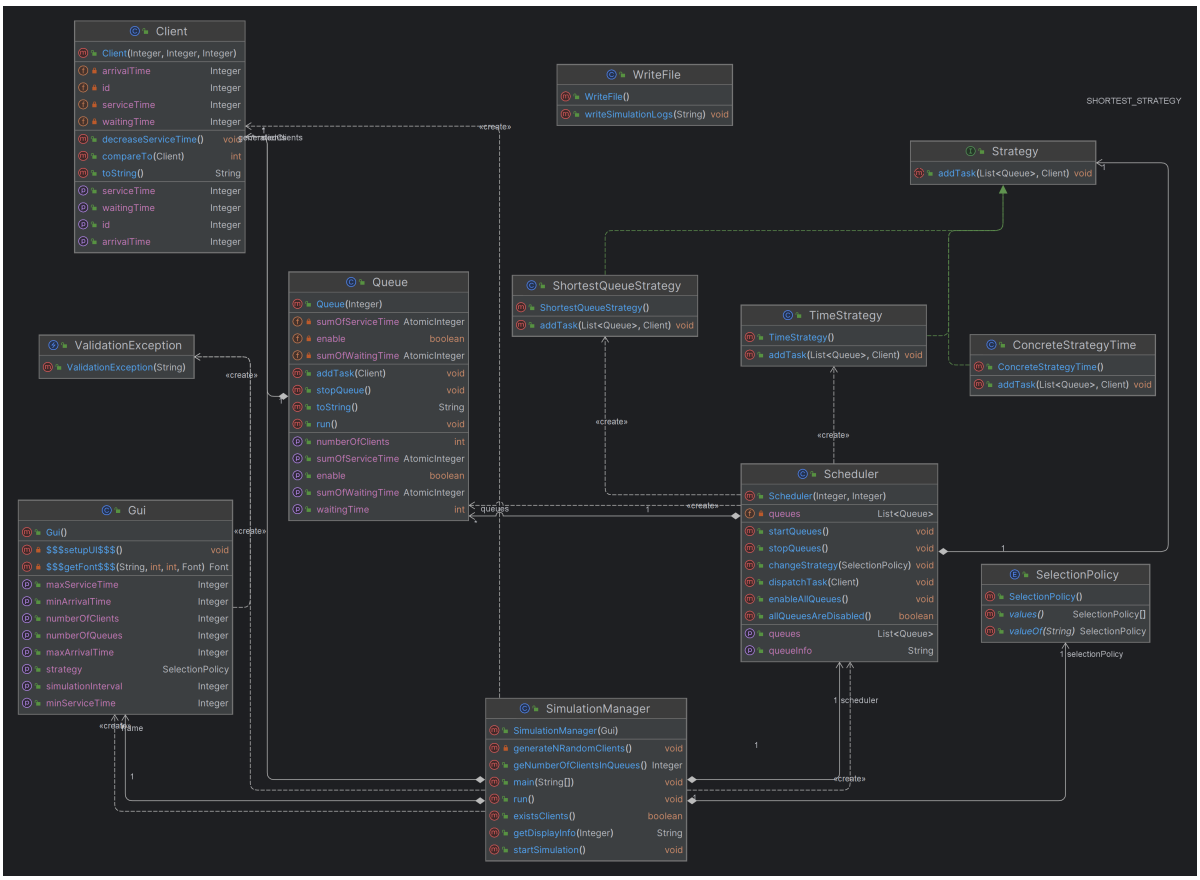

Figura 3: Diagrama UML pentru clase

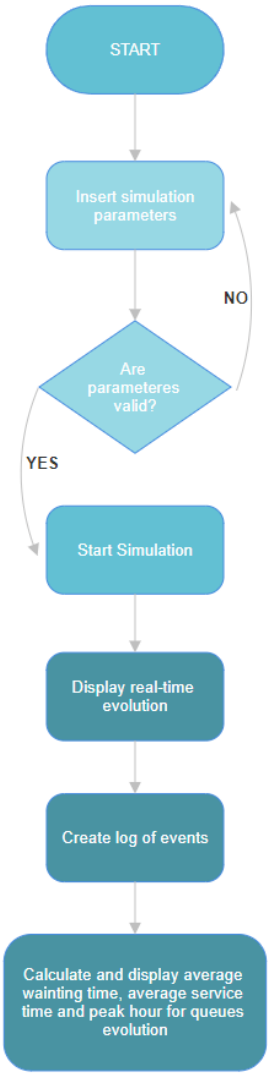

Figura 4: Flow Chart

4 Implementare

Implementarea codului s-a realizat structural, pe baza a 3 pachete menționate mai sus și pachetul de Swing UI Designer: gui_interface, unde s-a realizat implementarea pentru interfața grafică.

Structuri de date folosite:

(a) BlockingQueue: permite comunicația și sincronizarea între thread-uri, oferind metode de blocare care așteaptă să pună sau să extragă elemente din coadă în cazul în care aceasta este goală sau plină. Este utilizată pentru a menține clienții în coadă.

(b) AtomicInteger: este folosit pentru câmpurile din clasa Queue și permite operații

aritmetice asupra unei variabile întregi într-un mod sigur pentru thread-uri concurente, fără a necesita sincronizare manuală.

Algoritmi folosiți:

- Algoritmul pentru determinarea cozii cu cel mai scurt timp de așteptare.
- Algoritmul pentru determinarea cozii cu cel mai mic de număr de persoane.

Clasele conținute de proiect sunt următoarele:

(a) Clasa *Scheduler*:conține metodele de pornire și oprire a threadurilor *startQueues* și *stopQueues*, un constructor unde se calculează numărul maxim de cozi și numărul maxim de clienți la cozi. Tot în această clasă este metoda de *changeStrategy*, unde pe baza enumului *SelectionPolicy* se alege strategia după care se va așeza un client la o coadă (uitilizarea de strategy patterns). Precis, în programare, strategy pattern este un șablon de proiectare software comportamental care permite selectarea unui algoritm la runtime. În loc să implementeze direct un singur algoritm, codul primește instrucțiuni la runtime cu privire la care algoritm dintr-o familie de algoritmi să folosească. Atfel că în funcție de strategie adaug clientul la o coadă în metoda implementată tot în această clasă *dispatchTask*.

```
1  public class Scheduler {
2
3      private static final Strategy SHORTEST_STRATEGY = new
           ShortestQueueStrategy();
4      private static final Strategy TIME_STRATEGY = new
           TimeStrategy();
5
6      private final List<Queue> queues;
7      private Strategy strategy = SHORTEST_STRATEGY;
8      private Integer maxNumberOfServers;
9      private Integer maxNumberOfTasksForServer;
10
11     private ExecutorService executor;
12
13     public Scheduler(Integer maxNumberOfServers, Integer
           maxNumberOfTasksForServer) {
14         queues = new ArrayList<>();
15         for (int i = 0; i < maxNumberOfServers; i++) {
16             queues.add(new Queue(maxNumberOfTasksForServer));
17         }
18         this.maxNumberOfServers = maxNumberOfServers;
19         this.maxNumberOfTasksForServer =
               maxNumberOfTasksForServer;
20     }
21
22     public void startQueues() {
23         executor = Executors.newFixedThreadPool(queues.size()
               );
24         for (Queue queue : queues) {
```

```
25              executor.execute(queue); //se pornesc threadurile
26          }
27      }
28
29      public void stopQueues() {
30          for (Queue queue : queues) {
31              queue.stopQueue(); //sse opresc threadurile
32          }
33          executor.shutdown();
34      }
35
36
37      public void  enumulu(SelectionPolicy selectionPolicy) {
38          if (selectionPolicy == SelectionPolicy.SHORTHEST_TIME
                ) {
39              strategy = SHORTEST_STRATEGY;
40          } else {
41              strategy = TIME_STRATEGY;
42          }
43      }
44
45      public String getQueueInfo() {
46          String display = "";
47          int numberOfQueue = 1;
48          for (Queue queue : queues) {
49              display += "Queue " + numberOfQueue + ": " +
                    queue.toString() + "\n";
50              numberOfQueue++;
51          }
52
53          return display;
54      }
55
56      public void dispatchTask(Client client) {
57          strategy.addTask(queues, client); // in functie de
                strategie adaugam taskul la o coada
58      }
```

În această clasă, o altă metodă foarte importantă este *getQueueInfo()*, care permite scrierea cozilor cu informații dupa modul de mai sus. Se verifică dacă cozile sunt enable sau nu.

Observație: S-au utilizat thread pools din următoarele motive:

- Un thread pool este o colecție de fire pre-create care sunt pregătite să execute sarcini atribuite de un fir principal sau un planificator. Grupul de fire de execuție menține o coadă de sarcini care așteaptă să fie executate și le atribuie firelor inactive din grup. Atunci când un fir termină o sarcină, acesta se întoarce la grup și așteaptă o altă sarcină.

- Thread pools au mai multe avantaje față de crearea și distrugerea threadurilor la cerere: reduc costul și durata creării și distrugerii firelor, care pot fi scumpe și consumatoare de timp, îmbunătățesc performanța și simplifică logica de programare și evită problemele de concurență prin abstractizarea detaliilor gestionării firelor din aplicație.

(b) Clasa *ShortestQueueStrategy*: implementează metoda de addTask din interfața Strategy și mai exact, crează strategia în funcție de cozile care au cel mai mic număr de clienți. Este utilizată pentru a alege la ce coadă să se pună clientul.

```
1  @Override
2      public void addTask(List<Queue> queues, Client client) {
3          if (!queues.isEmpty()) {
4              int minNumberOfTasks = Integer.MAX_VALUE;
5              Queue queueWhereTaskIsAdded = null;
6
7              for (Queue queue : queues) {
8                  if (minNumberOfTasks > queue.
                       getNumberOfClients()) {
9                      minNumberOfTasks = queue.
                           getNumberOfClients();
10                     queueWhereTaskIsAdded = queue;
11                 }
12             }
13
14             queueWhereTaskIsAdded.addTask(client);
15         }
16     }
```

(c) Clasa *TimeStrategy*: implementează metoda de addTask din interfața Strategy și mai exact, crează strategia în funcție de timpul de așteptare al clienților la coadă până să fie serviți. Este utilizată pentru a alege la ce coadă să se pună clientul.

```
1  @Override
2      public void addTask(List<Queue> queues, Client client) {
3          if (!queues.isEmpty()) {
4              int minTime = Integer.MAX_VALUE;
5              Queue queueWhereTaskIsAdded = null;
6
7              for (Queue queue : queues) {
8                  if (minTime > queue.getWaitingTime()) {
9                      minTime = queue.getWaitingTime();
10                     queueWhereTaskIsAdded = queue;
11                 }
12             }
13
14
15             queueWhereTaskIsAdded.addTask(client);
16         }
```

```
17      }
```

Waiting Time-ul unui client se calculează pe baza timpilor de servire a clienților care sunt la coadă înaintea aceastuia, scăzând astfel odată cu servirea unui client.

```
1  waitingPeriod.set(waitingPeriod.get() + client.getServiceTime
       ())
```

Obținerea waiting time-ului se realizează printr-un getter, care ia waitingPeriod-ul clientului setat anterior.

```
1      public int getWaitingTime() {
2          return waitingPeriod.get();
3      }
```

(d) Clasa *SimulationManager*: În acestă clasă se găsește metoda run pentru threadS pools, dar și metoda *generateNRandomClients*, care generează în mod aleatgor clienți care urmează să fie prelucrați și adăugați în cozi.Totadată acesta clasa conține main-ul, fiind clasa de unde se compilează programul și este clasa în care are loc simularea. În metoda run din această clasă, s-a adăugat un thread.sleep(5), pentru ca să se aștepte ca toate cozile să proceseze o secundă, cea care este cerută pentru sleep-ul thread-urilor.Acest lucru se întâmplă pentru fiecare thread. În această clasă se calculează și media aritmetică a timpilor de așteptare și de servire și peak hour-ul, așa cum se poate vedea mai jos.

```
1   @Override
2      public void run() {
3          int currentTime = 0;
4          startSimulation();
5
6          Integer peakTime = 0;
7          int numberMaxOfClientsOnQueueInSimulation = 0;
8          while (currentTime < timeLimit && existsClients()) {
9
10             boolean ok = true;
11             while (ok && !generatedClients.isEmpty()) {
12                 Client client = generatedClients.get(0);
13                 if (client.getArrivalTime() == currentTime) {
14                     scheduler.dispatchTask(client);   //se
                           face remove cat timp sunt clienti care
                            au ArrivalTime() == currentTime
15                     generatedClients.remove(client);
16                 } else {
17                     ok = false;
18                 }
19             }
20
21             // Enable Queue processing
22             // Wait that all queues has proceeded the client,
                   but only for 1 second
```

```
23              scheduler.enableAllQueues();
24              while (!scheduler.allQueuesAreDisabled()) {
25                  try {
26                      Thread.sleep(5); //astept pana toate
                           cozile au procesat o secunda, sleep-ul
                            pentru fiecare thread
27                  } catch (InterruptedException e) {
28                      System.out.println("Error at sleeping the
                            simulation manager for waiting to
                           process queues");
29                  }
30              }
31
32              Integer allNumberOfClients =
                   geNumberOfClientsInQueues();
33              if (numberMaxOfClientsOnQueueInSimulation <
                   allNumberOfClients) {
34                  numberMaxOfClientsOnQueueInSimulation =
                       allNumberOfClients;
35                  peakTime = currentTime;
36              }
37
38              String messageToDisplay = getDisplayInfo(
                   currentTime);
39              System.out.println(messageToDisplay);
40              frame.textArea.setText(frame.textArea.getText() +
                    "\n" + messageToDisplay);
41              currentTime++;
42          }
43
44          scheduler.stopQueues();
45
46
47          Integer sumWaiting = 0;
48          for (Queue queue : scheduler.getQueues()) {
49              sumWaiting += queue.getSumOfWaitingTime().get();
50          }
51          medWaitingTime = 1.0 * sumWaiting / numberOfClients;
52          frame.textFieldAverageWaitingTime.setText(
               medWaitingTime.toString());  //afisez average
               waintingTime
53
54
55          Integer sumService = 0;
56          for (Queue queue : scheduler.getQueues()) {
57              sumService += queue.getSumOfServiceTime().get();
58          }
59          medServiceTime = 1.0 * sumService / numberOfClients;
60          frame.textFieldAverageServiceTime.setText(
```

```
            medServiceTime.toString()); //afisez average
            serviceTime
61          frame.textFieldPeakHour.setText(peakTime.toString());
                //peak hour for the simulation interval
62
63          String logs = frame.textArea.getText();
64          logs+="\n"+"Average Waiting Time: "+ medWaitingTime;
65          logs+="\n"+"Average Service Time: "+ medServiceTime;
66          logs+="\n"+"Peak Hour: "+ peakTime;
67
68          WriteFile.writeSimulationLogs(logs);
69      }
```

(e) Clasa *WriteFile*: În acestă clasă se găsește metoda *writeSimulationLogs* care face posibilă scrierea în fișier a raportului generat în urma simulării cozilor cu clienți de la datele adăgate de la tastatura în interfață. Acest fișier se suprascrie odată cu generarea unei noi simulări. Având în vedere că simularea se realizează cu date random, respectând datele introduse de la tastatura, acest raport va fi de fiecare dată altfel. Se verifică dacă nu se poate scrie în fișierul text și în această cauză se aruncă un mesaj de eroare.

```
1  public class WriteFile {
2
3      private static final String FILE_NAME = "simulation_logs.
           txt";
4
5      public static void writeSimulationLogs(String logs) {
6          try {
7              BufferedWriter writer = new BufferedWriter(new
                   FileWriter(FILE_NAME));
8              writer.write(logs);
9              writer.close();
10         } catch (IOException e) {
11             System.out.println("Error at writing in file");
12         }
13     }
14 }
```

(f) Clasa *Gui*: S-a realizat o interfață grafică prietenoasă, ușor de utilizat. Are 7 field-uri pentru introducerea datelor necesare(numărul de clienți, numărul de cozi, intervalul de timp pentru simulare, timpii min și max de sosire, timpii min și max de servire) și un field pentru alegerea strategiei, unde exista cele două variante posibile (strategia în funcție de timp sau strategia în funcție de numărul de persoane de la cozi). Este un buton pentru pornirea simulării. Un Text Area pentru afișare, unde se va vedea în timp real evoluția cozilor cu clienți, realizată în mod dinamic în funcție de cum se așază clienții la cozi. S-au introdus si scroll bar-uri pentru Area Text pentru ca sa se poat vedea tot ce conțin cozile și numărul de clienți, iar acest panel text area să nu își modifice dimensiunea.

La final sunt 3 field-uri pentru afișarea timpul mediului de așteptare (s-a calculat ca fiind media aritmetică a timpilor de așteptarea a clienților la cozi), timpul mediu de servire (s-a calculat fiind media artimetică a timpilor de servire a clienților la cozi) și peak hour for simulation interval, care reprezintă momentul de timp la care cozile conțin cel mai mare număr de clienți. Aceste rezultate se vor afișa în căsuțele corespunzătoare odată ce s-a încheiat simularea.
S-au implementat metode în aceeași clasă care prind și aruncă excepții și care verifică ce caractere s-au introdus, iar în cazul în care s-au introdus caractere invalide se va afișa o eroare pe fereastră cu mesajul respectiv. Se verifică dacă au fost lasate câmpuri goale, dacă se introduce altceva decât numere de tipul Integer și dacă numărul de clienți, numărul de cozi, timpii sunt mai mici sau egali cu 0 .

```
1     public Integer getNumberOfClients() throws
          ValidationException {
2         Integer x;
3         try {
4             x = Integer.parseInt(textFiledNumberOfClients.
                  getText());
5         } catch (Exception exception) {
6             throw new ValidationException("Invalid number of
                  clients");
7         }
8
9         if (x <= 0) {
10            throw new ValidationException("Number of clients
                  must be >=1");
11        }
12
13        return x;
14    }
15
16    public Integer getNumberOfQueues() throws
          ValidationException {
17        Integer x;
18        try {
19            x = Integer.parseInt(textFieldNumberOfQueues.
                  getText());
20
21        } catch (Exception exception) {
22            throw new ValidationException("Invalid number of
                  queues");
23        }
24        if (x <= 0) {
25            throw new ValidationException("Number of queues
                  must be >=1");
26        }
27
28        return x;
29    }
```

De asemenea odată cu încheierea simulării și pornirea unei noi simulări, cele trei câmpuri de afișare și text area se curăță, permițând apariția noii simulări. Odată cu închiderea ferestrei, programul se oprește.

Figura 5: Interfața grafică

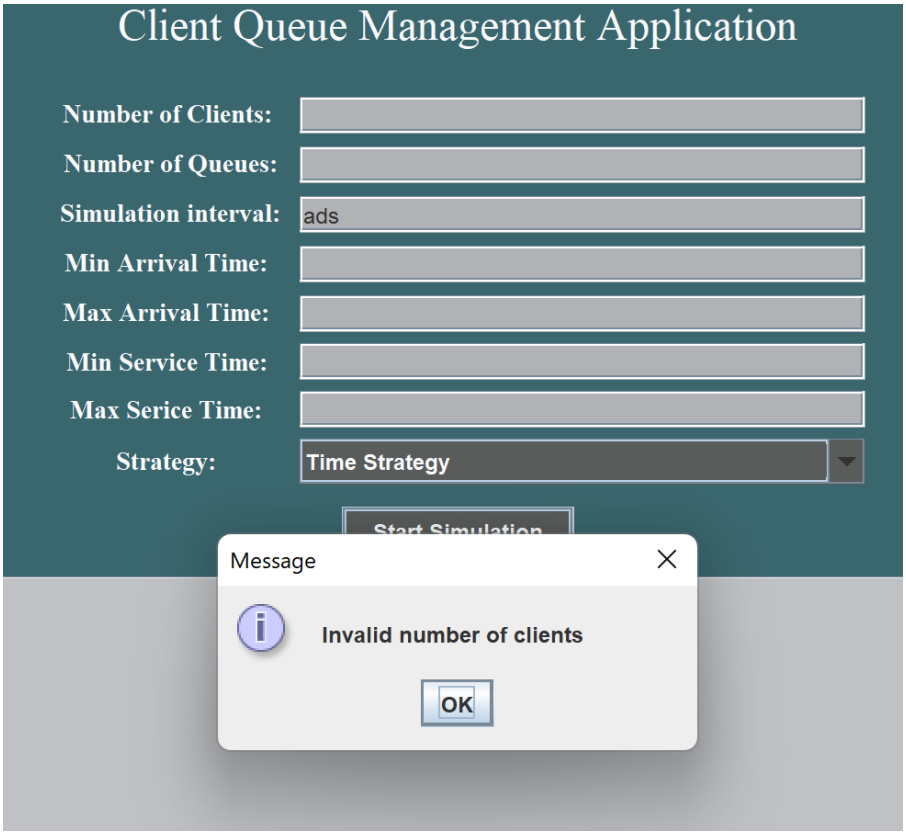

Figura 6: Afișarea unei eroare pentru introducerea caracterelor invalide

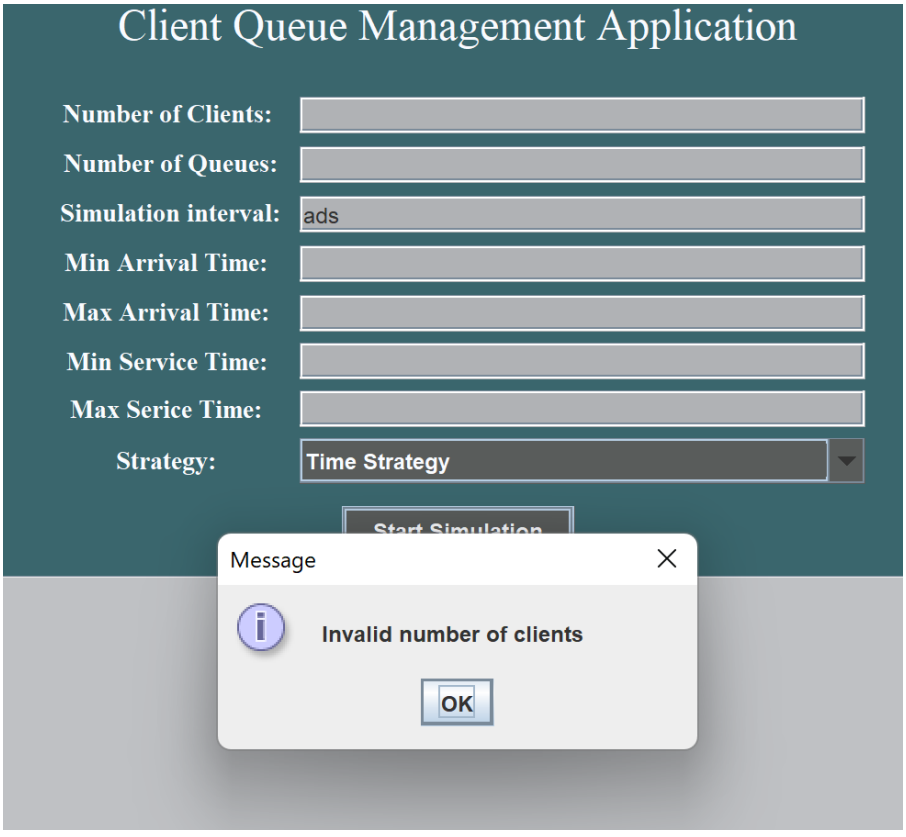

Figura 7: Afișarea unei eroare pentru adaugarea unei valori minime

necorespunzatoare

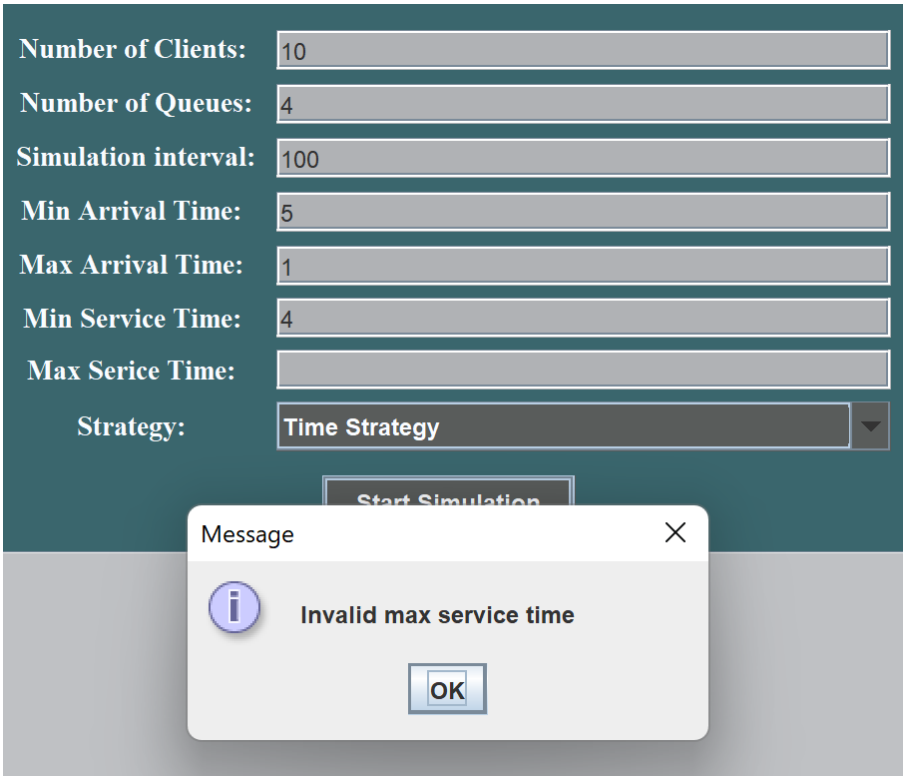

Figura 8: Afișarea unui rezultat pentru adăugarea unui field gol

5 Rezultate

Rezultatele se afisează în Text Area, care este un panel pentru text, unde se observă în mod dinamic cum se modifică cozile și care dintre clienți se atribuie acestor cozi. Totodată acest lucru se mai afișează la finalul simulării și într-un fișier .txt. Ultimele 3 field-uri din interfață au rolul de a se afișa pe acestea timpul mediu de așteptare, timpul mediu de servire și timpul la care cozile au avut cel mai mare număr de clienți. Exemplu: Un input este de forma: Number of Clients = 10, Number of Queues = 3, Simulation interval = 50, Min Arrival Time = 2, Max Arrival Time = 5, Min Service Time = 3, Max Service Time = 10, iar strategia aleasă va fi cea în funcție de timp *Time Strategy*. Rezultatul se va afișa sub forma:

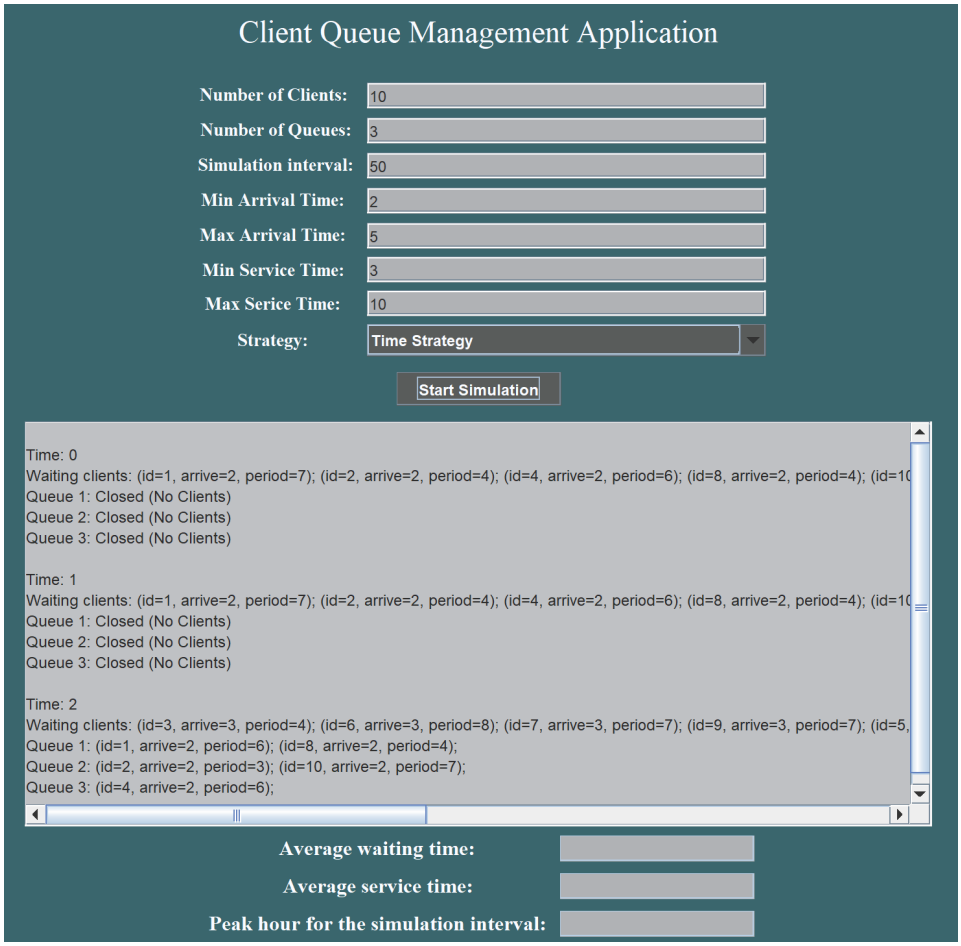

Figura 9.1: EXEMPLU Afișarea unui rezultat

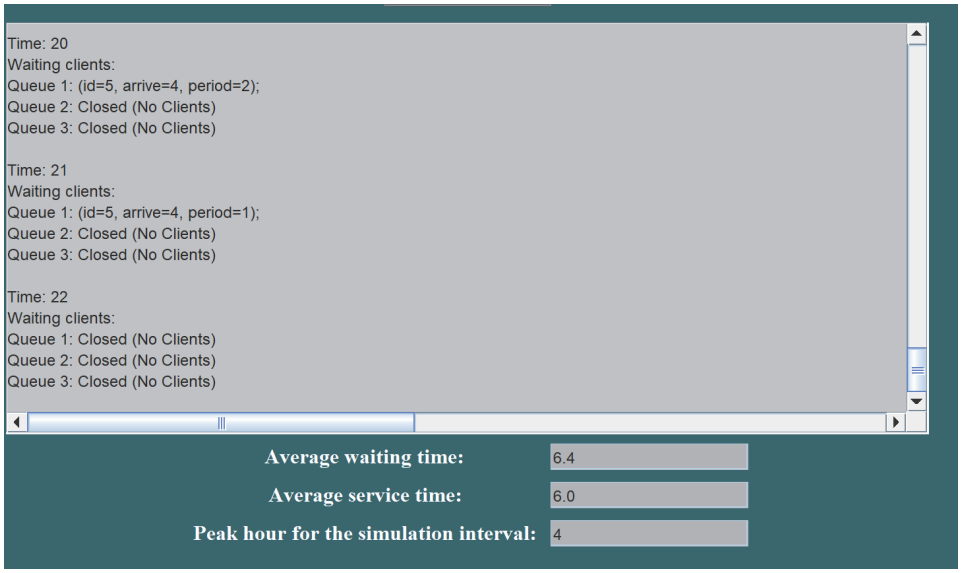

Figura 9.2: EXEMPLU Afișarea unui rezultat

6 Concluzii

În cele din urmă, se poate spune că, deși, crearea unui aplicației pentru managementul unor cozi poate părea la prima vedere o sarcină complicată, prin utilizarea tehnicilor de programare orientate pe Obiect, se poate obține o aplicație eficientă și utilă.
Îmbunătățiri ulterioare pot fi:

- Adăugarea mai multor strategii
- Calcularea mai multor rezultate ale simulării
- Adăugarea unor animații pentru evoluția simulării
- Interfață grafică îmbunătățită: pentru a fi mai intuitivă și ușor de utilizat. Adăugarea unor opțiuni de personalizare, cum ar fi selectarea culorilor temei, prezența unor butoane cu cifre pentru a face diferită datelor (de data aceasta, nu numai de la tastatură).
- O posibilă îmbunătățire priviind organizarea codului.

7 Bibliografie (Webografie)

(a) Cursuri OOP Anul2, Semestrul 1

(b) https://dsrl.eu/courses/pt/materials/lectures/

(c) https://www.geeksforgeeks.org/blockingqueue-interface-in-java/

(d) https://stackoverflow.com

(e) https://www.javacodegeeks.com/2013/01/java-thread-pool-example-using-executors-and-threadpoolexecutor.html

(f) https://www.geeksforgeeks.org/swingworker-in-java/

(g) https://www.overleaf.com/learn/latex

(h) https://libguides.eur.nl/overleaf/lists-tables-images-labelling