

PROGRAMMING TECHNIQUES

ASSIGNMENT 2

QUEUE BASED SYSTEM

Student: Chindea Miruna

Group: 30424

SUMMARY

1. Homework objective.....	1
1.1. Main objective.....	1
1.2. Secondary objectives.....	1
2. Problem analysis, scenarios, use cases.....	2
2.1. Functional requirements.....	2
2.2. Use cases.....	2
3. Design.....	4
3.1. Pattern.....	4
3.1.1. Model.....	4
3.1.2. View.....	4
3.1.3. Controller.....	4
3.2. Classes and packages.....	4
3.3. Interfaces.....	6
3.4. UML Diagram.....	7
3.5. Relationships.....	8
4. Implementation.....	9
4.1. Classes.....	9
4.1.1. Client.....	9
4.1.2. Queue.....	10
4.1.3. Simulator.....	11
4.2. Graphical User Interface.....	12
4.2.1. View.....	12
4.2.2. Controller.....	13
4.2.3. MVC.....	13
5. Results.....	16
6. Conclusions.....	17
7. Bibliography.....	17

1. Homework objective

1.1. Main objective

The main objective of the homework is to design and implement a simulation application aiming to analyze queuing based systems for determining and minimizing clients' waiting time. The main objective of a queue is to provide a place for a "client" to wait before receiving a "service". The management of queue based systems is interested in minimizing the time amount their "clients" are waiting in queues before they are served. The application should simulate a series of clients arriving for service, entering queues, waiting, being served and finally leaving the queue. It tracks the time the customers spend waiting in queues and outputs the average waiting time.

1.2. Secondary objectives

The secondary objectives of the homework, which will help in building the main objective, are:

1. Creating a Graphical User Interface for the system, in order to help the user interact with it.

The user will introduce the minimum and maximum interval of arriving time between customers, minimum and maximum service time, number of queues and simulation interval. The GUI will display the real-time queue evolution, as well as the log of events:

- time of arriving of each client, with client's service time;
- time of entering a queue, with queue's number;
- time of leaving the queue.

2. Simulation Setup from Graphical User Interface.

3. Using Multithreading for implementing the queues: one thread per queue.

4. Simulation results:

- average waiting time;
- average service time;
- peak hour;
- empty queue time.

These objectives will be described in Chapter 4.

2. Problem analysis, scenarios, use cases

2.1. Functional requirements

There will be displayed a graphical interface for the user to interact with. The user will introduce, in order:

- minimum and maximum arriving interval between clients;
- minimum and maximum service time for clients;
- number of queues;
- simulation interval.

If he introduced the above mentioned data correctly, the simulation of the queue system will start. The time will be displayed, as well as the queue evolution and the log of events every time something happens. When the simulation time is finished, the simulation stops.

2.2. Use cases

The use case will be presented and described below:

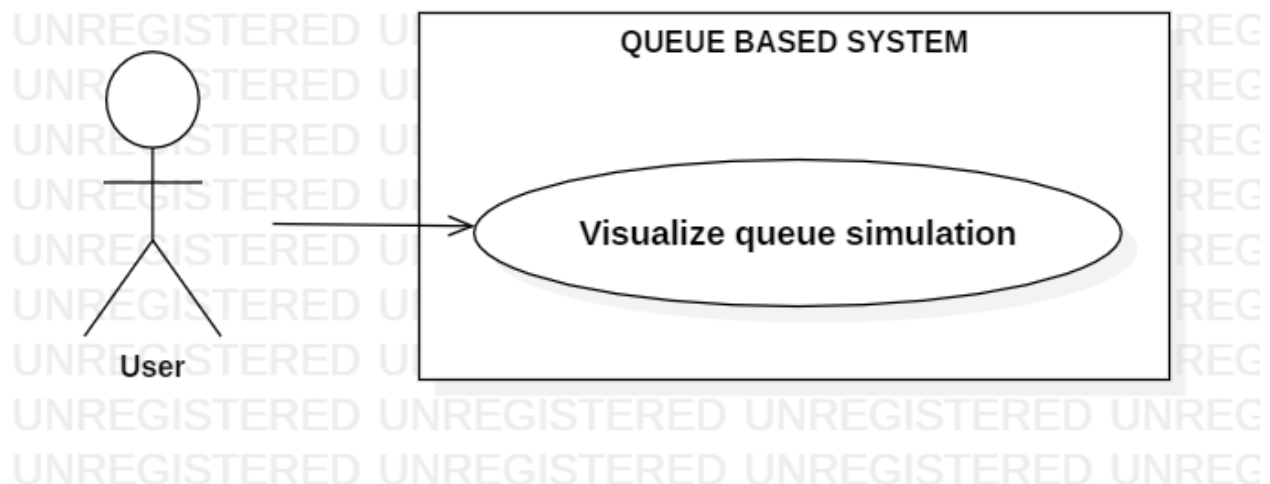


Fig. 1 Use case diagram

Use case title: Visualize queue simulation

- Summary: This use case displays the simulation of the queue based system

- Actors: User.

- Preconditions: The user introduces the setup data.

- Main success scenario:

1. The user introduces the minimum arriving interval between clients.
2. The user introduces the maximum arriving interval between clients.
3. The user introduces the minimum service time of clients.
4. The user introduces the maximum service time of clients.
5. The user introduces the number of queues.
6. The user introduces the simulation time interval.
7. Simulation starts.
8. After the simulation interval has passed, the simulation stops.

- Alternative sequences:

- a) The user introduced values smaller or equal to zero.
 1. The system tells the user that the values cannot be smaller or equal to zero.
 2. The scenario returns to specific step.
- b) The user introduced incorrect values (e.g. letters)
 1. The system tells the user that the input is incorrect.
 2. The scenario returns to specific step.

3. Design

3.1. Pattern

The design is based on the Model-View-Controller pattern. It divides the application into three interconnected parts, in order to separate the internal representations of information from the ways information is presented to and accepted from the user. The components are presented below:

3.1.1. Model

The model is the central component of the pattern. It is the application's dynamic data structure, independent of the user interface. It directly manages the data, logic and rules of the application. The model is responsible for managing the data of the application. It receives user input from the controller.

3.1.2. View

The view means presentation of the model in a particular format, Any representation of information such as a chart, diagram or table. Multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants.

3.1.3. Controller

The controller responds to the user input and performs interactions on the data model objects. It receives the input, optionally validates it and then passes the input to the model.

3.2. Classes and packages

The classes structured in packages according to the MVC pattern. Using these packages helps in organizing the classes and interfaces. The packages and classes will be presented on the next page. The classes and packages structure is illustrated in Fig. 2.

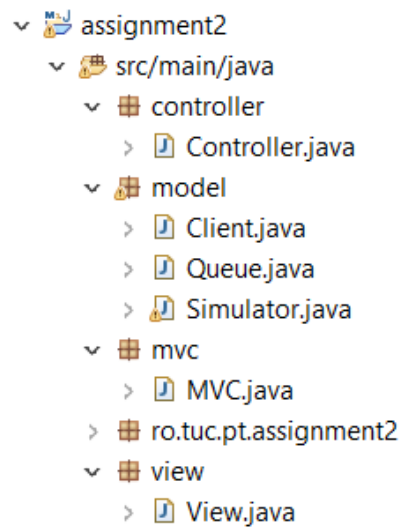


Fig. 2 Packages and classes

1. model

This is where the main classes for the simulation are implemented. This package contains the following classes:

- a) **Client:** the client entity; it is represented by a number, arriving time and service time.
- b) **Queue:** the queue entity; it is represented by a number and an *ArrayList<>* of clients. It extends the Thread class.
- c) **Simulator:** the simulator entity; it is represented by an *ArrayList<>* of queues, minimum and maximum arriving time, minimum and maximum service time, simulation interval, number of clients, total waiting and service time, current time and peak hour. It extends the Thread class. This class represents the **Model**.

2. view

This is where the Graphical User Interface is implemented, without any functionality. This package contains the **View** class, which implements the ActionListener interface. It contains frames, panels, text fields, text areas, buttons and labels.

3. controller

This package contains the Controller class, which creates a functionality to the View, by connecting it with the Model.

4. mvc

This package contains the MVC class, which interconnects the three classes, Model(Simulator) , View and Controller and offers a functionality to the whole application.

3.3. Interfaces

The interfaces that were implemented in this application are:

- *ActionListener*: implemented by the View and Controller classes. This interface is used for receiving action events. The class that is interested in processing an action event implements this interface, and the object created with that class is registered with a component, using the component's `addActionListener` method. When the action event occurs, that object's `actionPerformed` method is invoked.

3.4. Class Diagram

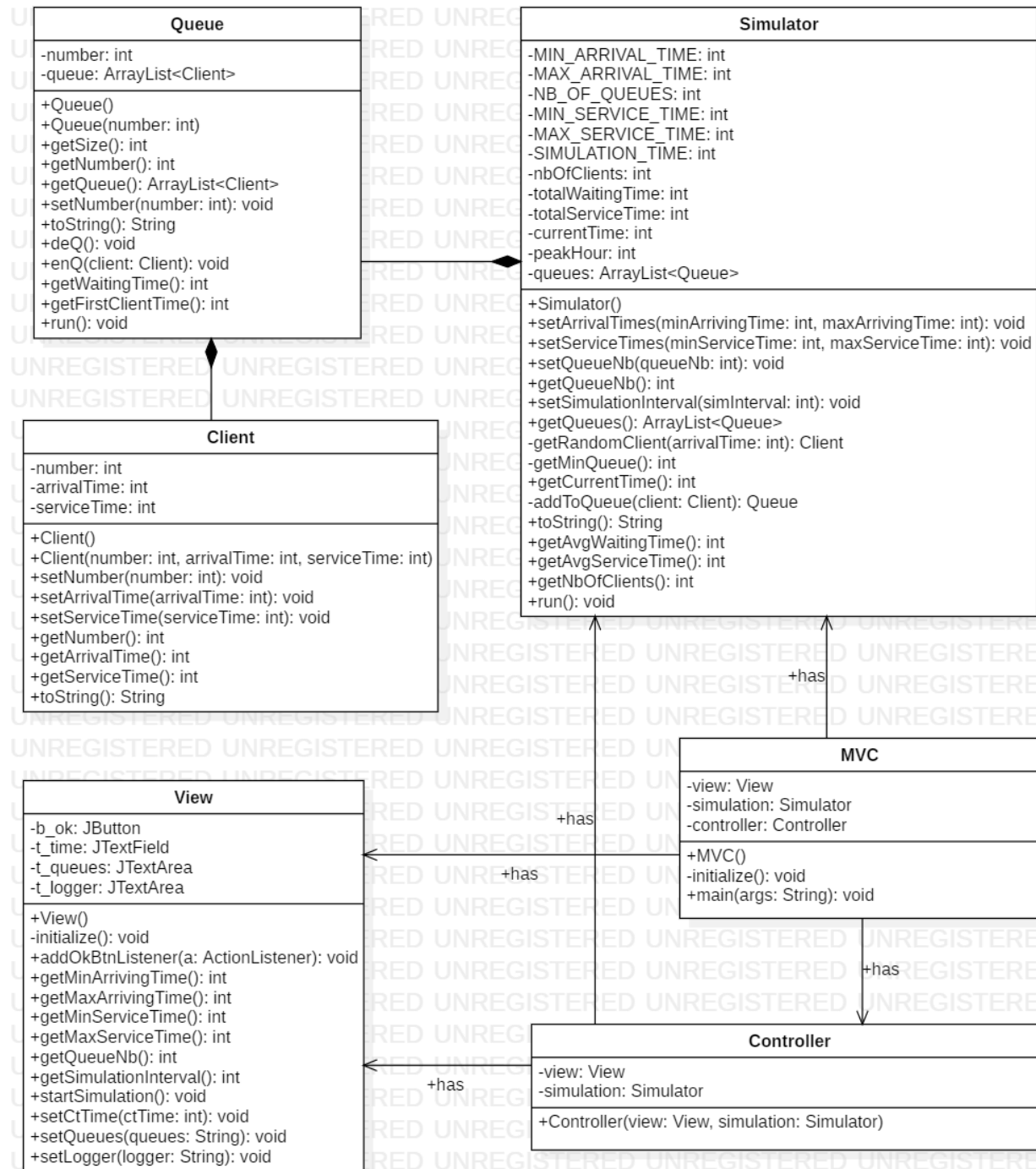


Fig. 3 Class Diagram

3.5. Relationships

- **Client-Queue:** There is a **Composition** relationship between these two classes. A Queue is composed of one or more clients, and a client cannot exist without a Queue to be part of. Client's lifespan depends on Queue's lifespan.
- **Queue-Simulator:** There is a **Composition** relationship between these two classes. The simulator cannot exist without one or more queues and a queue cannot exist without a simulator to be part of. These two classes are dependent on each other and their life span is the same.
- **Simulator-Controller:** there is an **Association** relationship between these two classes. The Controller class has as attribute an instance of the Simulator class.
- **View-Controller:** there is an **Association** relationship between these two classes. The Controller class has as attribute an instance of the View class.
- **Simulator-MVC:** there is an **Association** relationship between these two classes. The MVC class has as attribute an instance of the Simulator class.
- **View-MVC:** there is an **Association** relationship between these two classes. The MVC class has as attribute an instance of the View class.
- **Controller-MVC:** there is an **Association** relationship between these two classes. The MVC class has as attribute an instance of the Controller class.

4. Implementation

In this chapter, there will be rigorously described:

- all the classes with their attributes and important methods;
- the Graphical User Interface.

4.1. Classes

4.1.1. Client

Client class represents the client entity. It has two constructor, one parameterless and one with number, arrival time and service time as parameters. It is in a composition relationship with the Queue class, meaning that the client cannot exist if it is not part of a queue.

Attributes:

- *number*: *int* – index of the client
- *arrivalTime*: *int* – time of arrival
- *serviceTime*: *int* – time of service

Methods:

- *toString()*: *String* – returns a string of the client having its attributes (e.g. C1(1,10)).

```
public class Client {
    private int number;
    private int arrivalTime;
    private int serviceTime;
    /**
     * Client constructor with no parameters
     */
    public Client() {
        this.number = 0;
        this.arrivalTime = 0;
        this.serviceTime = 0;
    }
    /**
     * Client constructor
     * @param number order number
     * @param arrivalTime time of arrival
     * @param serviceTime time of service
     */
    public Client(int number, int arrivalTime, int serviceTime) {
        this.number = number;
        this.arrivalTime = arrivalTime;
        this.serviceTime = serviceTime;
    }
}
```

Fig. 4 Client class with attributes and constructors

4.1.2. Queue

This class represents the queue entity and extends the Thread class. It has two attributes, an index and a list of clients. Its most important method is *run()*. It is in a Composition relationship with Client class and with Simulator class, because a queue can only exist if it is a part of the simulation.

Attributes:

- *number: int* – queue's index
- *queue: ArrayList<Client>* -- list of clients

Methods:

- *toString(): String* – returns a String representing the queue's clients in order
- *deQ(): void* – removes the first client from the queue
- *enQ(client: Client): void* – adds a new client at the end of the list
- *getWaitingTime(): int* – returns the total waiting time of the clients
- *run(): void* – this is where all the action happens; the queue is put to sleep for a time equals to first client's service time.

```
public class Queue extends Thread {
    private int number;
    private ArrayList<Client> queue;
    /**
     * Queue constructor with no parameters
     */
    public Queue() {
        this.number = 0;
        this.queue = new ArrayList<Client>();
    }
    /**
     * Queue constructor
     * @param number queue's number
     */
    public Queue(int number) {
        this.number = number;
        this.queue = new ArrayList<Client>();
    }
}
```

Fig. 5 Queue class with attributes and constructors

4.1.3. Simulator

This class represents the simulator entity. This is where all the action happens, and this class represents the Model component from the Model-View-Controller design pattern. It has the following attributes and methods:

Attributes:

- *MIN_ARRIVAL_TIME*: *int* – minimum arrival time between clients
- *MAX_ARRIVAL_TIME*: *int* – maximum arrival time between clients
- *NB_OF_QUEUES*: *int* – number of queues
- *MIN_SERVICE_TIME*: *int* – minimum service time between clients
- *MAX_SERVICE_TIME*: *int* – maximum service time between clients
- *SIMULATION_TIME*: *int* – simulation interval time
- *nbOfClients*: *int* – total number of clients in the simulator
- *totalWaitingTime*: *int* – total waiting time of clients
- *totalServiceTime*: *int* – total service time of clients
- *currentTime*: *int* – current time of the simulation
- *peakHour*: *int* – peak hour of the simulation
- *queues*: *ArrayList<Queue>* -- list of queues in the simulator

Methods:

- *getRandomClient(arrivalTime: int): Client* – generates a random client
- *getMinQueue(): int* – gets the queue with the smallest waiting time
- *addToQueue(): Queue* – adds the new client to a queue and returns the queue
- *toString(): String* – returns a string with all the queues
- *run(): void* – the most important method; this is where new clients are generated and distributed to queues
- *getAvgWaitingTime(): int* – returns the average waiting time of the simulation
- *getAvgServiceTime(): int* – returns the average service time of the simulation
- *isEmptyQueue(): boolean* – returns true if one or more queues are empty.

```

public class Simulator extends Thread{
    private int MIN_ARRIVAL_TIME;
    private int MAX_ARRIVAL_TIME;
    private int NB_OF_QUEUES;
    private int MIN_SERVICE_TIME;
    private int MAX_SERVICE_TIME;
    public static int SIMULATION_TIME;
    private int nbOfClients;
    private int totalWaitingTime;
    private int totalServiceTime;
    private static int currentTime;
    private int peakHour;
    private ArrayList<Queue> queues;
    /**
     * Simulation constructor with no parameters
     */
    public Simulator() {
        this.nbOfClients = 0;
        this.totalWaitingTime = 0;
        this.totalServiceTime = 0;
        this.peakHour = 1;
        currentTime = 1;
    }
}

```

Fig. 6 Simulator class with attributes and constructor

4.2. Graphical User Interface

4.2.1. View

This class is the View component from the MVC pattern design. Here we instantiate the frames, panels, buttons, text fields and labels. The constructor of the class calls the *initialize(): void* method, which instantiates all the components.

```

public int getMinArrivingTime() {
    int arrivingTime = -1;
    try {
        arrivingTime = Integer.parseInt(t_minCT.getText());
    }
    catch(Exception ex) {
        JOptionPane.showMessageDialog(null, "Incorrect input!");
    }
    return arrivingTime;
}

```

Fig. 7 Method for View class to get and check minimum arriving time introduced by user

4.2.2. Controller

This class interconnects the Simulator and View classes and offers a functionality to the Graphical User Interface. It has as attributes a Simulator entity and a View entity and contains classes that implement the ActionListener interface. We have a class of Listeners for the button from the View class.

```
public class Controller {  
  
    private View view;  
    private Simulator simulation;  
    /**  
     * Controller constructor  
     * @param view view component  
     * @param simulation model component  
     */  
    public Controller(View view, Simulator simulation){  
        this.view = view;  
        this.simulation = simulation;  
        view.addOkBtnListener(new OkBtnListener());  
    }  
}
```

Fig. 8 Controller class with attributes and constructor

4.2.3. MVC

This is the class where we interconnect all the components from the Model-View-Controller pattern design. The application is ready for launching now.

```
public class MVC {  
  
    public View view;  
    public Simulator simulation;  
    public Controller controller;  
  
    public MVC() {  
        this.initialize();  
    }  
  
    private void initialize() {  
        View view = new View();  
        Simulator simulation = new Simulator();  
        @SuppressWarnings("unused")  
        Controller controller = new Controller(view, simulation);  
    }  
}
```

Fig. 9 Model-View-Controller class with attributes and constructor

When we run the MVC class, the Graphical User Interface will be displayed and the user will introduce the data.

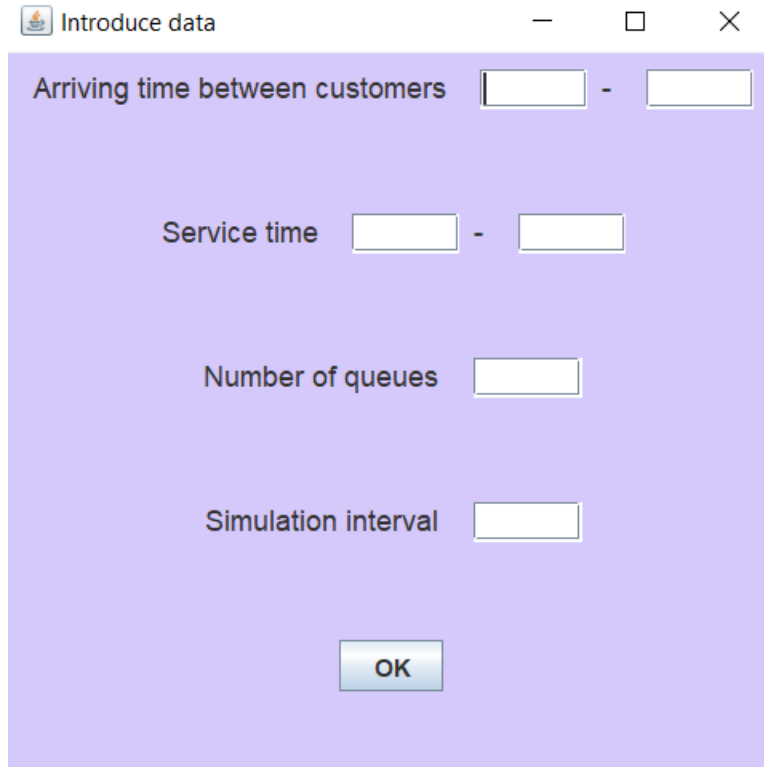
A screenshot of a graphical user interface window titled "Introduce data". The window has a light purple background and standard window controls (minimize, maximize, close) in the top right corner. It contains four input fields arranged vertically, each with a label to its left: "Arriving time between customers" followed by two empty text boxes separated by a hyphen; "Service time" followed by two empty text boxes separated by a hyphen; "Number of queues" followed by a single empty text box; and "Simulation interval" followed by a single empty text box. At the bottom center of the window is a blue "OK" button.

Fig. 10 Data set-up from the GUI

If the user doesn't introduce the input correctly (e.g. doesn't introduce anything or introduces letters) , a warning message will be displayed.

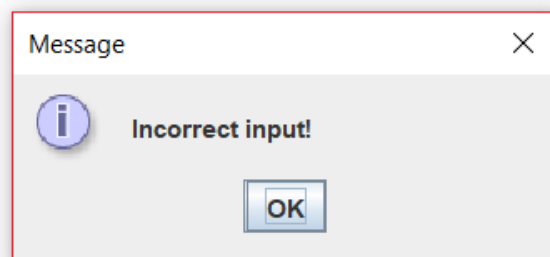


Fig. 11 Dialog message for incorrect input

If the user tries to introduce a value smaller or equal to zero, an error message will be displayed as well.

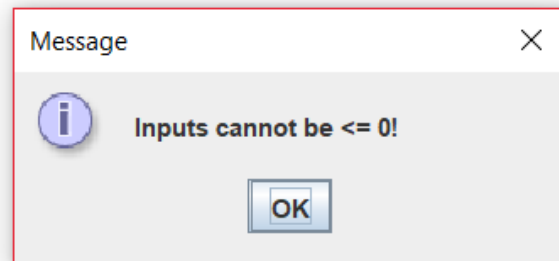


Fig. 12 Dialog message for inputs smaller or equal to zero

5. Results

If the inputs are correctly introduced, after pressing the button “OK”, the simulation will start. At the top of the frame, the current time(in seconds) will be displayed. The clients will arrive and will be efficiently placed at the queue with the minimal waiting time. The log of events(e.g. when a client arrives, when he is placed in a queue and when he leaves) will be displayed in the graphical user interface as well. When the simulation time interval passes, the average waiting time, average service time, peak hour and empty queue time will appear in the logger.

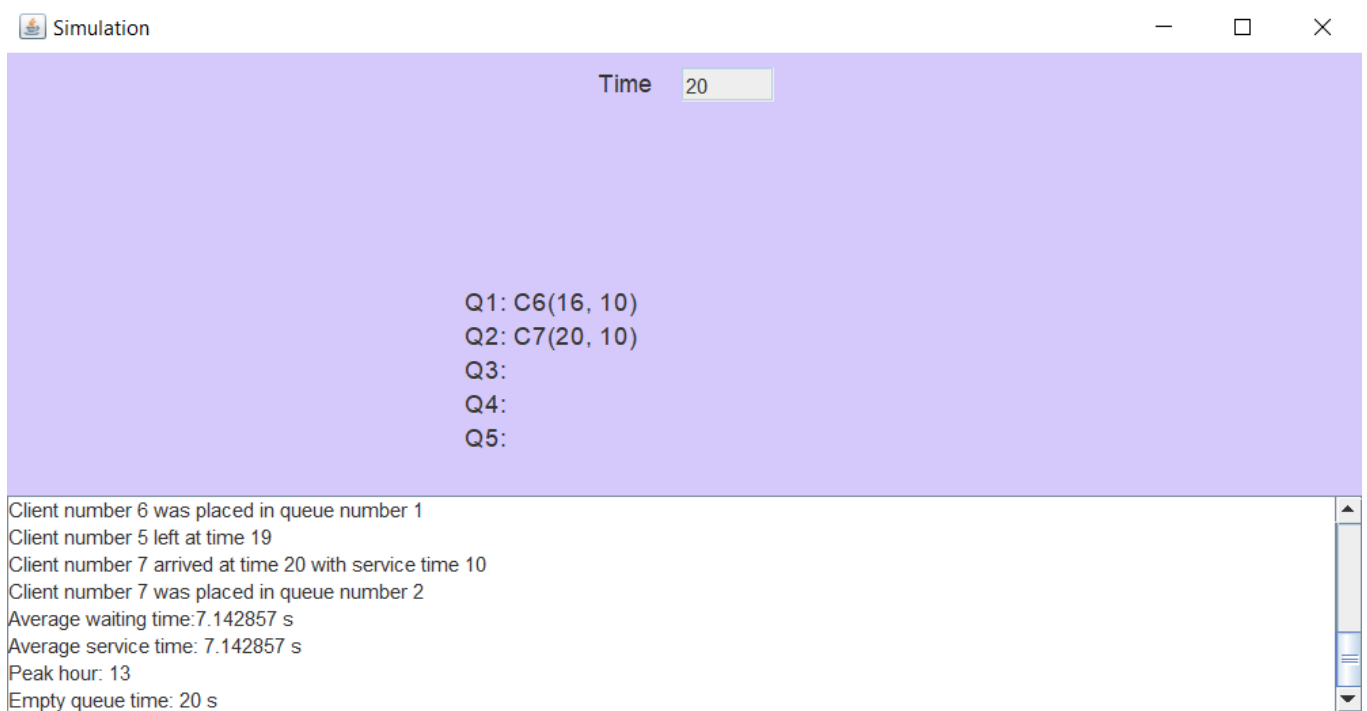


Fig. 13 Simulator frame at the end of the simulation

6. Conclusions

In conclusion, this system brings efficiency because, as they arrive, the clients are placed at the queue with the minimal waiting time. The results of the simulation (average waiting time, average service time, peak hour and empty queue time) help in building some statistics in order to help increasing the efficiency of the system in the future.

I learnt in this project more about OOP Paradigms, Javadoc comments, practiced my code writing, and learnt about the Thread class and concurrency.

This application can be improved in many ways, for example:

- when calculating the waiting time of the queue, there will be taken into consideration the remaining time of the client being currently served;
- opening and closing queues, depending on the number of clients, then distributing them in the queues evenly;
- building some “fast” queues, where clients with low service time(e.g. under 10 seconds) will be placed;
- improving the Graphical User Interface into a more complex, realistic one.

7. Bibliography

<https://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html>

<http://zetcode.com/tutorials/javaswingtutorial/>

<https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html>

<https://www.javacodegeeks.com/2013/01/java-thread-pool-example-using-executors-and-threadpoolexecutor.html>