Queues Management Application

NAME: DEAC DENISA BIANCA

CONTENTS TABLE

1. Objective

Main objective

The main objective of this assignment is to implement and design a queues management application, interested in minimizing the time amount the clients/tasks are waiting in queues/servers before they are served. The application is used to create an efficient organization for clients in queues by simulating a series of N clients, waiting to enter in one of the Q queues at a given time, being served another amount of time, and finally leaving. The application will also compute, at the end of the simulation the average waiting time, average service time and peak hour.

Sub-objectives

To create this management system, the requirements of the problem should be analyzed, identified and the design and the implementation should be done based on them. The testing of the problem should be done on several examples. In order to see them, a log of events will be displayed in the graphical user interface.

2. Problem analysis, scenarios, use cases

Problem analysis

This management system should simulate several clients entering queues, as in a real-life situation (like in a supermarket). Since there are multiple queues, they should process clients independent and simultaneously. The idea is to analyze the number of clients and the number of queues to be able to place each client in the most efficient queue at that time.

The simulation will be done through a user-friendly graphical interface in which the user will input several parameters needed to create a personalized simulation, to test different sets of data.

Scenarios and use cases

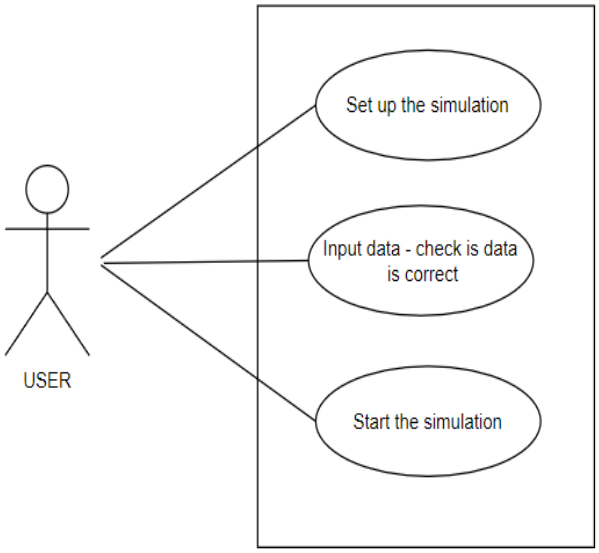
The customers are generated by giving a random service time and arrival time. The number of clients, queue and the time limits for simulation, arrival and service time depend on the input values, given by the user.

Use Case: setup simulation

Primary Actor: user

Main Success Scenario:

1. The user inserts numerical values for the: number of clients, number of queues, simulation interval, minimum and maximum arrival time, and minimum and maximum service time
2. The user clicks on the “input data” button
3. The application validates the data and since no message is displayed, the data was successfully introduced
4. The user clicks on the “Start Simulation” button
5. The log window will appear for the user to watch the simulation working



Alternative Sequence: Invalid values for the setup parameters

1. The user inserts invalid values for the application's setup parameters
2. Display a dialog error message:
   - ➔ For minimum service time greater than maximum service time
   - ➔ For minimum arrival time greater than maximum arrival time
   - ➔ For maximum arrival time greater than the maximum simulation time
   - ➔ For maximum service time greater than the maximum simulation time
   - ➔ For service + arrival time greater than maximum simulation time
3. Exit the dialog error message and returns to step 1

Functional requirements:

1. The application should allow the user to setup the simulation/ input data
2. The application should allow the user to start the simulation
3. The application should display the real-time queues evolution
4. The application should display the average waiting time, average service time and peak hour

Non-Functional requirements:

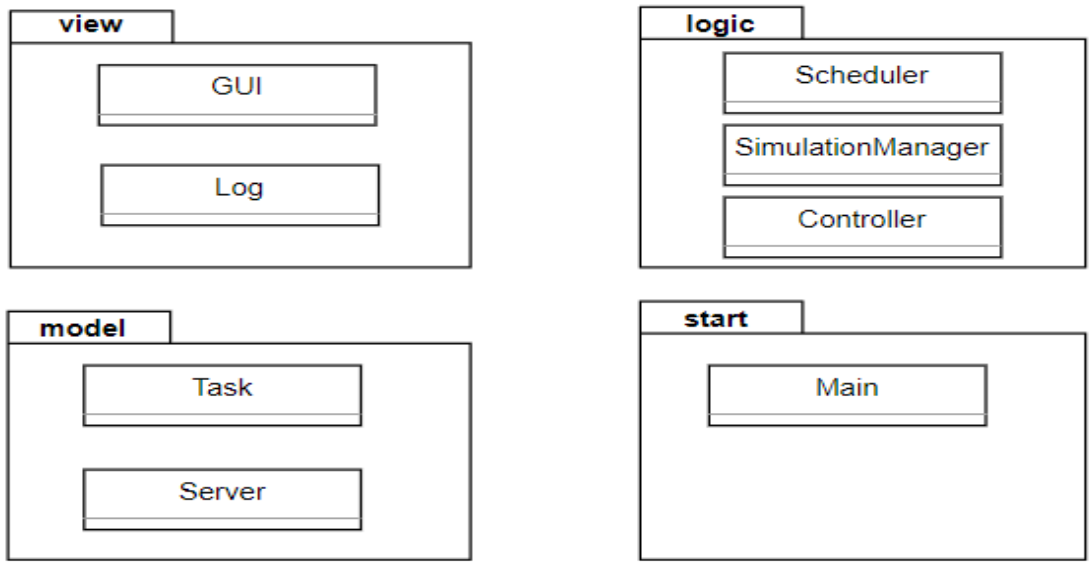
1. The simulation application should be intuitive for the user
2. The simulation application should be easy to use by the user
3. The simulation application should inform the user in case of any errors or input mistakes

4. Design

Packages

Java packages give a lot of help in organizing multiple modules and grouping together related classes and interfaces.

I chose for my application to be divided in 4 packages and, due to the way they are constructed, the MVC (model – view – controller) architectural pattern is also respected. The 4 packages proposed by me are called: start, model, logic, and view, each having different classes, as shown in the picture below.



The model-view-controller pattern proposes three main components to be used in software development, which I adapted to my project:

- *Model*, representing the underlying, logical structure of data in a software application and it does not contain any information about the user interface. It contains 2 classes that will be described below

- *View*, which represents all of the things the user can see and respond to on the screen. It contains buttons, display boxes, dialog messages and so forth. In my project, this package has 2 classes, one used for the user interface in which the parameters of the simulation will be given as a input and the second one sued to display the real-time results of the simulation.

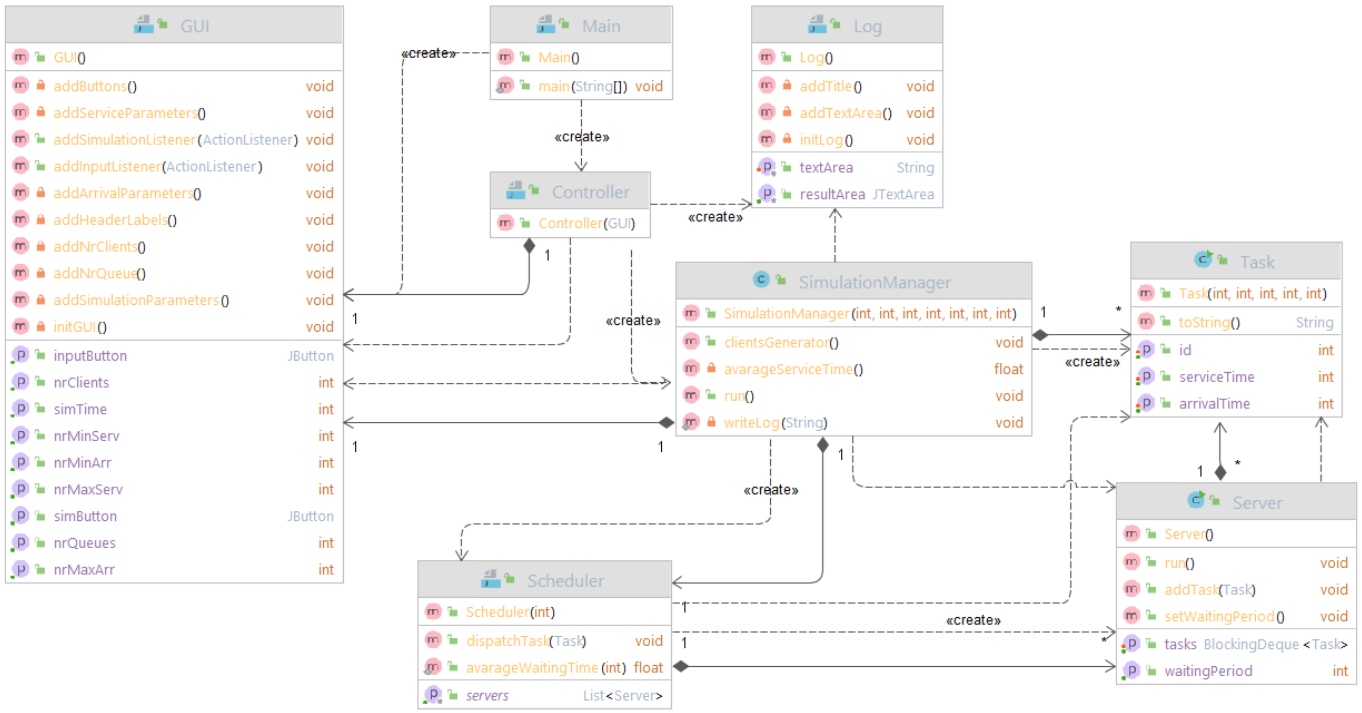
- *Controller*, represented by the "Logic" package contains the 2 classes connecting the model and the view, and is used to communicate between classes in the model and view. Even though it is called different, the package named "logic" has the role of the controller. The class "SimulationManager" takes care of the request handler and works as a link between the user and the system, while the "Controller" class links the SimulationManager and the View through the ActionListener added.

The last package, called "Start" has the class Main which will only start the execution of the program.

Classes diagram

The concept behind designing this management system is the following: the client will be represented by the "Task" and the "Server" will be the queue in which the clients must be added. The Simulation Manager is the one manipulating the data. It takes from the GUI the information about the simulation parameters given by the user. Based on them, it generates random tasks and verifies the conditions needed to be to true for the task to be added and eliminated from the queue. The distribution of the tasks in the Servers will be done by the Scheduler, in which the availability of the servers will be checked. Then, based on the waiting time for each server, the most efficient one will be chosen for each task to be added in it.

The following image illustrates the dependencies between classes and shows the methods existent in each of them.



Data Structures

One of the main data structure used is the "BlockingQueue",. Using the Unbounded Queue facilitates the code since the capacity of the blocking queue will be set to Integer.MAX_VALUE, and so, the queue will never block because it could grow to a

very large size when an element is added. Beside the BlockingQueue, I also used ArrayLists to temporarily store the tasks. It was useful because its propriety of returning any elements at a given index.

AtomicInteger is also a type of data I used in order to minimize the errors regarding threads.

5. Implementation

The management application is based on 2 main entities, the Task, which represents the "client", and the Server. The last class practically works as a Queue on which a certain number of tasks are waiting to be processed.

Task and Server

The class **"Task"** has , beside the getters, setters and the toString method that is overridden, a constructor that calculates random values for the service and arrival time as an easy way of implementing and setting the values for each client. It is used in the "Random Clients Generator" that will be explained later in the presentation, in the class named SimulationManager.

```
public class Task {
    private int id;
    private int arrivalTime;
    private int serviceTime;

    public Task(int id, int aMax, int aMin, int sMax, int sMin) {
        //the constructor automatically generates random values for arrival and service time in the given interval
        this.id = id;
        this.arrivalTime =  ( (int)Math.floor(Math.random() * (aMax - aMin + 1) + aMin) );
        this.serviceTime =( (int)Math.floor(Math.random() * (sMax - sMin + 1) + sMin) );
    }
```

The **"Server"** class takes as attributes a BlockingQueue of Tasks and the waitingTime for that server, representing the time left until the queue would be empty. The method setWaitingPeriod() doesn't receive anything as parameter because its purpose is to decrement the waiting time for each second spent by each task in the queue. In the addTask() method, when a task is added in the server, the service time of that task will be added to the waiting period of that server.

```
public class Server implements Runnable {
    private BlockingDeque<Task> tasks;
    private AtomicInteger waitingPeriod;

//constructor
    public Server() {
        this.tasks = new LinkedBlockingDeque();
        this.waitingPeriod = new AtomicInteger();
    }
```

```
//my methods
public void addTask(Task t){
    this.tasks.add(t);
    this.waitingPeriod.addAndGet(t.getServiceTime());
}
```

This class implements "Runnable" in order to use threads, one for each existing server. The threads from the servers will be started in the "Scheduler". In the run() method , for each task in the server the thread will be put to sleep until the service time has passed, since each task has to remain the queue for that period of time.

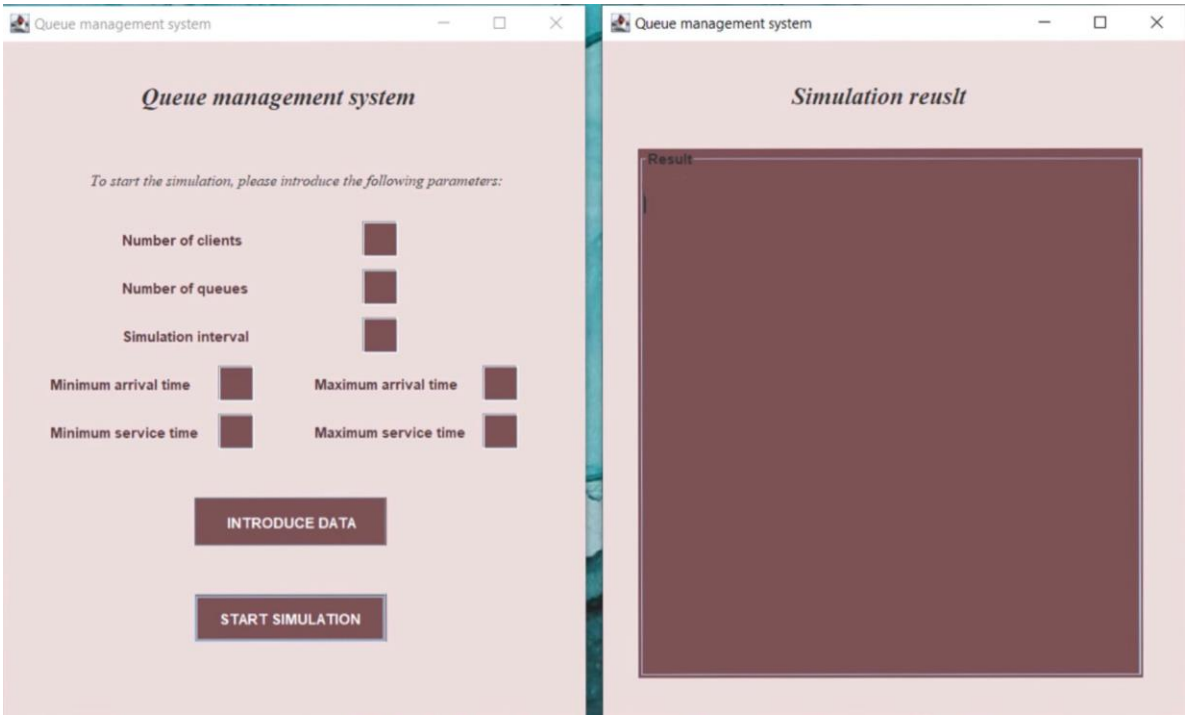
```java
@Override
public void run() {

    while(true){
        if(!tasks.isEmpty()){
            Task t  = tasks.peek();
            if(t != null){
                try {
                    Thread.sleep( millis: t.getServiceTime()*1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

Graphical Interface: GUI and Log

The graphical interface created by me is a very intuitive one, friendly, colorful and with very detailed instructions. It consists of two windows: GUI and Log.

The **"GUI"** window is the one that first opens when the user is starting the application. It asks for the parameters for the simulation: number of clients, number of queues, simulation time, minimum and maximum arrival and service time. After they are inputted, the "Input Data" button will check if the numbers are correctly given and if everything is ok, by pressing "Start Simulation" the new **Log** window, with the result starting to show will appear on its left.



Scheduler, Controller, SimulationManager

The class **"Scheduler"** is the one that creates the threads for every Server, based on the given number of queues. Its method dispatchTask() looks for the best queue in which the task should be implemented and calculates the total waiting time of the queues.

This class is used in the SimulationManager.

```
public class Scheduler {

    private  int nrQueues;
    private static List<Server> servers = new ArrayList<>();
    private static int totalWaitingTime = 0;

    public Scheduler( int nrQueues) {
        this.nrQueues = nrQueues;
        for(int i=1; i<= nrQueues; i++){
            Server s = new Server();
            Thread thread = new Thread(s);
            servers.add(s);
            thread.start();
        }
    }
```

```
public void dispatchTask( Task t){

    int minWait = Integer.MAX_VALUE;
    int pos = 0;

    for(Server s: servers){
        if(minWait > s.getWaitingPeriod()){
            pos = servers.indexOf(s);
            minWait = s.getWaitingPeriod();
        }
    }

//calculate total waiting time
// => each client has waiting time = queueWaitingTime + his service time
    totalWaitingTime = totalWaitingTime + minWait + t.getServiceTime();
     servers.get(pos).addTask(t);
}
```

The **"Controller"** is the class that "reads" the input data given in the graphical interface by the user. It adds the action listeners to the GUI and opens the new Log window in which the results will pe displayed. Is useful since it makes sure the data is first read, and only after, the main thread, the one for the SimulationManager, is created. Also here the dialog messages are added, in case the data is incorrect.

```
public Controller(GUI view){
    this.view = view;
    view.addInputListener(e->{
            if(e.getSource() == view.getInputButton()) {
                //read values from the view
                int timeLimit = view.getSimTime();
                int  nrQueues = view.getNrQueues();
                int  nrClients = view.getNrClients();
                int minArr = view.getNrMinArr();
                int  maxArr = view.getNrMaxArr();
                int  minService = view.getNrMinServ();
                int  maxService = view.getNrMaxServ();

                if (maxArr < minArr) {
                    JOptionPane.showMessageDialog( parentComponent: null,  message: "Incorrect 'Arrival Time' p
                }
                if (maxService < minService) {
                    JOptionPane.showMessageDialog( parentComponent: null,  message: "Incorrect 'Service Time' p
                }
                if (maxArr > timeLimit) {
                    JOptionPane.showMessageDialog( parentComponent: null,  message: "Incorrect parameters. Arri
                }
                if (maxService > timeLimit) {
                    JOptionPane.showMessageDialog( parentComponent: null,  message: "Incorrect parameters. Serv
                }
                //iei combobox

            }

    });
```

The most important class for the project, in my opinion, is represented by the "**SimulationManager**". This class practically manipulates the data from the view (through the Controller class) and "dictates" the work for the threads, since it also implements the "Runnable interface".

The clientGenerator() method is implemented in this class. For a given number of clients, this method creates the Task object with random values for the service time and the arrival time, in a given interval. By using the Comparator Interface, it also sorts the array of tasks by their arrival time, for an easier way of inserting them in the Queues.

```
public void clientsGenerator(){

    //generate n clients with random arrival and service time
    for( int id = 1; id <= nrClients; id++){
        Task client = new Task(id, maxArr, minArr, maxService, minService);
        clients.add(client);
    }

    //sort the clients based on their arrivalTime
    Collections.sort(clients, new Comparator<Task>(){
        @Override
        public int compare(Task o1, Task o2) {
            if(o1.getArrivalTime() < o2.getArrivalTime()){
                return -1;
            }
            else if( o1.getArrivalTime() == o2.getArrivalTime()){
                return 0;
            }
            else{
                return 1;
            }
        }
    });

}
```

In the run() method, each task is added to a certain queue by calling the dispatchTask() method existing in the Scheduler. It add the task to the most "efficient" queue, meaning the one with the smallest current waiting time, and after that, the task is eliminated from the array. Later on, by going through each queue, the service time for each first task in the queue is checked and is decremented until it has to be removed since it becomes 0.

```
for(int i = 0; i < nrClients; i ++){
    if(clients.size() > 0) {
        Task t = clients.get(i);
        if (t.getArrivalTime() == currentTime) {
            scheduler.dispatchTask(t);
            clients.remove(t);
            i--;
        }
        if(t.getArrivalTime() != currentTime){
            break;
        }
    }
}
```

```
for(int i = 0; i < nrQueues; i ++) {
    Task task = Scheduler.getServers().get(i).getTasks().peek();
    if ( task != null) {
        int t = task.getServiceTime();
        if (t == 0) {
            Scheduler.getServers().get(i).getTasks().removeFirst();
        } else
            task.setServiceTime(t - 1);
            Scheduler.getServers().get(i).setWaitingPeriod(  );
    }
```

Here, the avarage waiting time for the clients, the avarae service time and the peal hour are also computed and displayed in the console and also in the graphical user interface.

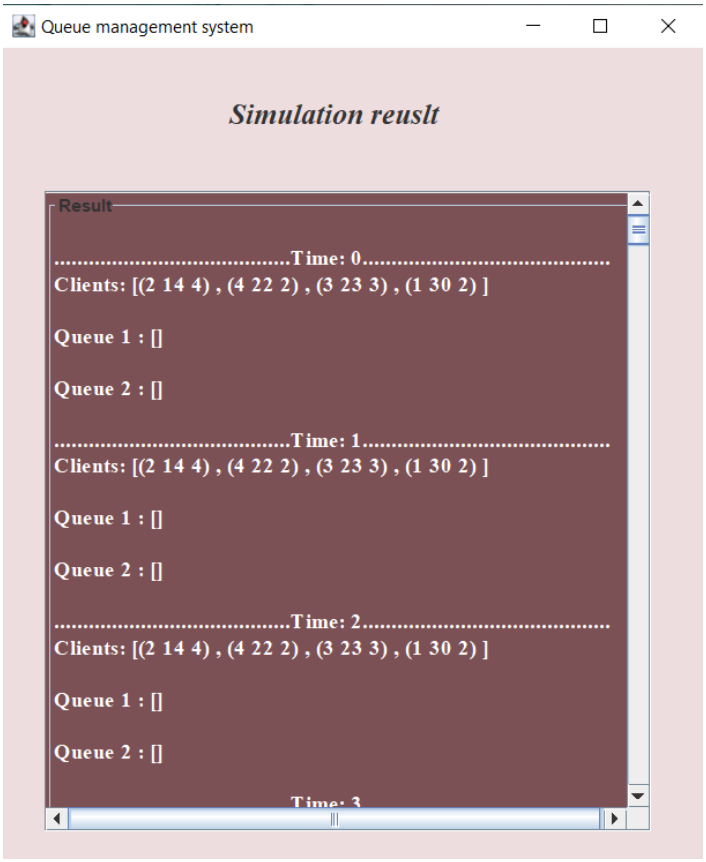
Every step of the simpulation and the content of the queues are displayed on the user interface in the window called Log. All the information is stored in a file, called output.txt.

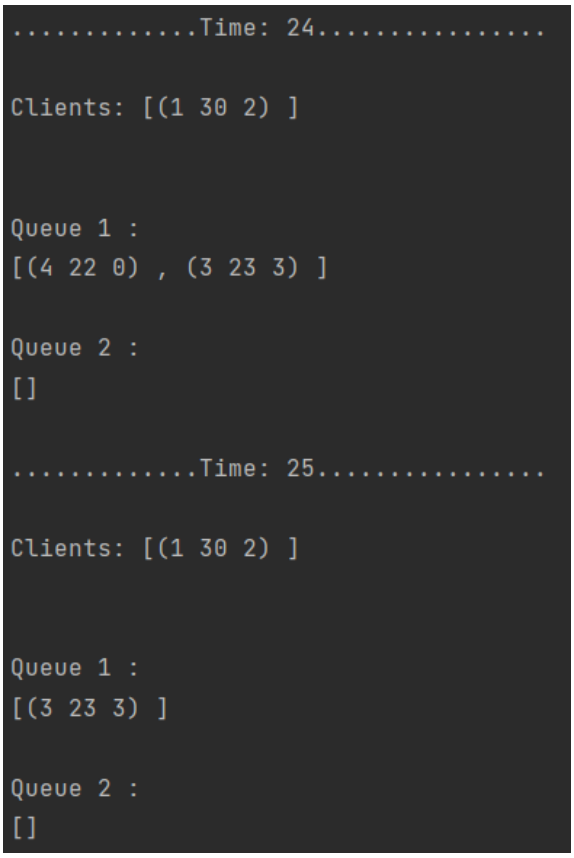
6. Results

The result of the simulation can be seen in 3 ways: in the console, in an output file and in the graphical interface.

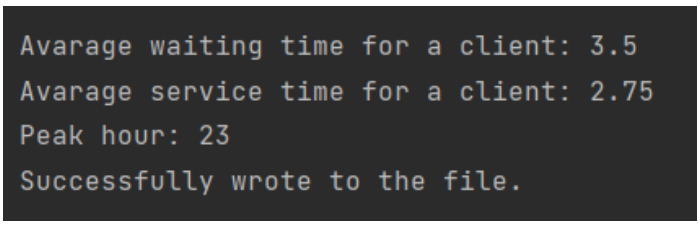
The data in the output file will appear only after the simulation is finished, while on he other two ways the log of events will be displayed in a real-time manner.

The program was tested with different sets of inputs, with a small simulation time and also a large one.





At the end of the simulation, statistics with the average waiting time, average service time and peak hour are also displayed.



7. Conclusions

Designing and implementing this application made me challenge myself, my knowledge on OOP principles and especially, my knowledge about threads. I came to understand that even though it seems hard to first model the entire application, all its classes and packages and only after to start implementing, it is extremely useful. Due to a good organization and a clear vision on everything I had to do, making the code was more cursive and logical.

The problem of queue management, being the theme of this assignment, is a very important one since a lot of real-life situations and applications are based on an efficient way of organizing clients and tasks. Beside the usefulness of this management system, the concept of simultaneously doing different things is applicable in a lot of other kind of applications. Furthermore, not only this project challenged me to learn and seek new information, but also made me understand the importance of this concept of concurrency which will be of great use for my future projects and for me as a future engineer and programmer.

8. Bibliography

- https://www.color-hex.com/color/be9d94
- https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html
- https://www.youtube.com/watch?v=lDqP5Y01ce0
- https://www.javacodegeeks.com/2013/01/java-thread-pool-example-using-executors-and-threadpoolexecutor.html
- https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicInteger.html#:~:text=An%20AtomicInteger%20is%20used%20in,deal%20with%20numerically-based%20classes.