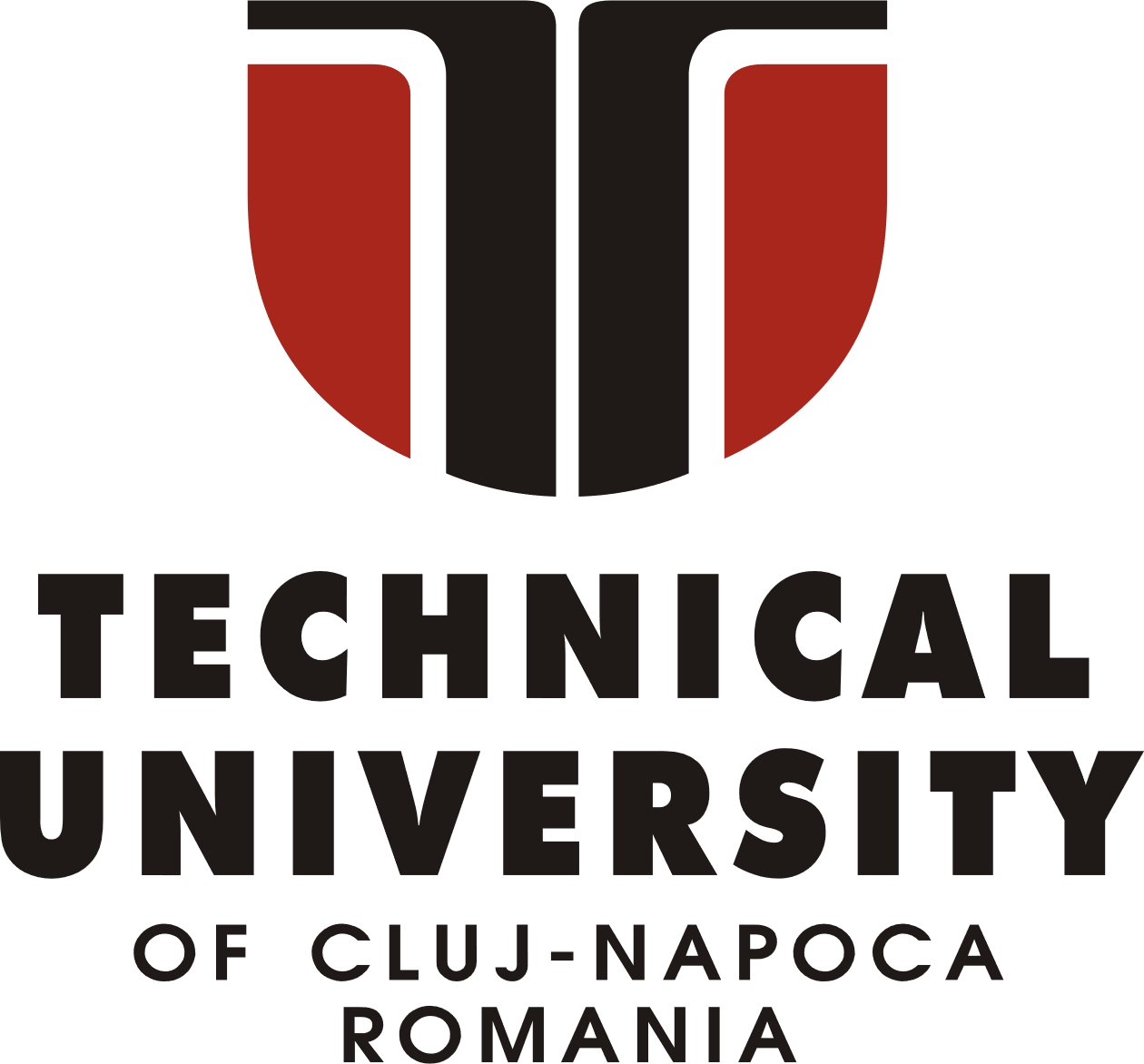
****

**\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_**

**DOCUMENTATION**

**Assignment 2. Queues Management Application**

Fundamental Programming Techniques

Student: **Radu-Augustin Vele**

Group: **30422**

Table of Contents

1. Project Objectives3

2. Problem Analysis. Modelling. Scenarios. Use cases3

3. Design4

4. Implementation5

5. Results7

6. Conclusions7

7. Bibliography8

**1. Project objectives**

The project's **main goal** is to provide the user an environment for simulating queuing-based systems. In the environment, the users need to be able to input the number of clients (or tasks), the number of queues and other information about the simulation. Afterwards, they should see the representation of the arrival of the clients to the queues, the waiting, the serving, and finally the departure from the queue. Also, the environment should provide valuable information about the average waiting time, average service time, and peak hour. This way, the users could see the impact of modification in the input conditions.

**Secondary objectives:**

* analyze the problem and identify its requirements (see Problem Analysis section)
* design the simulation application (see Design section)
* implement the simulation application following the Object-Oriented Programming development paradigm (see Implementation section)
* create a simulation environment that is not prone to errors (see Implementation section)
* use threads and synchronization mechanisms
* test the functionality and look for edge cases and limitations (see Results section)

**2. Problem Analysis. Modelling. Scenarios. Use cases.**

The problem needed to solve brings multiple functional requirements alongside non-functional requirements

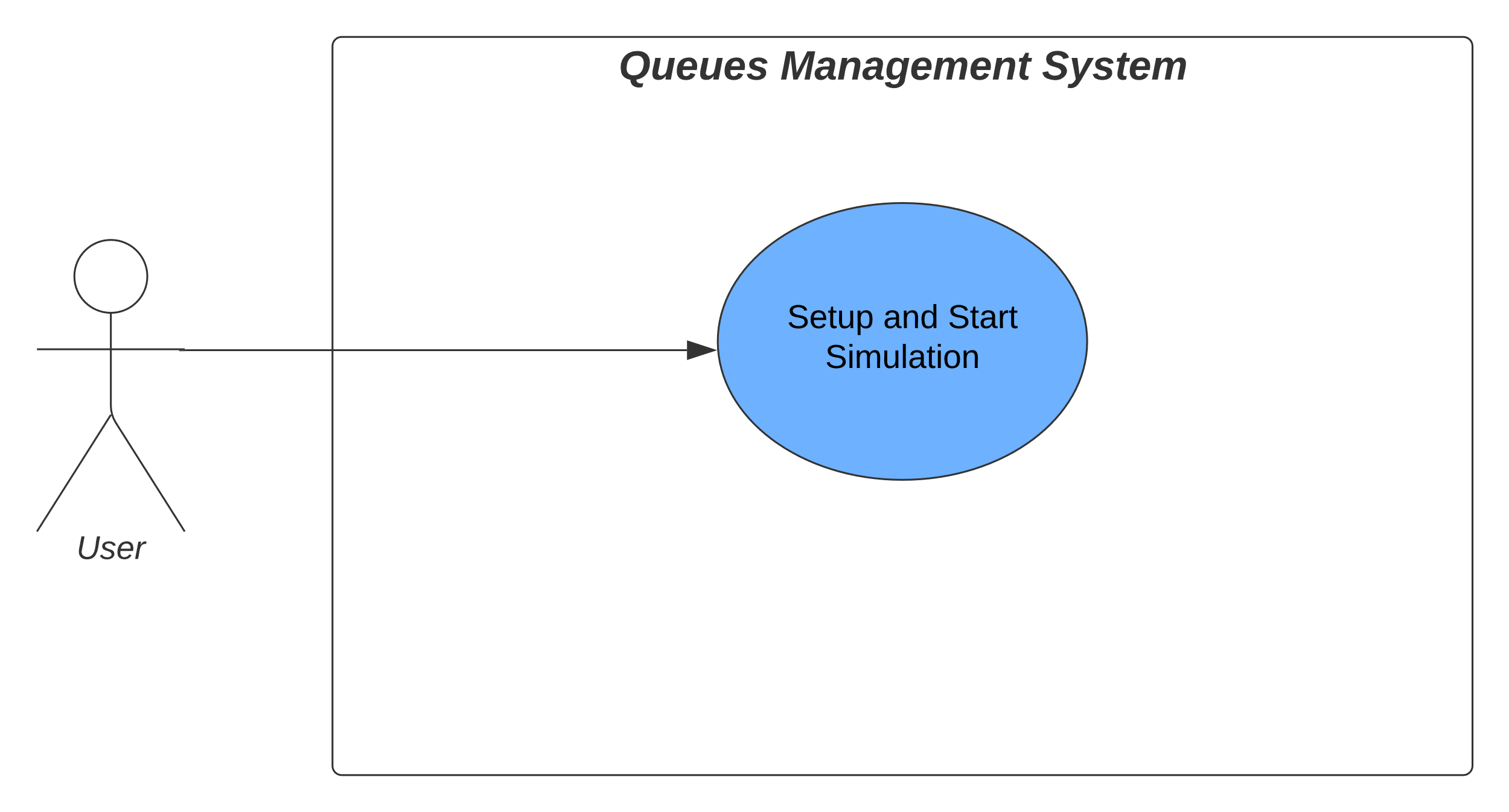
**Functional requirements:**

* the users should be allowed to set the input for the simulation.
* the simulation environment should provide a manner to start the simulation.
* the GUI should display a model of the evolution of the queues in real time.
* the simulation should incorporate a specific scheduling algorithm for deciding where the clients need to be enqueued.
* the environment informs the user about the statistical data computed during the simulation process.
* the simulation should also generate a text file containing the log of events.
* in case of errors, the users should see a pop-up window that explain the cause.

**Non-functional requirements:**

* the graphical user interface should be easy to use and intuitive.
* the users should be able to easily perform the simulation again with changed inputs.
* the output that shows the real-time evolution of the queues should be illustrative and easy to understand.

**Use cases**

The use cases of the queues management application are shown in the associated figure. The detailed steps in each use case are also described in the following paragraphs.

**Use case: Setup and Start the simulation**

Actor: user

Steps:

a. Success scenario steps:

1. the user inputs the number of clients that will wait.
2. the user inputs the number of queues available.
3. the user inputs the time of simulation (in seconds).
4. the user inputs the minimum and maximum arrival time (used for generating random clients).
5. the user inputs the minimum and maximum service time.
6. the user clicks on the "START" button.
7. the application fetches, validates the data and randomly generates clients, and starts the simulation.
8. the application displays the value of the average service time for the generated clients.
9. the evolution of the queues is displayed in the graphical user interface in real time.
10. after the end of the simulation, the application computes and prints in the simulation window the statistical data obtained (average waiting time, peak hour).
11. a text file containing the log of events is generated and made avaliable.
12. the user could input data for another simulation or close the main window.

b. Alternative sequence steps:

1. the user inputs data in all/some of the fields described above (1-5).
2. the user clicks on the "START" button
3. the simulator fetches and validates the data.
4. some of the fields are missing, or do not contain valid data (non-negative integers)
5. the simulation does not start, and an error window is displayed on the screen.
6. the user closes the pop-up window and inputs other data or closes the environment.

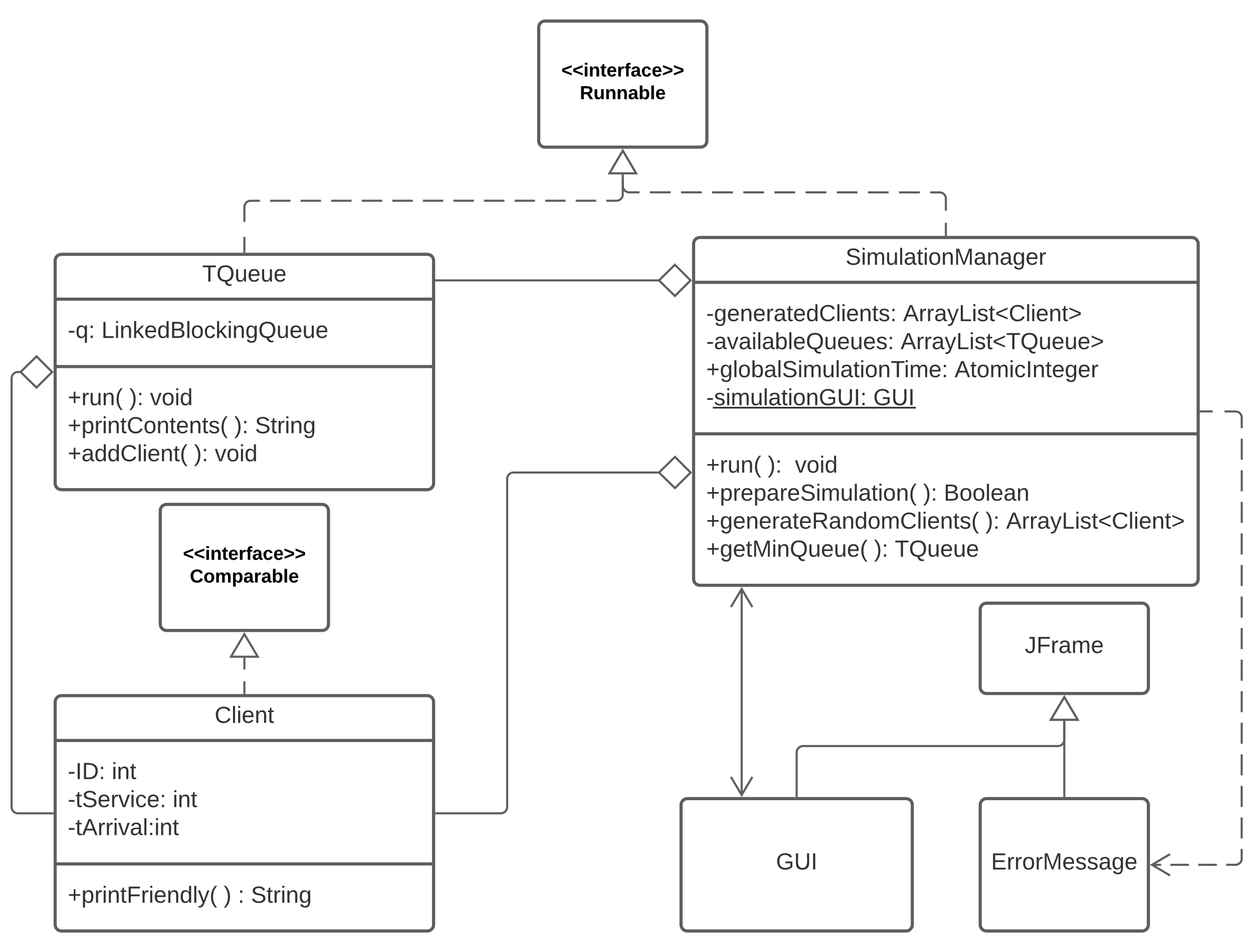
c. Alternative sequence steps:

1. the user inputs data in all the fields described above.
2. the user clicks on the "START" button.
3. the simulator fetches and validates the data.
4. random client’s generation begins, but the application checks that minimum values are smaller than maximum values, which is not the case.
5. the random generation stops, the simulation is not started, and an error window is displayed on the screen.
6. the user closes the window and inputs other data or closes the environment.

**3. Design**

**OOP Design**

The solution was designed according to the object-oriented programming paradigm without the strict usage of a specific design pattern, although there is a connection between the class SimulationManager and the class GUI that resembles the Model View Controller design pattern. The overall structure and relationships between the classes are specified in the UML diagram below. The fields of the classes are usually protected. There is, though, an important exception, the public static field: globalSimulationTime that needs to be visible from other classes as well.

****

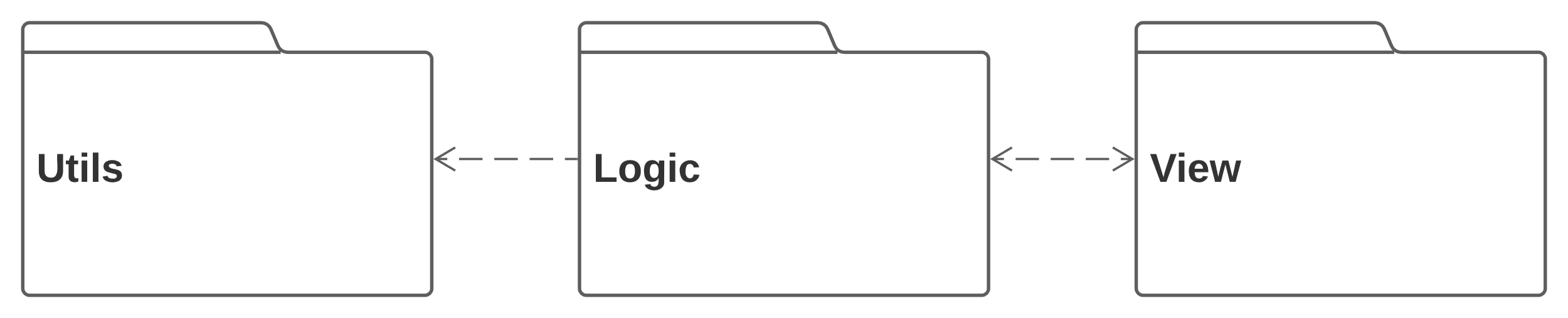
As depicted in the diagram, the classes SimulationManager and TQueue implement the interface Runnable with its run() method. That is because these classes will work with threads. Threads are needed in the application as all the queues must "move" at the same time, and the multithreading feature supported by java proves to be the right solution. Also, the Client implements the Comparable interface such that the list of waiting clients could be sorted with respect to the arrival time.

There is a strong dependency between the GUI and SimulationManager (going both sides), as the SimulationManager also adds new components to the GUI during the execution of the simulation.

Also, the simulationManager contains collections of queues and clients (**composition**).

**Package diagrams**

The package diagram depicted below shows the separation of the application's classes into three categories: Logic, Utils and View. The first package mentioned contains the SimulationManager class, and its name is well chosen as this class encapsulates most part of the logic behind the execution of the simulation. The package Utils contains the Client and TQueue classes, useful in the Logic package as they are the "building blocks" of the queue management system. Finally, the view class contains the GUI classes (GUI and ErrorScreen).



**Data Structures**

Give that the application is implemented using java concurrency there is a need for certain measures such that errors are avoided. Such errors could be the interference of threads, inconsistency of memory and contention, starting from the particularities of threads (shared resources within the process).

As solution to some of these problems thread-safe collections are used for building the queues, and atomicity is used when values of some fields need to be changed within the execution of a thread.

a. Thread-safe Collection

The chosen data structure for the queue is the LinkedBlockingQueue that implements the BlockingQueue interface. This particular kind of queue as all the queueing methods (add, remove and others) are being performed atomically. This is achieved through internal locks and other kinds of concurrency control. This is particularly useful as the enqueueing and dequeuing operations are performed in different threads, concurrently.

b. Atomicity

Atomicity determines actions that impact the content of the memory to happen "all at once". In our case AtomicInteger instances are use in different situation for actions like incrementation (getAndIncrement()) and addition(getAndAdd()).

**Algorithm**

The need for an algorithm in the queues management application comes in the case of scheduling. Although more complex procedures could be derived, the chosen strategy is "the shortest waiting time - queue strategy". Concretely, at each step when a client waiting in the queue has to be enqueued, the destination queue will be the one with the smallest waiting time (as a sum of the waiting time of the clients currently in the queue).

**4. Implementation**

This section will dig deeper into the implementation of the needed classes.

**Client**

The class consists of three fields: ID, tArrival, and tService. All of them are private and accessed through mutators. As the class implements Comparable, the method compareTo() is implemented as well determining the comparison with respect to the time of arrival. Also, a method that prints the class in a more user-friendly way is provided.

**TQueue**

This class is the model of a first in first out real-life queue and has a corresponding thread whose behavior is described in the run() method. At each step, the first element of the queue is inspected, and the thread waits for a time equal to the service time of that element, then the element is popped and additional information is saved for computing the average waiting time and the total waiting time in the queue.

The fields are the q (a LinkedBlockingQueue), the waitingPeriod, the totalWaitingTime, and totalClientsServed, alongside the Boolean simRunning that can be used to stop the thread. Beyond mutators and the run() method, a printing method and a method for adding a new client are implemented.

**Simulation Manager**

The "brain" of the application is the SimulationManager class that works similar to a controller. Its main purposes are to fetch data from the GUI, generate clients, start threads for each available queue and send clients from the waiting lists to the appropriate queue (following a scheduling procedure). Moreover, it computes the needed statistical data, and last but not least to run the main thread of the application (that works with the global simulation time and is started after the user presses the "START" value and the data is validated). Its multitude of fields is described in the UML class diagram, and this great number is justified by the fact that this class also deals with writing to file and creating new user interface components that change in real time.

The method prepareSimulation() takes the input data from the GUI and throws an exception if the data is not valid, afterwards it calls the method for generating clients, creates the queues and starts a thread for each queue and sets-up the "dynamic" part of the user interface.

The method run() (initially defined in the Runnable interface) deals with sending the waiting clients to queues, updating the user interface, generating the file, and finally computing the statistical data.

The method generateRandomClients() (that uses an instance of the Random class) and getMinQueue() are self-explanatory through their names.

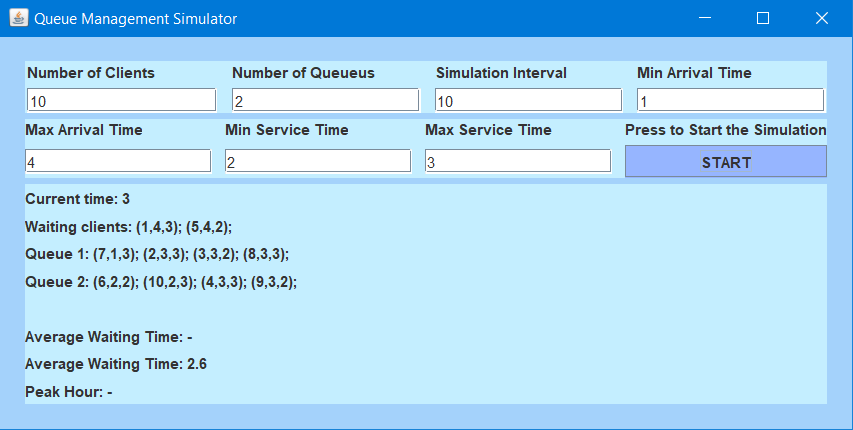
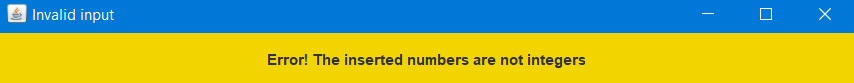
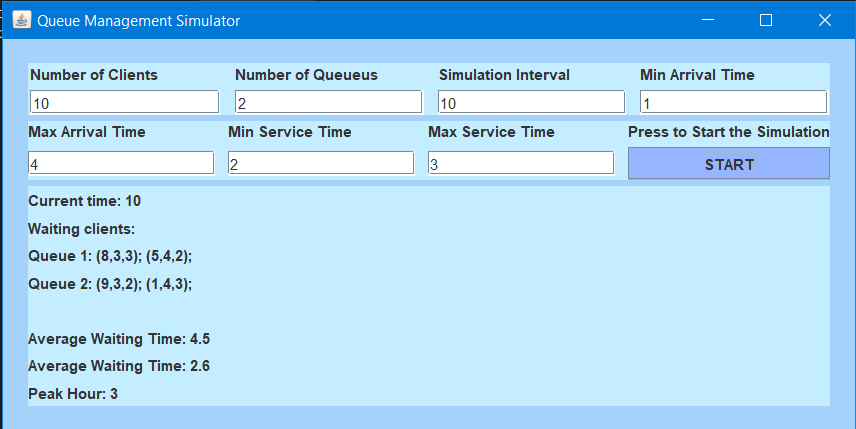
**Graphical User Interface**

The main element of the graphical user interface is the class called GUI that is essentially the simulation environment. The upper panel contains the input area where the user could add the desired specifications for the simulation and initiate it. The lower panel is dealt with by the SimulationManager and has a grid layout. Its content, represented by the current time, content of each queue, and statistical data, is known at runtime and updated once every 1000 milliseconds (as a result of the Thread.sleep(1000) call).

The GUI creates an instance of SimulationManager in its constructors, and the implementation of the action listener for the button START determines a call to the prepareSimulation(). The simulation is started only if the method returns true.

Graphical user interface, application

Description automatically generatedThe other element of the user interface is the ErrorScreen class that pops up when the input is not valid.

****

**5. Results.**

To test the correctness of the queues management system application, several tests have been provided. The input data sets are presented below.

|  |
| --- |
| Test 1 |
| N = 4  Q = 2  𝑡𝑠𝑖𝑚𝑢𝑙𝑎𝑡𝑖𝑜𝑛𝑀𝐴𝑋 = 60 seconds  [𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙𝑀𝐼𝑁, 𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙𝑀𝐴𝑋] = [2, 30] [𝑡𝑠𝑒𝑟𝑣𝑖𝑐𝑒𝑀𝐼𝑁, 𝑡𝑠𝑒𝑟𝑣𝑖𝑐𝑒𝑀𝐴𝑋]= [2, 4] |
| Test 2 |
| N = 50  Q = 5  𝑡𝑠𝑖𝑚𝑢𝑙𝑎𝑡𝑖𝑜𝑛𝑀𝐴𝑋 = 60 seconds  [𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙𝑀𝐼𝑁, 𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙𝑀𝐴𝑋] = [2, 40] [𝑡𝑠𝑒𝑟𝑣𝑖𝑐𝑒𝑀𝐼𝑁, 𝑡𝑠𝑒𝑟𝑣𝑖𝑐𝑒𝑀𝐴𝑋]= [1, 7] |
| Test 3 |
| N = 1000  Q = 20  𝑡𝑠𝑖𝑚𝑢𝑙𝑎𝑡𝑖𝑜𝑛𝑀𝐴𝑋 = 200 seconds  [𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙𝑀𝐼𝑁, 𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙𝑀𝐴𝑋] = [10, 100] [𝑡𝑠𝑒𝑟𝑣𝑖𝑐𝑒𝑀𝐼𝑁, 𝑡𝑠𝑒𝑟𝑣𝑖𝑐𝑒𝑀𝐴𝑋]= [3, 9] |

The log of events from the simulations are uploaded in the project repository.

**6. Conclusions. Takeaways. Further Development**

The queues management system brings a solution for simulating various real-life situations that involve the management of many tasks/people through a fixed number of "channels". Especially, the data provided could be useful to estimate the necessary number of queues for the desired number of clients. The structure of the application could be improved using a different scheduling technique, maybe in a different class. Also, another important improved aspect could be the way of displaying the contents of the queue (using icons, more sophisticated labels, or other components).

**7. Bibliography**

1. Distributed systems research laboratories - TUCN, resources.

2.Oracle Linked Blocking Queue documentation

3. Oracle Random documentation

4. Oracle Runnable documentation

5. Oracle Concurrency documentation

6. Stack Overflow