Queues Simulator

Miu Daria

Assignment Objectives

Main objective

The main objective of this project is to design and implement a simulation application aiming to analyze queuing-based systems for determining and minimizing clients' waiting time.

Sub-objectives

- Analyze the problem and identify the functional and non-functional requirements
- Create the design of the Queues Simulator
- Implement the Queues Simulator
- Test the Queues Simulator in multiple scenarios

Problem Analysis, Modeling, Scenarios, Use-cases

Problem Analysis

Queues are present in various real-world circumstances. The queue management system is a set of tools and sub-systems that assist in controlling customers flow, managing the waiting time, and enhancing the customer experience for multiple industries.
The main purpose of a queue is to minimize the time amount their "clients" are waiting in queues before they are served. One way to minimize the waiting time is to add more servers, but this approach increases the costs of the service supplier.
This Queues Simulator brings a solution to this problem. The application makes use of a system that places the newly arrived clients in the queue where the waiting time should be minimum at that moment. This behavior of the application is simulated in the following manner: a number of N clients are waiting for service, each of them is entering one of the Q queues, they are being served, and finally they are leaving the queues. All clients are characterized by 3 values (an id, the arrival time, and the time it takes to serve that client). These values are randomly generated when the simulation is started. The selection policy for choosing in which queue to insert the next client is to select the queue with the minimum waiting time.

Modeling

To reach its purpose, such an application should meet certain requirements like:

Functional requirements:

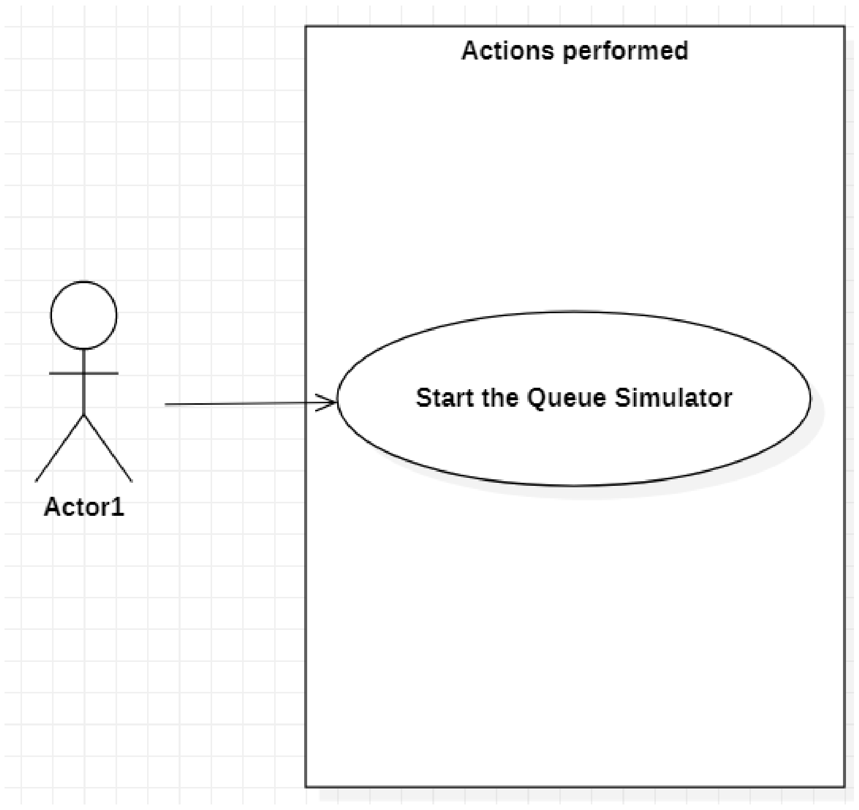
- The simulator should allow the inserting of the following parameters: number of clients, number of queues, the time to simulate, the intervals in which the arrival time and service time of the clients should be situated.
- The simulator should be started by the user, with the press of a button
- The simulator should be efficient and place each client in the minimum waiting queue
- The simulator should display the flow of events happening in the simulator in real-time in the graphical user interface.

Non-functional requirements:

- The simulator should be intuitive and easy to use.

Use-cases and Scenarios

Use Case Diagram



Use Case Scenarios

Successful scenario
Step 1: The user inserts the needed parameters for the simulation: number of clients, number of queues, the time to simulate, the intervals in which the arrival time and service time of the clients should be situated.
Step 2: The user presses the “START” button.
Step 3: The application checks if the input is valid.
Step 4: The application launches the simulation.
Step 5: The logs are displayed in real time.

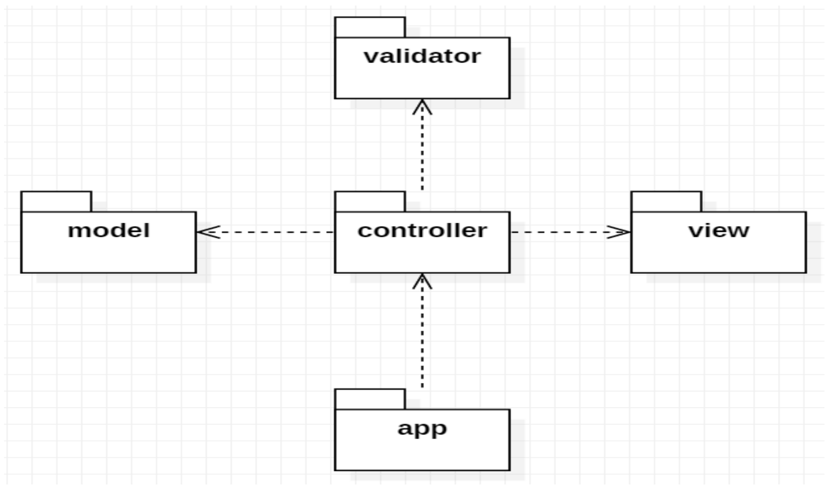
Unsuccessful scenario
Step 1: The user inserts the needed parameters for the simulation: number of clients, number of queues, the time to simulate, the intervals in which the arrival time and service time of the clients should be situated.
Step 2: The user presses the “START” button.
Step 3: The application checks if the input is valid.
Step 4: The input is not valid, and a message is shown to the user.

Design

The application was developed according to the Model-View-Controller pattern. The model is independent of the user interface, it manages the logic and the background of the application. The view represents the part that generates the graphical user interface. The controller creates the connection between model and view, it takes inputs and converts them into commands for the model or view.
To clearly represent this pattern, the application is divided in multiple packages.

Packages

Package Diagram



The Model Package: contains the **Client**, **Queue**, **QueueManager** classes, the fundamental objects of the app;

The Controller Package: contains the class **Controller**, which intermediates the relationship between view and model;

The View Package: contains the class **UserInterface**, where the entire aspect of user interface is created;

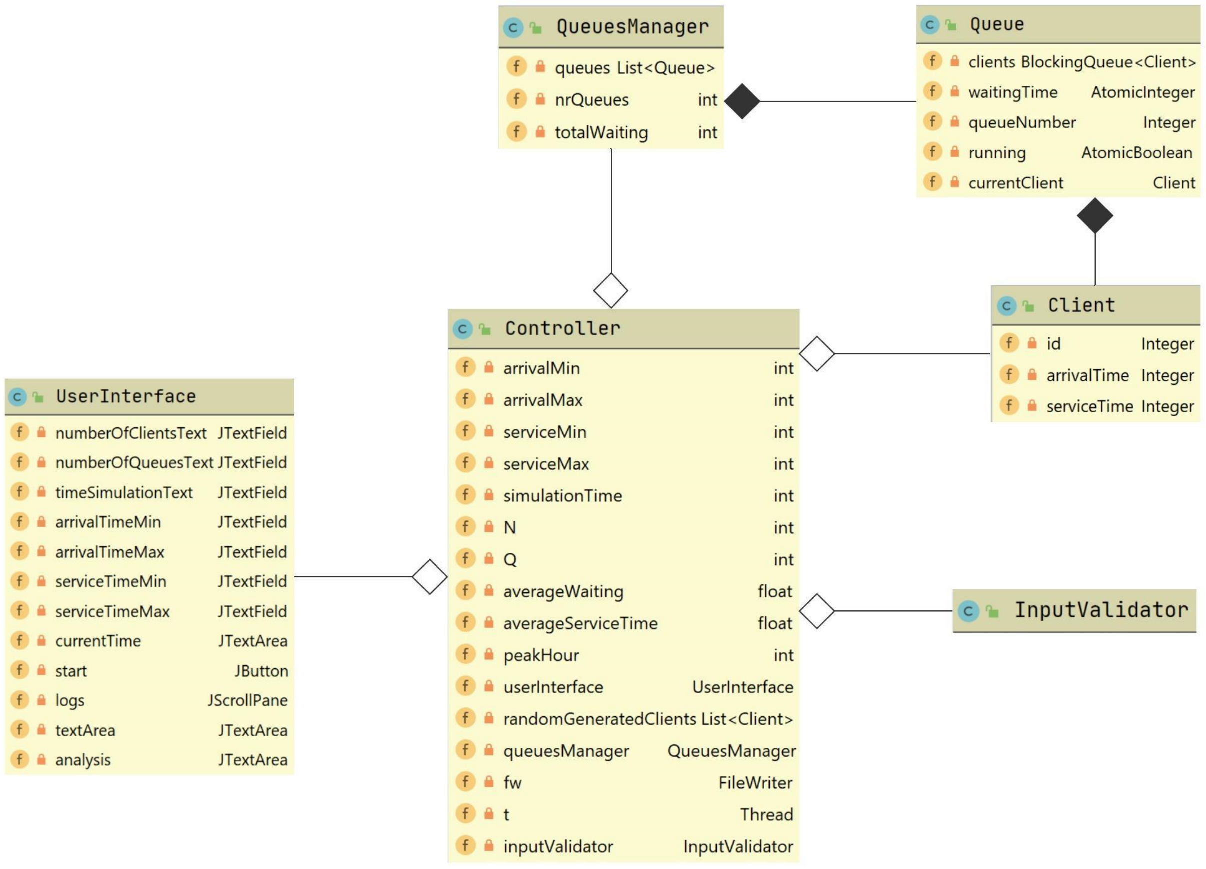
The Validator Package: contains one class **InputValidator**, used in the Controller to check if the data introduced by user is valid to be used in the application;

The App Package: contains only the **Main** class, which is run to start the application.

Classes

Class Diagram

The relationships between classes are represented in the following diagram.



Class design

Client Class

The class is a simplified representation of a real client, which gets at the check-out process after a certain time(arrivalTime attribute) and has a specific serviceTime based on his/hers needs.

Queue Class

Is an entity that imitates a real life queue situation where clients wait in line to receive a service. Clients are queued, served and dequeued.

QueueManager Class

The Class QueueManager has its main purpose to keep track of the queue system. It will be used to perform operations on the queues, such as opening and closing the queues or dispatching clients to queues and to get information about data in the queues, like what is the minimum waiting queue.

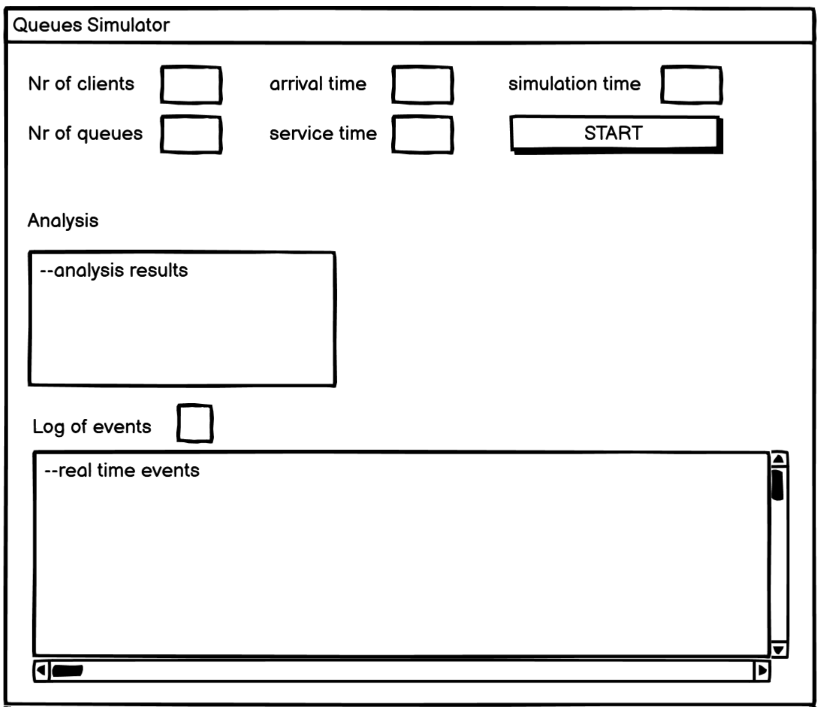
Controller

Represents the core of the application, intermediating the relation between the objects of the application and the view. It takes the main simulation manager role, having as some of its attributes the parameters of the simulation (simulation time, number of clients, number of queues).

Graphical User Interface - design

The graphical user interface should be intuitive and easy to understand and to use. The idea of the design is the following. The interface will contain text fields in which the user will introduce the parameters for the simulation, a button to start the simulation, 2 text areas to print the results of the analysis and the real time log and a text area in which the current time will be displayed (like a clock). The text field for the log of events should contain a scroll bar, since the number of queues and clients is unknown at the beginning, the size of the output will vary.

Mockup of the GUI

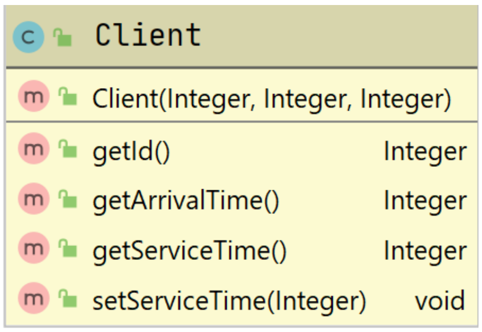


Data Structures

Data Structure represents a way in which the data can be stored and organized so that it can be used efficiently and easily. The data structures used in this project are BlockingQueue and List.
The BlockingQueue data structure is similar to a normal Queue but it has some additional features: it supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element.
The choice of designing with Blocking Queues was made based on the fact that BlockingQueues are thread-safe, since all queuing methods achieve their effects atomically.
The choice for the List data structure, more precisely, ArrayList, was made based on the fact that it comes with multiple helpful features and methods that make the process of writing code simpler and faster. Specific methods such as add, remove or isEmpty were used. For iterating through lists the iterator and list iterator were used and also the foreach instead of the simple for.

Implementation

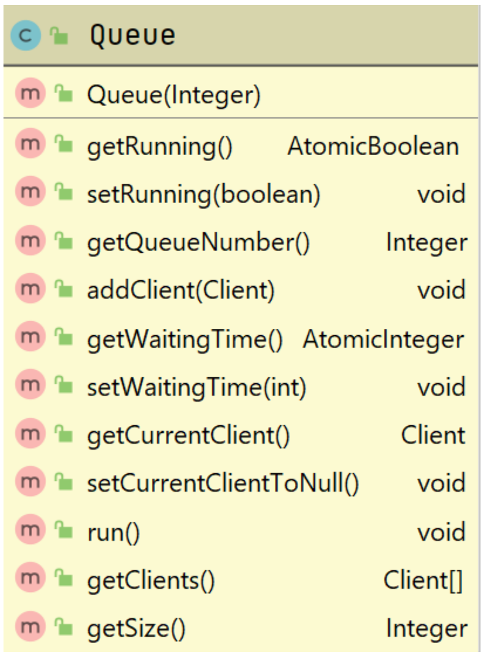
Client Class



Client	
Client(Integer, Integer, Integer)	
getId()	Integer
getArrivalTime()	Integer
getServiceTime()	Integer
setServiceTime(Integer)	void

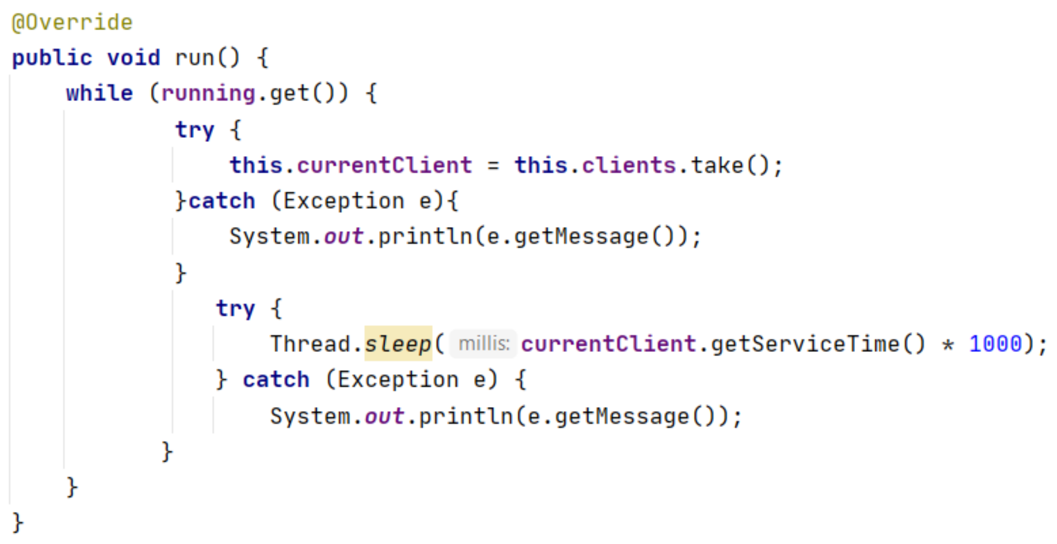
The attributes of the class are : id, arrivalTime, serviceTime. The only methods the class contains are : constructor with parameters for its attributes, getters and setters for some of its attributes.

Queue Class



Queue	
Queue(Integer)	
getRunning()	AtomicBoolean
setRunning(boolean)	void
getQueueNumber()	Integer
addClient(Client)	void
getWaitingTime()	AtomicInteger
setWaitingTime(int)	void
getCurrentClient()	Client
setCurrentClientToNull()	void
run()	void
getClients()	Client[]
getSize()	Integer

The Queue Class represents a thread that implements the Runnable interface and overrides the *run()* method that will be executed during its lifecycle. The *run()* method is presented below. While the queue is not stopped, the first client is dequeued, and the thread execution is suspended for a client's serviceTime period. The method *take()* is a great choice for dequeuing because it waits for an element to be available if the queue is empty.



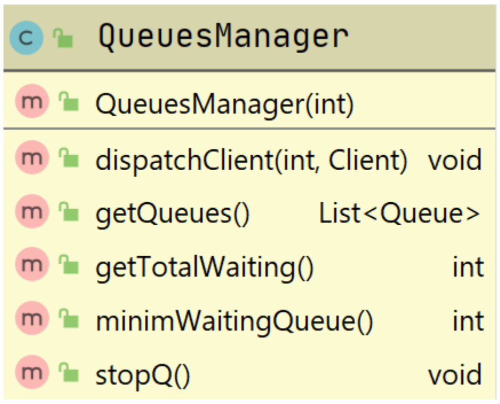
```
@Override
public void run() {
    while (running.get()) {
            try {
                this.currentClient = this.clients.take();
            }catch (Exception e){
                System.out.println(e.getMessage());
            }
               try {
                   Thread.sleep( millis: currentClient.getServiceTime() * 1000);
               } catch (Exception e) {
                   System.out.println(e.getMessage());
               }
           }
    }
}
```

Some important attributes of this class are the BlockingQueue of clients, the Atomic Integer waiting time and the Atomic Boolean running, that are specific for working with threads. The currentClient attribute represents the client that is being served at that certain time.

An important method is the *addClient()* method, which is used to add clients in the queue, and also to increase the waiting time of the queue with the service time of the added client.

The other methods are getters and setters for the attributes. The class also has a constructor with 1 parameter, the number of the queue.

QueueManager Class



QueuesManager	
QueuesManager(int)	
dispatchClient(int, Client)	void
getQueues()	List<Queue>
getTotalWaiting()	int
minimWaitingQueue()	int
stopQ()	void

The class has important attributes such as an ArrayList of Queues, the number of queues, and the totalWaiting time.

The method *dispatchClient()* does 2 important jobs, it adds a client to the queue and it also increments the totalWaitingTime. The implementation of the method can be seen below.

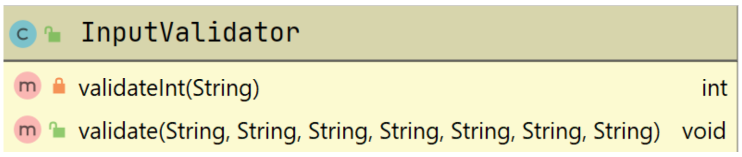
```java
public void dispatchClient(int i, Client client){
    this.totalWaiting += queues.get(i).getWaitingTime().intValue();
    queues.get(i).addClient(client);
}
```

The total waiting time sums the period of times all the clients have to wait, and will be used later to get the average waiting time.

Another relevant method is *getMinimumWaitingQueue()* which practically represents the main strategy on which the simulation is based on, more precisely, finding the best queue to place a client in, taking into consideration the waiting time. It is implemented as a simple minimum determination on the waiting times of each queue.

The method *stopQ()* stops all the queues by setting their running value to false.

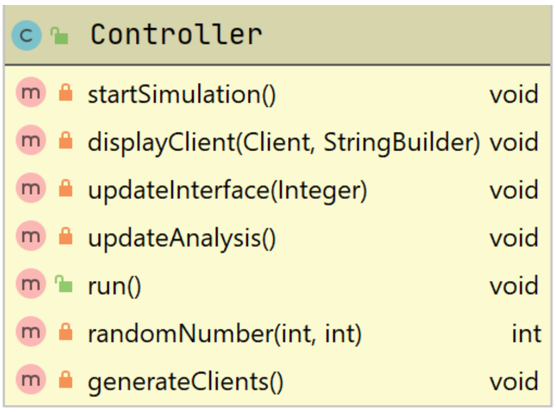
InputValidator Class



InputValidator	
validateInt(String)	int
validate(String, String, String, String, String, String, String)	void

Contains 2 methods used for validation of the input.
These methods check if the input is valid: the input should have no empty fields; no other characters than integer numbers are introduced; check conditions for the intervals and the simulation time to be valid, throw exceptions if something went wrong.

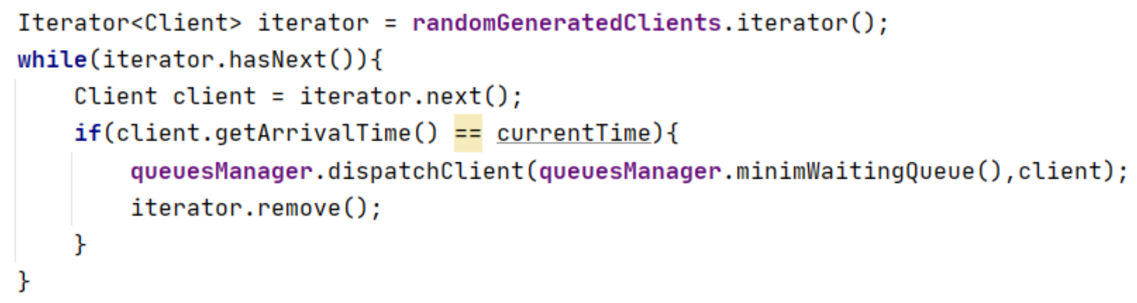
Controller Class



Controller	
startSimulation()	void
displayClient(Client, StringBuilder)	void
updateInterface(Integer)	void
updateAnalysis()	void
run()	void
randomNumber(int, int)	int
generateClients()	void

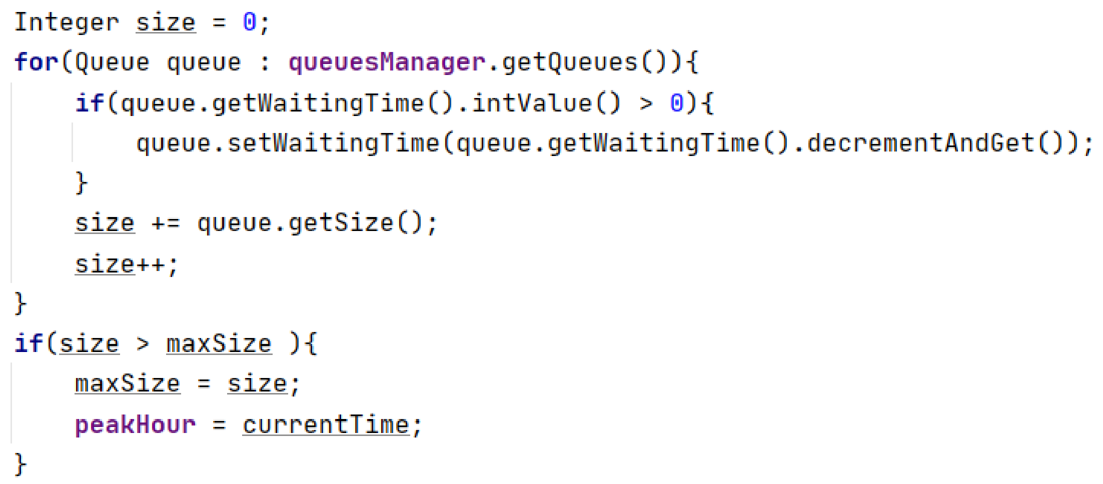
The Controller Class has some important attributes : number of clients, number of queues, simulation time, bounds of arrival time, bounds of service time; an instance of the UserInterface; an instance of QueueManager; a FileWriter(for writing the log); an InputValidator; a List of the random generated Clients.

TheController Class represents a thread, the main thread of the application that implements the Runnable interface and overrides the *run()* method that will be executed during its lifecycle.
Here are the main steps implemented in the *run()* method.
Adding the clients to the queues when their arrivalTime = currentTime



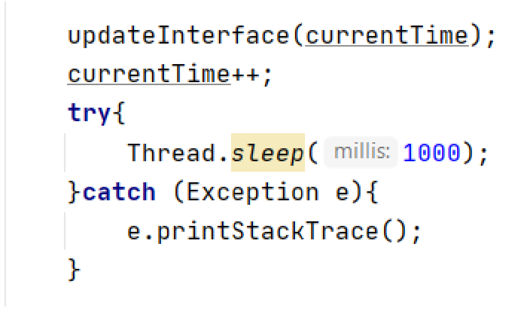
```
Iterator<Client> iterator = randomGeneratedClients.iterator();
while(iterator.hasNext()){
    Client client = iterator.next();
    if(client.getArrivalTime() == currentTime){
        queuesManager.dispatchClient(queuesManager.minimWaitingQueue(),client);
        iterator.remove();
    }
}
```

Iterating the list of queues and decrementing their waiting time, but also making use of the iteration to get an insight about the peak hour(by counting the number of clients at the certain time)



```
Integer size = 0;
for(Queue queue : queuesManager.getQueues()){
    if(queue.getWaitingTime().intValue() > 0){
        queue.setWaitingTime(queue.getWaitingTime().decrementAndGet());
    }
    size += queue.getSize();
    size++;
}
if(size > maxSize ){
    maxSize = size;
    peakHour = currentTime;
}
```

Updating the interface and waiting for 1 second



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

The final step from *run()* is to stop the queues, to calculate the average waiting time, and to update the part in the interface in which the final analysis is displayed. The formula for the average waiting time is : Sum of all waiting times of clients/the number of clients.

```
queuesManager.stopQ();
averageWaiting = (float) queuesManager.getTotalWaiting() / N;
updateAnalysis();
```

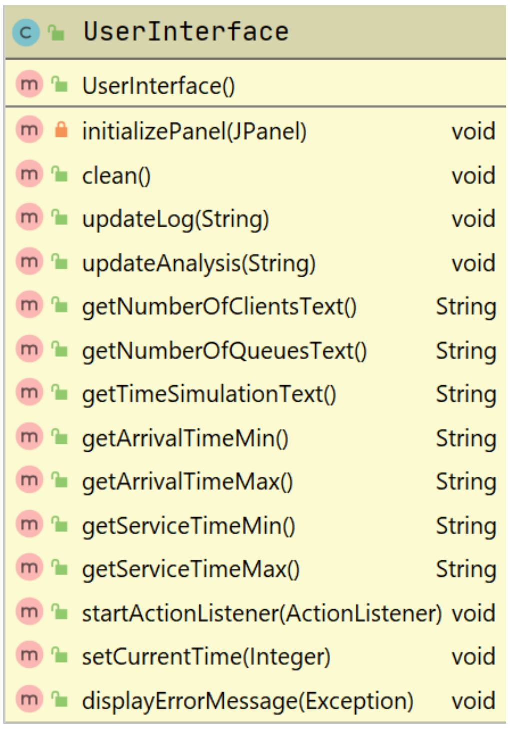
The updateInterface and updateAnalysis methods are based on the idea of building a string using StringBuilder then writing the string in the interface and appending the string to the log in the text file.

The method *randomNumber()* is used to randomly generate the attributes of the Clients in the randomGeneratedClientsList. The idea is that it generates a random number from 0 to max-min then adds min to bring the number in the wanted interval.

```
private int randomNumber(int min, int max) {
    Random random = new Random();
    return random.nextInt( bound: max - min) + min;
}
```

The *startSimulation()* method activates the action listener for the START button. The file is open for writing, the input is taken from the interface and validated.Then the main thread is started.

UserInterface Class

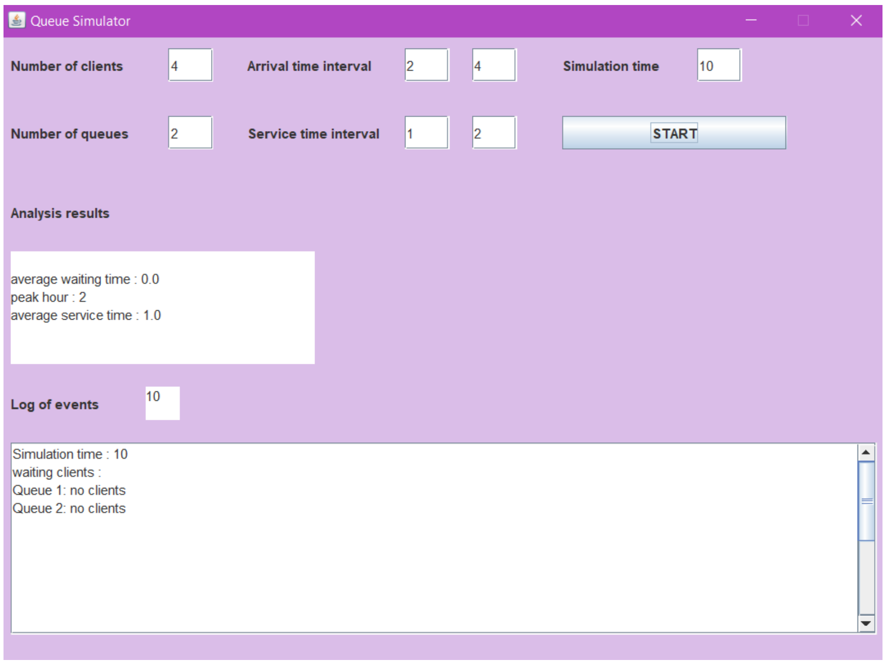


UserInterface	
UserInterface()	
initializePanel(JPanel)	void
clean()	void
updateLog(String)	void
updateAnalysis(String)	void
getNumberOfClientsText()	String
getNumberOfQueuesText()	String
getTimeSimulationText()	String
getArrivalTimeMin()	String
getArrivalTimeMax()	String
getServiceTimeMin()	String
getServiceTimeMax()	String
startActionListener(ActionListener)	void
setCurrentTime(Integer)	void
displayErrorMessage(Exception)	void

The graphical user interface was implemented using Java Swing.
The attributes of the class are : 1 button(for starting the simulation) 7 JTextFileds (for introducing the parameters of the simulation), 3 JTextAreas (for displaying results from the simulation) and 1 JScrollPanel(because the size of the output is variable).

The class contains one method for its initialization, the method *initializePanel* in which the aspect of the GUI is set (bounds of components, colors, sizes), and the rest of the methods are used by the Controller to keep the connection between the View and the Model up to date, and display error messages if exceptions are thrown.

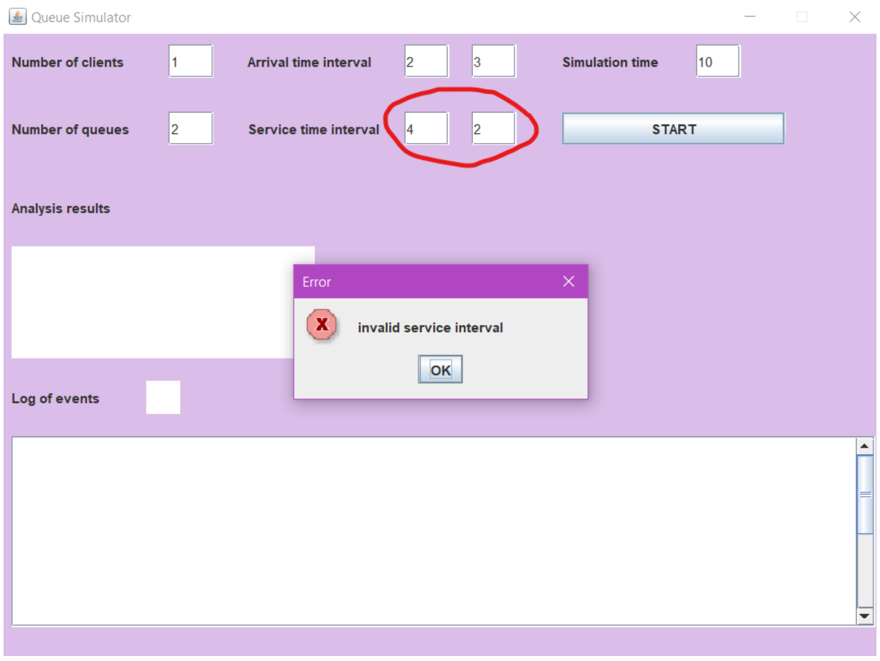
Some examples of the final aspect of the Graphical User Interface are shown below.
Graphical User Interface after a successful simulation



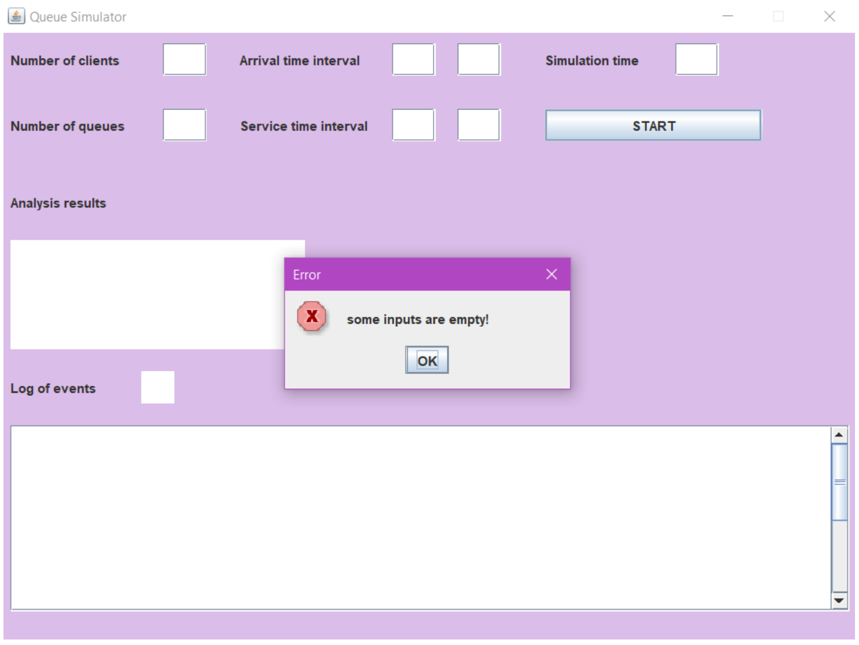
```
average waiting time : 0.0
peak hour : 2
average service time : 1.0
```

```
Simulation time : 10
waiting clients :
Queue 1: no clients
Queue 2: no clients
```

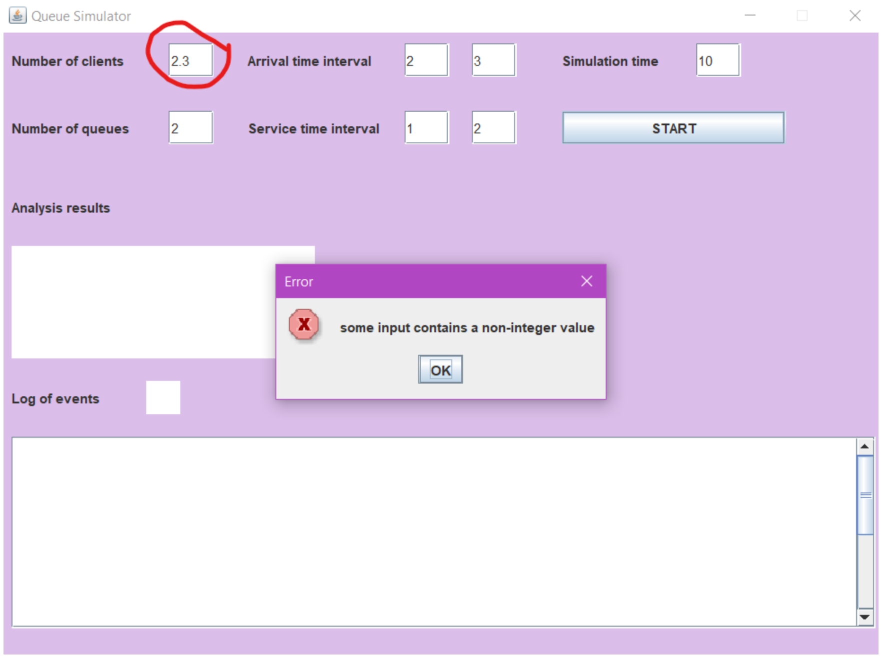
Errors that might be shown in the User Interface



invalid service interval



some inputs are empty!

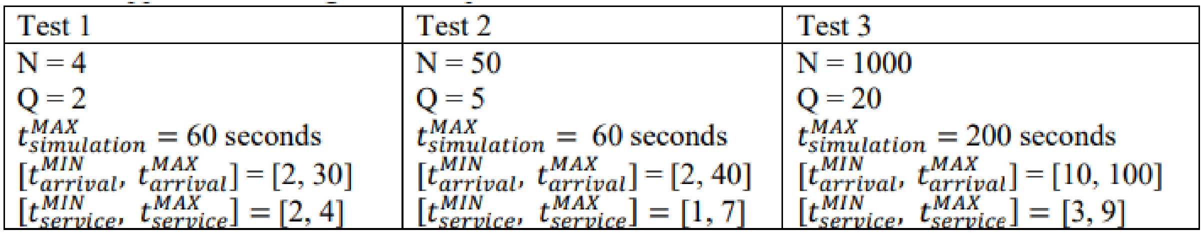


Number of clients: 2.3
Arrival time interval: 2 3
Simulation time: 10
Number of queues: 2
Service time interval: 1 2

some input contains a non-integer value

Testing and Results

For testing the working of the application the following cases were taken into consideration.



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]

The values for each test were introduced in the GUI and the outputs for each test's log were printed in 3 text files. As an example, a simplified part of the result from the first test, can be seen below. It is truncated from simulation time 32, since from that moment till the end, nothing happens.

Test 1 (content of test1.txt)

```
Simulation time : 0
waiting clients : ( 3 7 3 )( 1 18 3 )( 2 27 3 )( 4 28 3 )
Queue 1: no clients
Queue 2: no clients

Simulation time : 1
waiting clients : ( 3 7 3 )( 1 18 3 )( 2 27 3 )( 4 28 3 )
Queue 1: no clients
Queue 2: no clients

Simulation time : 2
waiting clients : ( 3 7 3 )( 1 18 3 )( 2 27 3 )( 4 28 3 )
Queue 1: no clients
Queue 2: no clients

Simulation time : 3
waiting clients : ( 3 7 3 )( 1 18 3 )( 2 27 3 )( 4 28 3 )
Queue 1: no clients
Queue 2: no clients

Simulation time : 4
waiting clients : ( 3 7 3 )( 1 18 3 )( 2 27 3 )( 4 28 3 )
Queue 1: no clients
Queue 2: no clients

Simulation time : 5
waiting clients : ( 3 7 3 )( 1 18 3 )( 2 27 3 )( 4 28 3 )
Queue 1: no clients
Queue 2: no clients

Simulation time : 6
waiting clients : ( 3 7 3 )( 1 18 3 )( 2 27 3 )( 4 28 3 )
Queue 1: no clients
Queue 2: no clients

Simulation time : 7
waiting clients : ( 1 18 3 )( 2 27 3 )( 4 28 3 )
Queue 1: ( 3 7 3 )
Queue 2: no clients

Simulation time : 8
waiting clients : ( 1 18 3 )( 2 27 3 )( 4 28 3 )
Queue 1: ( 3 7 2 )
Queue 2: no clients

Simulation time : 9
waiting clients : ( 1 18 3 )( 2 27 3 )( 4 28 3 )
Queue 1: ( 3 7 1 )
Queue 2: no clients

Simulation time : 10
waiting clients : ( 1 18 3 )( 2 27 3 )( 4 28 3 )
Queue 1: no clients
Queue 2: no clients

Simulation time : 11
waiting clients : ( 1 18 3 )( 2 27 3 )( 4 28 3 )
Queue 1: no clients
Queue 2: no clients

Simulation time : 12
waiting clients : ( 1 18 3 )( 2 27 3 )( 4 28 3 )
Queue 1: no clients
Queue 2: no clients

Simulation time : 13
waiting clients : ( 1 18 3 )( 2 27 3 )( 4 28 3 )
```

```
Queue 1: no clients
Queue 2: no clients

Simulation time : 14
waiting clients : ( 1 18 3 )( 2 27 3 )( 4 28 3 )
Queue 1: no clients
Queue 2: no clients

Simulation time : 15
waiting clients : ( 1 18 3 )( 2 27 3 )( 4 28 3 )
Queue 1: no clients
Queue 2: no clients

Simulation time : 16
waiting clients : ( 1 18 3 )( 2 27 3 )( 4 28 3 )
Queue 1: no clients
Queue 2: no clients

Simulation time : 17
waiting clients : ( 1 18 3 )( 2 27 3 )( 4 28 3 )
Queue 1: no clients
Queue 2: no clients

Simulation time : 18
waiting clients : ( 2 27 3 )( 4 28 3 )
Queue 1: ( 1 18 2 )
Queue 2: no clients

Simulation time : 19
waiting clients : ( 2 27 3 )( 4 28 3 )
Queue 1: ( 1 18 1 )
Queue 2: no clients

Simulation time : 20
waiting clients : ( 2 27 3 )( 4 28 3 )
Queue 1: no clients
Queue 2: no clients

Simulation time : 21
waiting clients : ( 2 27 3 )( 4 28 3 )
Queue 1: no clients
Queue 2: no clients

Simulation time : 22
```

```
waiting clients : ( 2 27 3 )( 4 28 3 )
Queue 1: no clients
Queue 2: no clients

Simulation time : 23
waiting clients : ( 2 27 3 )( 4 28 3 )
Queue 1: no clients
Queue 2: no clients

Simulation time : 24
waiting clients : ( 2 27 3 )( 4 28 3 )
Queue 1: no clients
Queue 2: no clients

Simulation time : 25
waiting clients : ( 2 27 3 )( 4 28 3 )
Queue 1: no clients
Queue 2: no clients

Simulation time : 26
waiting clients : ( 2 27 3 )( 4 28 3 )
Queue 1: no clients
Queue 2: no clients

Simulation time : 27
waiting clients : ( 4 28 3 )
Queue 1: ( 2 27 3 )
Queue 2: no clients

Simulation time : 28
waiting clients :
Queue 1: ( 2 27 2 )
Queue 2: ( 4 28 3 )

Simulation time : 29
waiting clients :
Queue 1: ( 2 27 1 )
Queue 2: ( 4 28 2 )

Simulation time : 30
waiting clients :
Queue 1: no clients
Queue 2: ( 4 28 1 )
```

```
Simulation time : 31
waiting clients :
Queue 1: no clients
Queue 2: no clients

average waiting time : 0.0
peak hour : 7
average service time : 3.0
```

Test 2 (content of test2.txt)

For test 2, just the analysis log is presented, the rest of the result can be seen in the test2.txt file.

```
average waiting time : 1.04
peak hour : 7
average service time : 3.56
```

Test 3 (content of test3.txt)

For test 3, just the analysis log is presented, the rest of the result can be seen in the test3.txt file.

```
average waiting time : 93.564
peak hour : 99
average service time : 5.589
```

Conclusions

The task of implementing a Queues simulator was successfully completed, since it can be seen that the application meets all the functional and non-functional requirements.

In conclusion, the Queues simulator application is a useful tool to simulate an efficient queue management system in which the waiting time for each client is minimal.

The modeling, design and implementation of this software are based on programming techniques and Object-Oriented-Programming principles, so the creation of this app was a great learning opportunity. I’ve learned about working with Threads and concurrent programming features, since I’ve never used these concepts before. Also, the process of writing data into a file in a Java program was something new for me. I think that I have also improved my skills in writing OOP code. Moreover, facing and fixing all sorts of errors and bugs were an important step in learning.

Further Developments

Some improvements that could make the software better would be:

- implementing a more friendly graphical user interface, in which the current evolution is seen as an animation of clients arriving and leaving the queues.
- adding more analysis features to be displayed, such as: average time spent by the client in the store before arriving at the queue, average waiting time per queue or the total number of clients that were served during the simulation time.

Bibliography

- https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html
- https://www.baeldung.com/java-blocking-queue
- https://www.baeldung.com/java-atomic-variables
- https://www.programcreek.com/2011/03/java-write-to-a-file-code-example/
- https://stackoverflow.com/
- Assignment 2 support presentation