

# Queues Simulation Application

Buzea Vlad-Calin

Group: 30422

<b>1. Objective of the assignment .....</b>	<b>3</b>
<b>2. Problem analysis .....</b>	<b>3</b>
<b>2.1. Modeling (CRC CARDS) .....</b>	<b>3</b>
<b>2.2. Use cases.....</b>	<b>6</b>
<b>2.3. Scenarios .....</b>	<b>7</b>
<b>3. Design .....</b>	<b>8</b>
<b>3.1. UML Class Diagram .....</b>	<b>8</b>
<b>3.2. Data structures .....</b>	<b>10</b>
<b>3.3. Algorithms .....</b>	<b>10</b>
<b>3.4. Interface .....</b>	<b>11</b>
<b>4. Implementation and testing .....</b>	<b>11</b>
<b>5. Conclusions, what has been learned, further developments .....</b>	<b>13</b>
<b>6. Bibliography .....</b>	

# 1.Objective of the assignment/ Task description

## Objective

Design and implement a simulation application aiming to analyze queuing based systems for determining and minimizing customers waiting time.

## Description

Queues are commonly seen both in real world and in the models. The main objective of a queue is to provide a place for a "customer" to wait before receiving a "service". The management of queue based systems is interested in minimizing the time amount its "customers" are waiting in queues. One way to minimize the waiting time is to add more servers, i.e. more queues in the system (each queue is considered as having an associated processor) but this approach increases the costs of the supplier. When a new server is added the waiting customers will be evenly distributed to all current available queues.

The system should simulate a series of customers 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. To calculate waiting time we need to know the arrival time, finish time and service time. The arrival time and the service time depend on the individual customers – when they show up and how much service they need. The finish time depends on the number of queues, the number of other customers in the queue and the service needs of those other customers.

### Input data:

- Minimum and maximum interval of arriving time between customers;
- Minimum and maximum service time;
- Number of queues;
- Simulation interval;
- Other information you may consider necessary;

### Minimal output:

- Average of waiting time, service time and empty queue time for 1, 2 and 3 queues for the simulation interval and for a specified interval;
- Log of events and main system data
- Queue evolution
- Peak hour for the simulation interval

## 2. Problem analysis

! Due to the ambiguity of the problem, along this project we will refer to the units that process customers/ clients as being **SERVERS**. Also I would like to point out the fact that the terms *customer* and *client* refer to the same thing and do not bring any different properties. (only linguistically)

### 2.1 Modeling

A simulation of queue management presumes clients coming randomly to open servers. Clients are independent entities. They do not reference one another. A client chooses the queue based on the approximate service time of the customers already in a queue. For instance, if we have 2 servers one with a client will a full basket and one with 2 clients with only one product each, we will choose the second server.

We presume that a customer can't arrive at the same time as another customer, fact sustained even by the minimum and maximum interval of arriving time between customers requested in the input data. So, we can assume that the unit step time is one minute. Thus the simulation can be sequentially traversed, having the step size one minute.

The servers however must run simultaneously and process clients independent of the queue size and other events that happened in other servers. The “step” of each server is a client. At each server a queue is formed, which can also be referred to as a Linked List with the FIFO property.

#### CRC Cards

Client	
<ul style="list-style-type: none"> <li>Stores and can return the <u>arriving time</u>, the <u>service time</u> and the <u>waiting time</u></li> </ul>	<u>-arriving time</u> <u>-service time</u> <u>-waiting time</u>
<ul style="list-style-type: none"> <li>Each store uniquely identifies its customers, generally using the tax receipt number</li> </ul>	-ID
<ul style="list-style-type: none"> <li>Can be assigned to a queue</li> </ul>	
<ul style="list-style-type: none"> <li>Can move to another queue</li> </ul>	

Server	
<ul style="list-style-type: none"> <li>Runs in parallel time with other servers</li> </ul>	-extends Thread
<ul style="list-style-type: none"> <li>Keeps track of the queue (FIFO Linked List), containing customers</li> </ul>	-LinkedList<Client> queue
<ul style="list-style-type: none"> <li>Adds a client to its queue (add as last)./ Receives clients (PUSH)</li> </ul>	-addClient(Client c)
<ul style="list-style-type: none"> <li>Client leaves the queue.</li> </ul>	-extractClient()
<ul style="list-style-type: none"> <li>Can be open or closed</li> </ul>	-boolean open -isOpen() -openServer() -closeServer()
<ul style="list-style-type: none"> <li>Knows the current time</li> </ul>	startTime< <u>currentTime</u> <endTime
<ul style="list-style-type: none"> <li>Has a number (ex Server 1 ,Server 2, Server 3...)</li> </ul>	-number
<ul style="list-style-type: none"> <li>Computes the waiting time for the queue</li> </ul>	-getMaxTime
<ul style="list-style-type: none"> <li>Knows how many clients are in the queue</li> </ul>	-nbClients
<ul style="list-style-type: none"> <li>Sees the first customer</li> </ul>	
<ul style="list-style-type: none"> <li>Logs all events that occur</li> </ul>	String log
<ul style="list-style-type: none"> <li>Processes clients. (POP)</li> </ul>	Run()

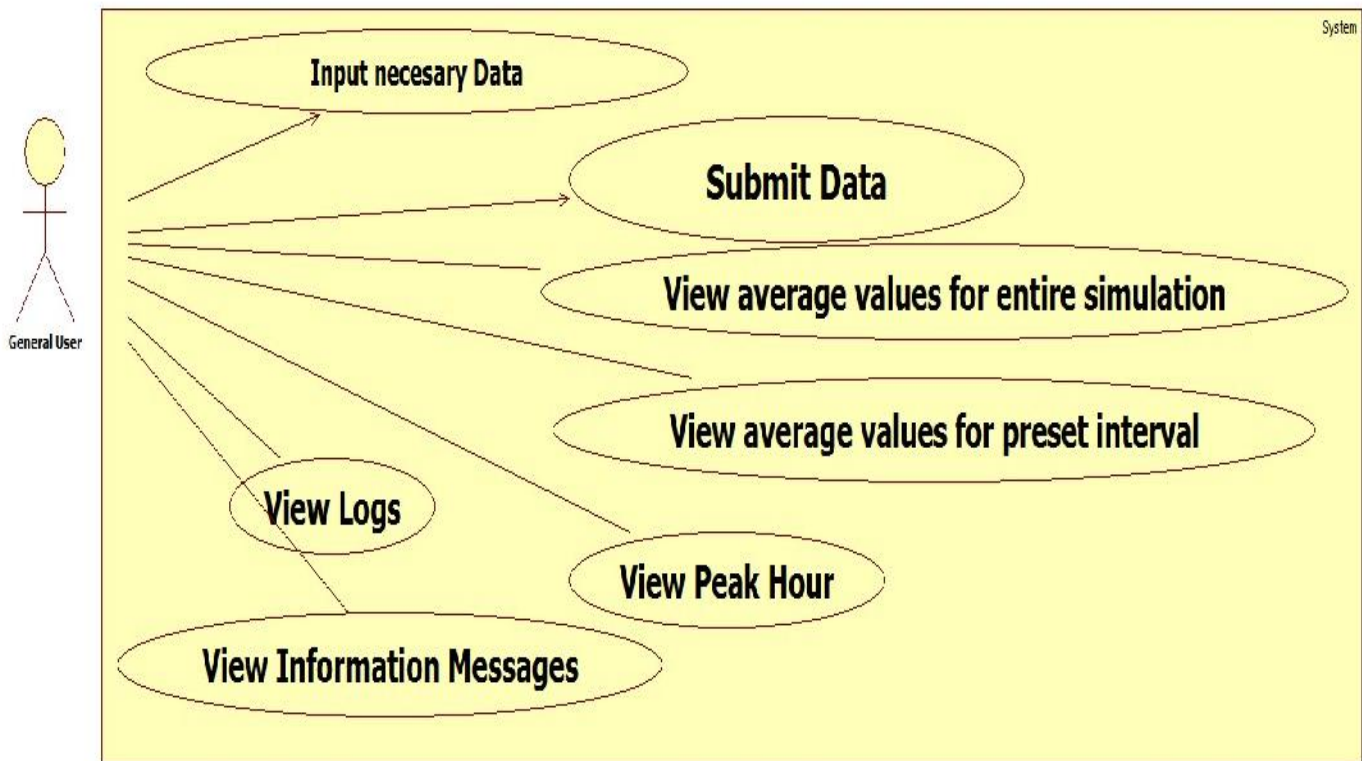
<b>Simulation</b>	
<ul style="list-style-type: none"> <li>Should put servers and clients in correlation. Clients arrive randomly, and must be assigned to a queue. Because Client generation has so many particularities, it calls for a separate class</li> </ul>	<b>CustomerGenerator</b>
<ul style="list-style-type: none"> <li>Has a beginning time and an end time</li> </ul>	-startTime -endTime
<ul style="list-style-type: none"> <li>Has the same reference unit of time (Synchronizes) with the servers. They both reference the minute as basic time step</li> </ul>	Extends Thread
<ul style="list-style-type: none"> <li>- Simulates the queuing, based on the transmitted number of queues</li> </ul>	queuesNb <b>Server</b> servers[queuesNb]
<ul style="list-style-type: none"> <li>Has as constants the minimum length a queue needs to have in order be open and efficient and the maximum time a customer should wait in line</li> </ul>	MAX_WAITING_TIME MIN_WAITING_TIME
<ul style="list-style-type: none"> <li>Can open and close servers, arranging the scene such that both MAX_WAITING_TIME and MIN_WAITING_TIME conditions are respected.</li> </ul>	arrangeServers()
<ul style="list-style-type: none"> <li>Print message logs: retrieve logs from each server and compute other necessary messages</li> </ul>	File Log, BufferedWriter writeMessages();
<ul style="list-style-type: none"> <li>Transform minutes into real time format</li> </ul>	toTime(int minutes)
<ul style="list-style-type: none"> <li>Compute average waiting time, average service time and idle time for first 3 queues.</li> </ul>	
<ul style="list-style-type: none"> <li>Compute peak hour</li> </ul>	
<ul style="list-style-type: none"> <li>Do the actual action : correlate clients an servers</li> </ul>	run()
<ul style="list-style-type: none"> <li>Know the min length queue and the max length queue</li> </ul>	getMinQueue() getMaxQueue()

<b>CustomerGenerator</b>	
<ul style="list-style-type: none"> <li>Customers must be generated beginning with a start time, until a preset end time</li> </ul>	<u>startTime</u> <u>endTime</u>
<ul style="list-style-type: none"> <li>There is a minimum and maximum time between the generation of clients</li> </ul>	-minArrival -maxArrival
<ul style="list-style-type: none"> <li>Respects the minimum and maximum service time</li> </ul>	minServ maxServ
<ul style="list-style-type: none"> <li>Takes random values for each customer</li> </ul>	<b>Random</b>

<ul style="list-style-type: none"> <li>Generates new Customer, available for queuing</li> </ul>	<b>newClient</b>
<ul style="list-style-type: none"> <li>Performs actions in same step time as the servers and the simulation</li> </ul>	Extends Thread run()

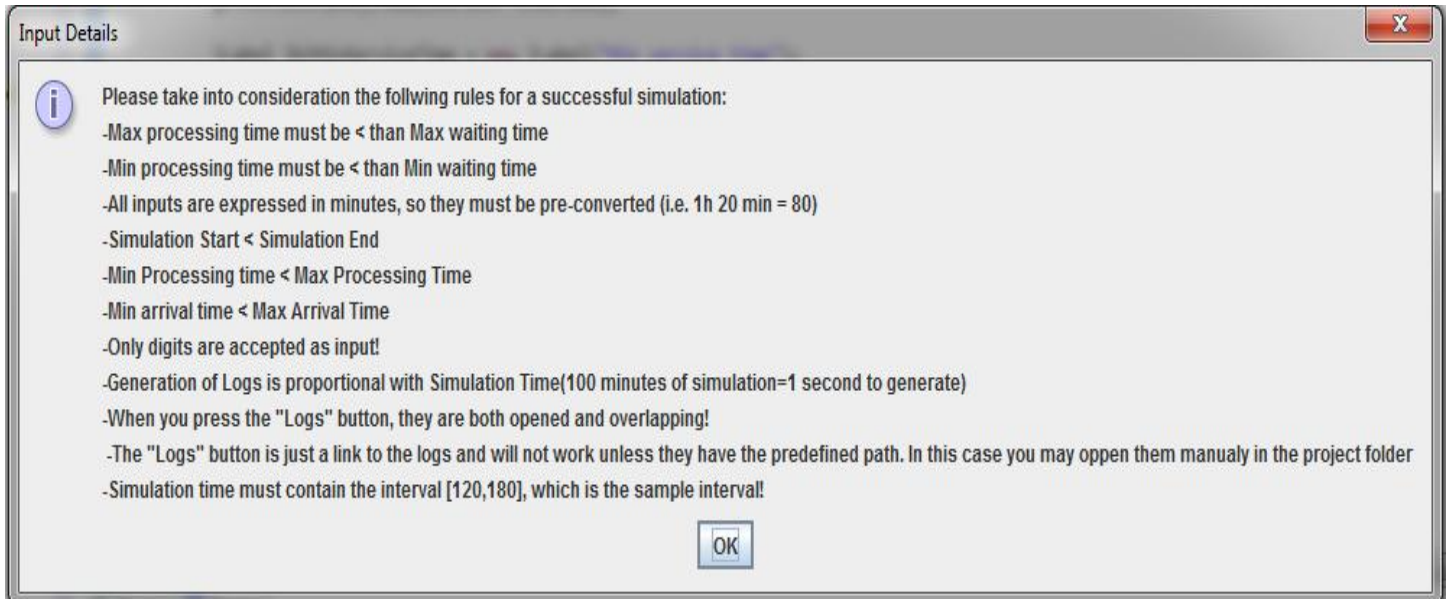
GUI	
<ul style="list-style-type: none"> <li>Displays a frame for input, with minimal validation</li> </ul>	<u>JFrame</u> <u>JLabel</u> <u>TextField</u>
<ul style="list-style-type: none"> <li>Submits data</li> </ul>	JButton submit
<ul style="list-style-type: none"> <li>Displays any informational messages</li> </ul>	JOptionPane
<ul style="list-style-type: none"> <li>Gives user link to Log files</li> </ul>	JButton Logs

### 2.3 Use cases



## 2.4 Scenarios

In order to inform the user about the particularities of the application, a message will be displayed at the beginning of each run. From this point there are 2 main scenarios: The error input scenario and the normal simulation. From the normal case, a small branch diverges, that when the system is not configured. The initial info message will look like this:



### Scenario I: Error Input

If the user tries to input data that is not coherent as format, he still has time repair his mistake until pressing the submit button. At this stage, the data is checked (for minimal requested structure). An information message will be displayed and the user can modify what is needed so that the fields pass validation. The firstly texted data will not be discarded, such that there is no need for re-searching all the content. An example of error message is shown below:



This cycle repeats until correct data is introduced, or user closes the application.

### Scenario II: Normal Case

If data is introduced correctly and passes validation, the simulation is started. Of course, the data validation will not assure that the simulation results are realistically correct. For instance, if the simulation start time is greater than the simulation end time, no simulation should take place. The result is empty logging files, which of course is still relevant. Now, coming back to the normal case scenario, the simulation of the input takes place in the background, and needs an amount of time directly proportional with the simulation time (simulation start – simulation end). This however translates to a real time almost incomprehensible. Thus, we may say it's "instant". One hundred minutes of simulation time translates to one second of real time.

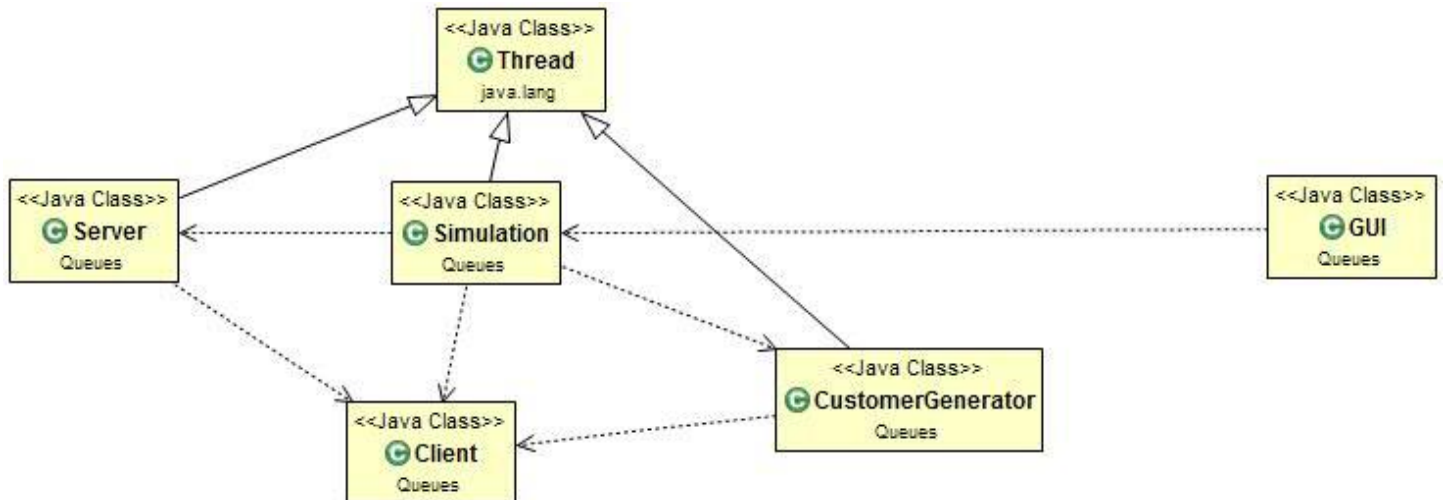
At this stage the user has 2 possibilities:

1. Manually access the Log files in the directory from where the project was run. He must open “LOG.txt” and “Queue Evolution.txt”.
  2. Configure project path such that a link button is correctly referenced and opens the needed files.
- In these files the user observes the wanted information, closes them and also closes the application for complete exit or runs another simulation.

### 3.Design

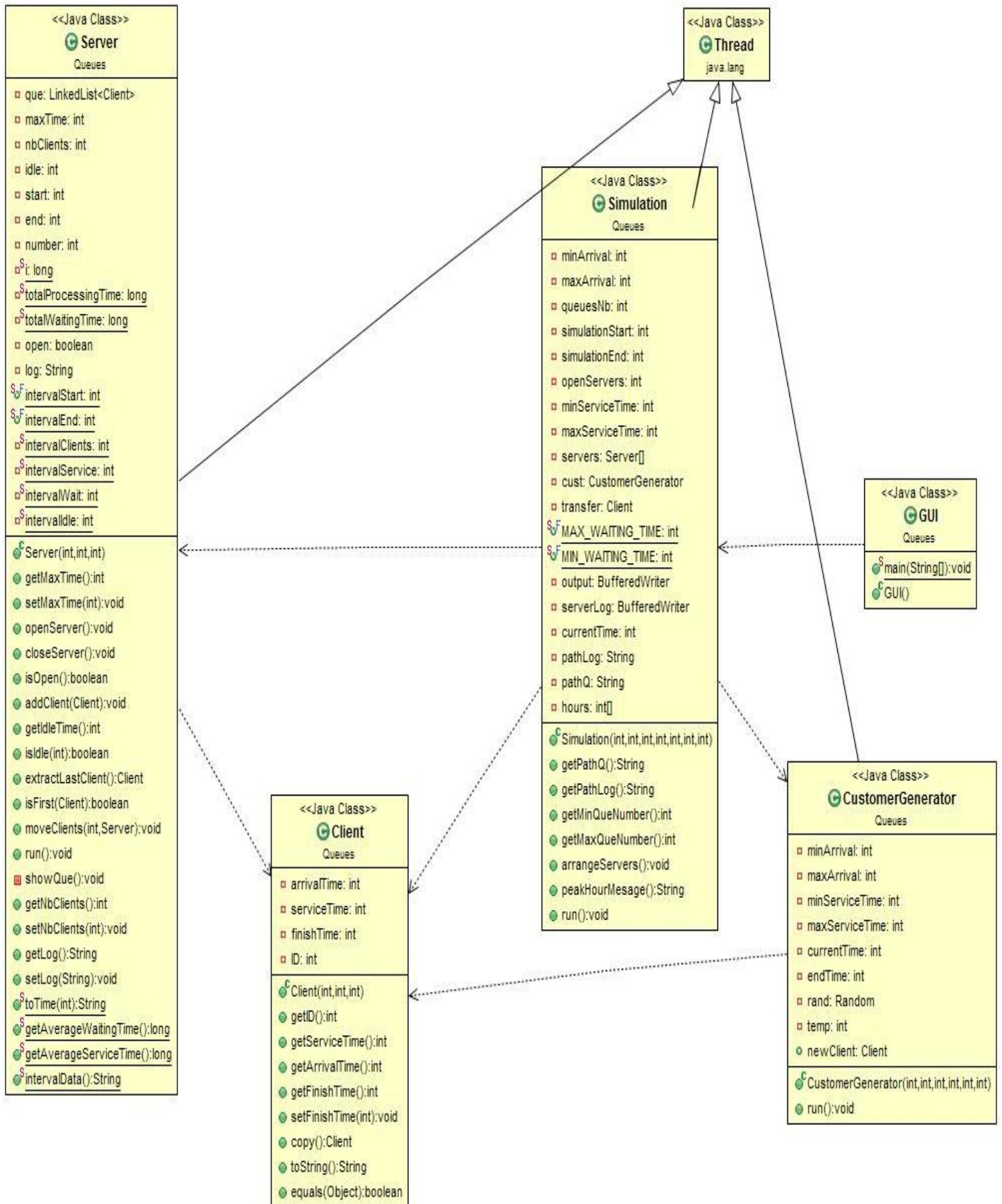
#### 3.1 UML Class Diagram

First I have inserted a Class Diagram without any attributes, such that the relationships can be clearly seen.



The detailed UML will be presented below:





## 3.2 Data Structures

As mentioned in the beginning of this document, when analyzing the CRC cards, we see the need for queues, also known as First In First Out (FIFO) lists. They are implemented in java in the class `LinkedList<T>`.

Doubly-linked list implementation of the `List` and `Deque` interfaces. Implements all optional list operations, and permits all elements (including null).

All of the operations perform as could be expected for a doubly-linked list. Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index.

Note that this implementation is not synchronized. If multiple threads access a linked list concurrently, and at least one of the threads modifies the list structurally, it must be synchronized externally. (A structural modification is any operation that adds or deletes one or more elements; merely setting the value of an element is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the list. If no such object exists, the list should be "wrapped" using the `Collections.synchronizedList` method.

Of greater interest are the following methods:

- `public void addLast(E e)`                   --PUSH  
Appends the specified element to the end of this list.
- `public E pollFirst()`                   --POP  
Retrieves and removes the first element of this list, or returns null if this list is empty.  
Returns: the first element of this list, or null if this list is empty
- `public E pollLast()`                   --Customer leaves  
Retrieves and removes the last element of this list, or returns null if this list is empty.  
Returns: the last element of this list, or null if this list is empty
- `public static <T extends Comparable<? super T>> void sort(List<T> list)`  
Sorts the specified list into ascending order, according to the natural ordering of its elements. All elements in the list must implement the `Comparable` interface. Furthermore, all elements in the list must be mutually comparable (that is, `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the list).

Arrays were also used. We need an array of servers in order to keep an indexed access to all of them and run them ordered. Another array was used for counting the customers that arrived in each hour. In this case we have 24 entries (from 0 to 23). When a client arrives we increment the counter specific to that hour in the array. The hour is actually kept by the index in the array.

## 3.3 Algorithms

We used the sort algorithm imposed by the `List` interface :

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

Sorts the specified list into ascending order, according to the natural ordering of its elements. All elements in the list must implement the `Comparable` interface. Furthermore, all elements in the list must be mutually comparable (that is, `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the list).

This sort is guaranteed to be stable: equal elements will not be reordered as a result of the sort.

The specified list must be modifiable, but need not be resizable.

Implementation note: This implementation is a stable, adaptive, iterative mergesort that requires far fewer than  $n \lg(n)$  comparisons when the input array is partially sorted, while offering the performance of a traditional mergesort when the input array is randomly ordered. If the input array is nearly sorted, the implementation requires approximately  $n$  comparisons. Temporary storage requirements vary from a small constant for nearly sorted input arrays to  $n/2$  object references for randomly ordered input arrays.

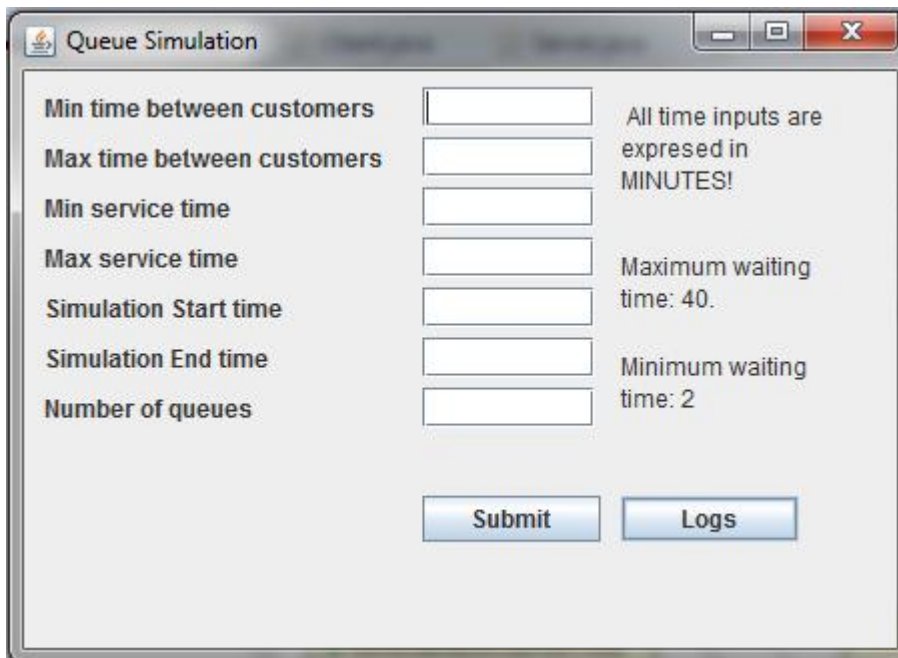
The implementation takes equal advantage of ascending and descending order in its input array, and can take advantage of ascending and descending order in different parts of the same input array. It is well-suited to merging two or more sorted arrays: simply concatenate the arrays and sort the resulting array.

The implementation was adapted from Tim Peters's list sort for Python ( TimSort). It uses techniques from Peter McIlroy's "Optimistic Sorting and Information Theoretic Complexity", in Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, pp 467-474, January 1993.

This implementation dumps the specified list into an array, sorts the array, and iterates over the list resetting each element from the corresponding position in the array. This avoids the  $n^2 \log(n)$  performance that would result from attempting to sort a linked list in place.

### 3.4 Interface

The User Interface has mainly the role to facilitate data introduction and provide easy access to the necessary regions. Thus we want to keep it simple: associate a label to each text field that is used for transmission of one variable.



The screenshot shows a window titled "Queue Simulation". It contains several input fields and labels. On the left, there are labels for "Min time between customers", "Max time between customers", "Min service time", "Max service time", "Simulation Start time", "Simulation End time", and "Number of queues". Each label is followed by a text input field. To the right of these fields, there are two lines of text: "All time inputs are expressed in MINUTES!" and "Maximum waiting time: 40. Minimum waiting time: 2". At the bottom of the window, there are two buttons: "Submit" and "Logs".

One can easily observe that the UI displays the Maximum and Minimum waiting time. Of course these values are constants, but we may change them for a specific customer's needs. A greater maximum waiting time than a queue may ever reach will result in no dynamic behavior of the simulation. Of course a balance is required with the minimum waiting time as also. If the minimum waiting time is too small, the servers will likely enter a loop like case and never close. That is because all the clients will be assigned to queues that have waiting time as small as possible.

Another message can be seen on the frame: "All inputs are expressed in MINUTES!". That is because all fields must take integer values (and positive). Breaking this rule will enter the error scenario.

The UI associates labels to text fields such that the user can have easy handling and a non-ambiguous data transmission. The meaning of the labels:

- "Min time between customers" = minutes that MUST pass until a new customer may appear
- "Max time between customers" = maximum amount of minutes that CAN pass until a new customer appears
- "Min service time" = amount of minutes that MUST pass when serving a client
- "Max service time" = amount of minutes that a client MUSN'T surpass for service
- "Simulation Start time" = minute at which the simulation should start.
- "Simulation End time" = minute at which the simulation should end.
- "Number of queues" = maximum number of servers available for the simulation.

We can also observe the presence of two buttons: "Submit" and "Logs". The "Submit" button validates and transmits the entered data, starting the simulation and the "Logs" button opens the log files.

## 4.Implementation and testing

I will not start copying code! For this purpose you may see it directly. I will however briefly describe the functionality of each method, excepting the Getter and Setter methods which are obvious. I will also skip the fields of each class because they can be seen in the [UML](#) .

For the class Client one can find of interest the following methods

- **public** Client copy()  
- returns a copy of the caller Client, by simply copying the information in each field
- **public** String toString()  
-returns a string containing the information placed in the arrivalTime, serviceTime and finishTime field.
- **public boolean** equals(Object c)  
-Overrides the method in class Object, returning true if the caller and the parameter have the same ID. (IDs are unique identifiers)

For the class Server we have the complementary methods openServer() and closeSerer() that are setter methods fot the open field. Other methods of interest are:

- **public synchronized void** addClient(Client c)  
-adds a client to the queue formed at this server. It is a simple addLast(c) operation on the LinkedList<Client> corresponding to the instance.
- **public** Client extractLastClient()  
-pops the last client from the queue. This is a simple pollLast operation, needed when a client leaves the queue (without being processed)
- **public boolean** isFirst(Client c)  
-checks if the transmitted client in first in line. Returns true in so, false otherwise
- **public void** run()  
-overrides the method from class Thread. Takes the first client from the queue, using pollFirst(), registers the activity, sleeps as much as the client has processing time, then resumes this sequence. All actions are placed as registries in a log. Also, in this method all counters are incremented such that the other computations needed can be performed later.
- **public static** String toTime(int minutes)  
-implements an method for transforming minutes into time format. It is set static because it doesn't refer to a specific instance and needs to be accessible from any point in the application.It simply performs 2 division operations on the minutes, thus obtaining as quotient the hour and as remainder the minutes.
- **public static** String intervalData()  
-Computes a String containing the data regarding the preset interval. In our case, we just divide the sums of waiting and processing time contained between intervalStart=120 and intervalEnd=180 by the number of clients counted in this interval. They are already formed in the run() method, all we need to do is perform division and concatenate to the String we will return. The total idle time is also computed here, so a more facile printing is met

The CustomerGenerator class is another thread that runs in parallel with the servers and performs the task of providing new Customers to be placed in the queues. They are generated in the run() method respecting the specified parameters (min/max processing time, min/max arrival time). This is easily done by an instance of the class Random. We use the nextInt(int n) method:

```
public int nextInt(int n)
```

Returns a pseudorandom, uniformly distributed int value between 0 (inclusive) and the specified value (exclusive), drawn from this random number generator's sequence. The general contract of nextInt is that one int value in the specified range is pseudorandomly generated and returned. All n possible int values are produced with (approximately) equal probability

So nextInt((maxServiceTime-minServiceTime)) + minServiceTime will randomly generate our service time. This will be assigned to the newClient. In the same manner we will generate a random sleep time between minArrival and maxArrival in which no newClient will be created.

The Simulation class ties all the previous classes into a harmonic functionality. We operate on an array of servers, that are assigned clients if the server is open.

The methods getMinQueNumber() and getMaxQueNumber() simply select the desired index (ID) of the server that is open and has minimum, respectively maximum waiting time.

- **public synchronized void** arrangeServers()  
-this is the most complex method of the application, along with the run() method that uses it. This method first checks if the max\_waiting\_time is surpassed. If so it opens new servers until it will no longer be surpassed. Of course when a new server is opened, the clients already in a queue will move to it. We presume an ideal case, when the clients who first got to a queue are let by the other to be first in line at the new queue as also.  
-if the max\_waiting\_time is not surpassed, then we check if a server is inefficient (has waiting time bellow min\_waiting\_time). If the case is so we simply close the server, without discarding the clients in the queue. They will be processed, but no further clients will be able to attend the queue.
- **public void** run()  
-simply steps into the simulated interval by sleeping one minute at a time. If at a step a new client is detected, the he will be added to the queue with minimum waiting time. Before all this the arrangeServers() method is called.  
-at the end of the simulation time the logs are printed in the corresponding files

The **Testing** has been done using the Test class. The results of the tests are provided within the project folder. Also, a video of the tested cases will be found. This can also serve as **Results** for the application.

## 5. Conclusions, what has been learned, further developments

From this homework I have learned how to use threads, what is the role of synchronization and how thread parallelism works. Also, this has been my first experience with the Random class and Collections.

Further developments:

- Universal available "Logs" button
- Display interval data and peak hour on GUI panel not in LOG

## 6. Bibliography

- JDK(j2se7)
- <http://www.youtube.com/watch?v=F-CkaU8aQZI>
- <http://stackoverflow.com/>