Dan Mitrea
Group 30421
Year of Study : 2016 - 2017

Technical University of Cluj Napoca

Programming Techniques
-Assignment 2-
Queue management system

Contents

Assignment Objective:

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

Description

Queues are commonly used to model real world domains. 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. 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 service supplier. When a new server is added the waiting customers will be evenly distributed to all current available queues.

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. 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 clients – when they show up and how much service they need. The finish time depends on the number of queues, the number of clients in the queue and their service needs.

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:
- The 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 (other useful information may be also considered);

- Log of events and main system data;
- Queue evolution;
- Peak hour for the simulation interval;

Problem Analysis:

Simulations are very important in our days. As our daily problems become more and more complex, designing a real application or project directly implies a risk that is not worth taking. A better solution would be to design a virtual simulation to test your plan before putting it in action. As a result, the simulation will replicate the real plan, but with minimal resources, and can greatly help us find out if our plan works as it should, or if it has some flaws that we did not foresee. We must clearly know the problem so that we can create an accurate model of it. Our simulation must have the same logic as the real project we want to simulate. Also, we need a clear representation of the data, and of the results of the simulation, so that we can see if the results are as they should or not .

Queues are a very common thing in our days. As our society has become such greatly based on consumption, everywhere we go, we see people standing at queues, waiting to buy the newest phone, or just some food to eat. Managing these queues can be extremely beneficial both for the shop (at being more efficient with its customers) and also for the clients (by staying less at a queue, they can have more time for other activities). Our simulation can be used in a large variety of cases, ranging from a local convenience store, to any shop in every mall.

Modelling

Scientific modelling is an activity with the aim of making a particular part or feature of the world easier to understand , define, quantify, visualize , or simulate by referencing it to existing and usually commonly accepted knowledge . It requires selecting and identifying relevant aspects of a situation in the real world and then using different types of models for different aims, such as conceptual models to better understand, operational models to operationalize , mathematical models to quantify, and graphical models to visualize the subject.

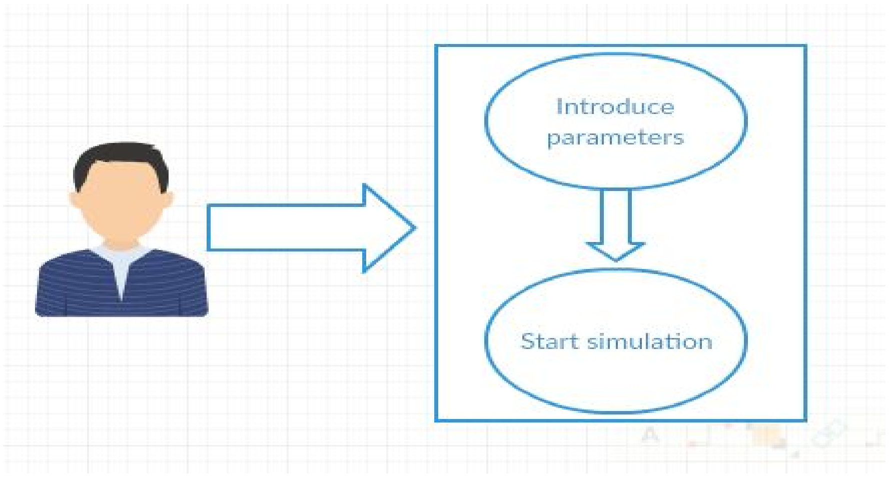
In our case, the best way to model a queue, is by using the data structure with the same name. Utilizing the F I F O concept (first in, first out), it perfectly describes a modern day queue. The operations of adding and removing a client from the queue, can be done directly in the queue class. The operations that deal with managing the queues and properly adding the clients in the shortest queue must be done from a controller class, that simulates the flow of a shop, or anything that needs to optimize its queues. Because each queue will evolve by its own, we can manage them in parallel, for a better optimization of the simulation and of the resources that we have.

Threads:

In computer science, a **thread** of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system. The implementation of threads and processes differs between operating systems, but in most cases a thread is a component of a process. Multiple threads can exist within one process, executing concurrently and sharing resources such as memory, while different processes do not share these resources. In particular, the threads of a process share its executable code and the values of its variables at any given time.

Multithreading is mainly found in multitasking operating systems. Multithreading is a widespread programming and execution model that allows multiple threads to exist within the context of one process. These threads share the process's resources, but are able to execute independently. The threaded programming model provides developers with a useful abstraction of concurrent execution. Multithreading can also be applied to one process to enable parallel execution on a multiprocessing system. Some benefits of multithreading are: responsiveness, faster execution, lower resource consumption, better system utilization, simplified sharing and communication , parallelization .

Scenarios:



The user has the option to introduce the requested parameters, and after filling all fields, he can press a button to start the simulation and see the time of the simulation, and also the queue evolution, each client being represented by a ‘ * ’ character. Separately, the user can see the peak hour of the simulation (the hour with the most clients at the queues) and the average waiting time at the queues running in the simulation.

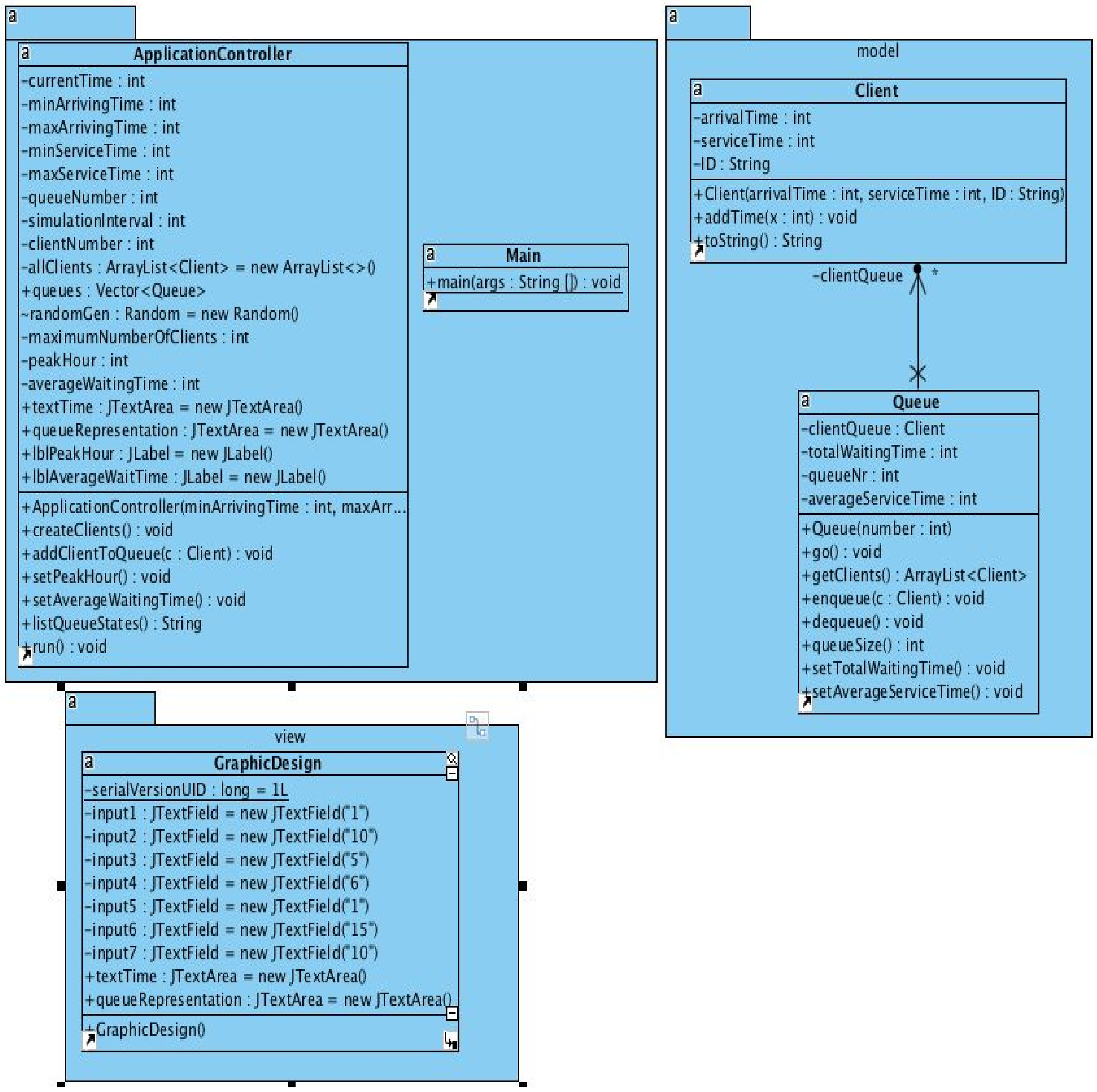
Use case:

I will present how the application runs in case the user introduces the requested parameters.

- the user starts the simulation and a window containing the graphical interface will pop up.
- at each requested field, the user introduces a correct parameter, as written above the text fields, on the labels
- these parameters shape the way the simulation works, with how many queues the user wants, with how many clients the user wants, and for a time period he chooses
- by pressing the ‘ START ’ button, the simulation begins
- at each second, the time of the simulation will be displayed, together with the state of the queues at that given moment, and the peak hour of the simulation, separately

In case the user introduces wrong parameters, or leaves empty text fields, a pop up window appears, signaling that the input data is incorrect.

Design

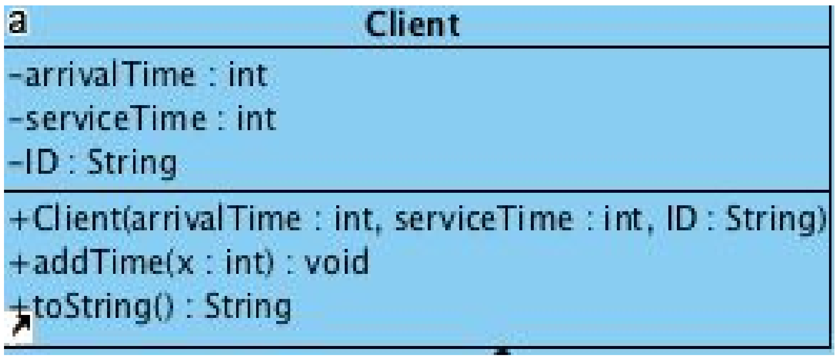

ApplicationController
-currentTime : int
-minArrivingTime : int
-maxArrivingTime : int
-minServiceTime : int
-maxServiceTime : int
-queueNumber : int
-simulationInterval : int
-clientNumber : int
-allClients : ArrayList<Client> = new ArrayList<>()
+queues : Vector<Queue>
~randomGen : Random = new Random()
-maximumNumberOfClients : int
-peakHour : int
-averageWaitingTime : int
+textTime : JTextArea = new JTextArea()
+queueRepresentation : JTextArea = new JTextArea()
+lblPeakHour : JLabel = new JLabel()
+lblAverageWaitTime : JLabel = new JLabel()
+ApplicationController(minArrivingTime : int, maxArr...
+createClients() : void
+addClientToQueue(c : Client) : void
+setPeakHour() : void
+setAverageWaitingTime() : void
+listQueueStates() : String
+run() : void
Main
+main(args : String []) : void
model
Client
-arrivalTime : int
-serviceTime : int
-ID : String
+Client(arrivalTime : int, serviceTime : int, ID : String)
+addTime(x : int) : void
+toString() : String
-clientQueue
*
Queue
-clientQueue : Client
-totalWaitingTime : int
-queueNr : int
-averageServiceTime : int
+Queue(number : int)
+go() : void
+getClients() : ArrayList<Client>
+enqueue(c : Client) : void
+dequeue() : void
+queueSize() : int
+setTotalWaitingTime() : void
+setAverageServiceTime() : void
view
GraphicDesign
-serialVersionUID : long = 1L
-input1 : JTextField = new JTextField("1")
-input2 : JTextField = new JTextField("10")
-input3 : JTextField = new JTextField("5")
-input4 : JTextField = new JTextField("6")
-input5 : JTextField = new JTextField("1")
-input6 : JTextField = new JTextField("15")
-input7 : JTextField = new JTextField("10")
+textTime : JTextArea = new JTextArea()
+queueRepresentation : JTextArea = new JTextArea()
+GraphicDesign()

Packages

I have used the standard MVC design pattern (Model - View - Controller) . The packages ar designed as following:

- the ‘ model ’ package contains the ‘ Client ’ and ‘ Queue ’ classes, with their specific methods.
- the ‘ controller ’ package contains the ‘ Application Controller ’ and ‘ Main ’ classes. Their use will be described immediately.
- the ‘ view ’ package contains the ‘ GraphicDesign ’ class, where I create the user interface and based on the read parameters, it launches the application.

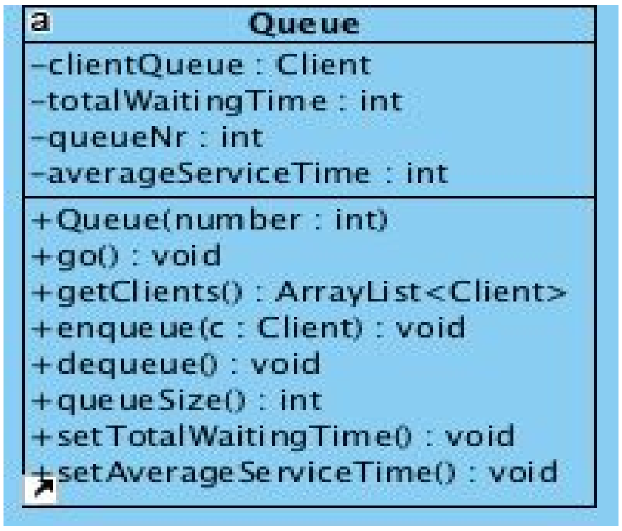
Client class



The client class has been designed to model the client that arrives at the queue. The fields that represent the class ar the arrival time of each client, meaning the time of the simulation at which he / she arrives at the queues. The service time represents the duration for which the client will stay once he / she arrives at the front of the queue. The ID is a random string of 5 characters that represents each client. The class has the usual getters and setters and, more than that, it overrides the toString () method, in order to create a string containing information about the client, that will be later on displayed.

```
@Override
        public String toString() {
                return "Client " + this.ID + " arrives at: " +
this.getArrivalTime() + " and has a service time of: " +
this.getServiceTime();
        }
```

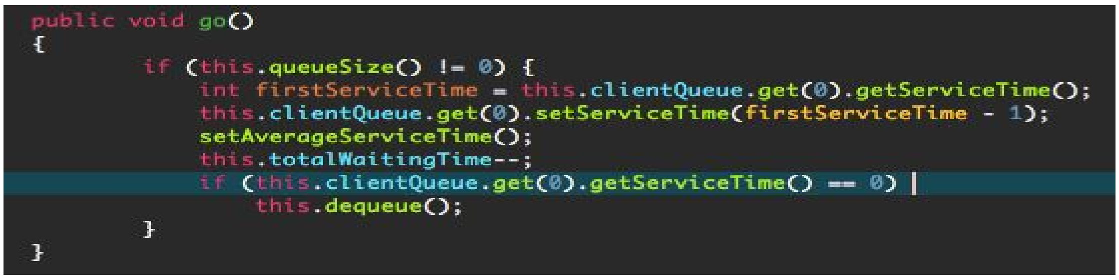
Queue class



```
Queue
-clientQueue : Client
-totalWaitingTime : int
-queueNr : int
-averageServiceTime : int
+Queue(number : int)
+go() : void
+getClients() : ArrayList<Client>
+enqueue(c : Client) : void
+dequeue() : void
+queueSize() : int
+setTotalWaitingTime() : void
+setAverageServiceTime() : void
```

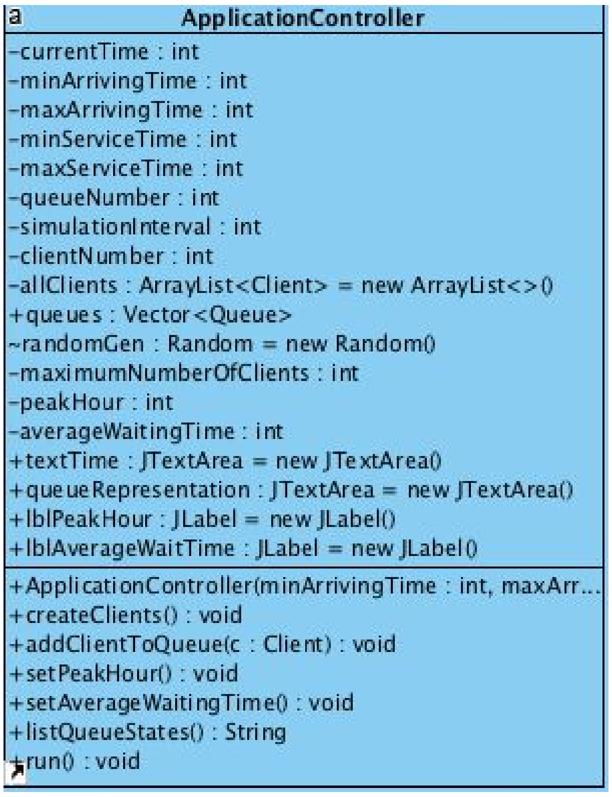
The Queue class is the class that models the actual queue. It contains an Array List with all the clients in it, the total waiting time of the queue, which is the sum of the service time of each client waiting. The attribute queue number is an identifier, set in the controller to be starting from 1 and incremented for each additional queue created. The methods from this class ar pretty straight forward. I have a method that returns all the clients of the queue, getClients() . enqueue() and dequeue() are the methods that implement the operations of the same name over a queue, which is the addition and removal of the first element of the queue. I also have a method that returns the size of the queue, one that sets the total waiting time by summing the service times of each client, and one that sets the average service time, by summing them and dividing to the number of elements of the queue.

The go() method is called at each incrementation of the current time from the controller, and it basically represents what happens to the queue after a second of the simulation has passed. The service time of the first client decrements and, if it reaches 0, that element will be removed from the list, using the dequeue () method. It also decrements the total waiting time of the queue each time it is called.



```
public void go()
{
        if (this.queueSize() != 0) {
            int firstServiceTime = this.clientQueue.get(0).getServiceTime();
            this.clientQueue.get(0).setServiceTime(firstServiceTime - 1);
            setAverageServiceTime();
            this.totalWaitingTime--;
            if (this.clientQueue.get(0).getServiceTime() == 0) |
                this.dequeue();
        }
}
```

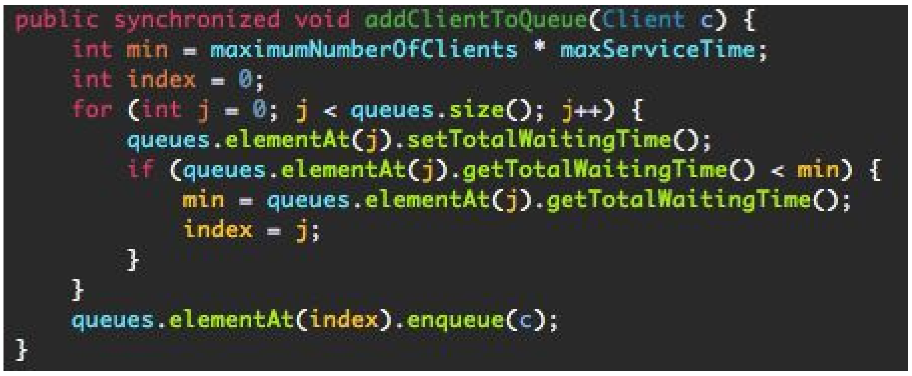
ApplicationController class



```
ApplicationController
-currentTime : int
-minArrivingTime : int
-maxArrivingTime : int
-minServiceTime : int
-maxServiceTime : int
-queueNumber : int
-simulationInterval : int
-clientNumber : int
-allClients : ArrayList<Client> = new ArrayList<>()
+queues : Vector<Queue>
~randomGen : Random = new Random()
-maximumNumberOfClients : int
-peakHour : int
-averageWaitingTime : int
+textTime : JTextArea = new JTextArea()
+queueRepresentation : JTextArea = new JTextArea()
+lblPeakHour : JLabel = new JLabel()
+lblAverageWaitTime : JLabel = new JLabel()
+ApplicationController(minArrivingTime : int, maxArr...
+createClients() : void
+addClientToQueue(c : Client) : void
+setPeakHour() : void
+setAverageWaitingTime() : void
+listQueueStates() : String
+run() : void
```

This is the class that manages the whole simulation. The constructor takes as parameters that parameters of the actual simulation, which are : minimum and maximum arriving time between clients, minimum and maximum service time of the clients, the number of queues, time of simulation, the number of queues, and the labels and text areas where it will print the required information. Besides these, I also have an array list of the generated clients, from which they will be added to the queues when the time comes. The method createClients() generates the clients randomly, with an arrival time which is the arrival time of the last client, plus a random value between 0 and the difference between the max arrival time and min arrival time. All the generated clients will be added to the client pool, which is the list allClients().

The method addClientToQueue(Client: c) add the client given as a parameter to the queues. The addition is based on a simple condition, which is: the client is added to the queue with the least totalWaitingTime. Said in other words, it is added to the queue at which it will wait for the least amount of time.



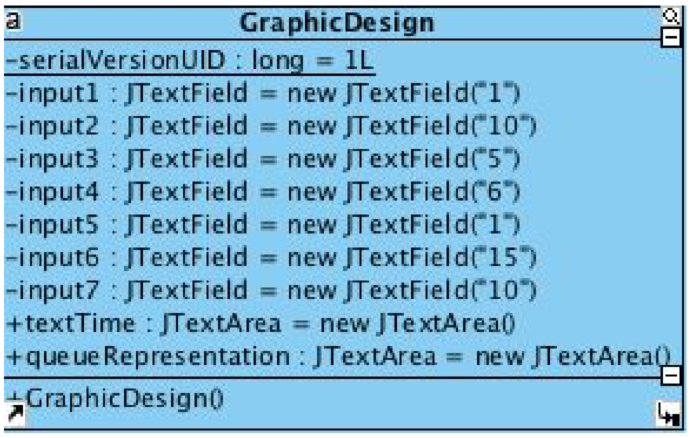
```java
public synchronized void addClientToQueue(Client c) {
    int min = maximumNumberOfClients * maxServiceTime;
    int index = 0;
    for (int j = 0; j < queues.size(); j++) {
        queues.elementAt(j).setTotalWaitingTime();
        if (queues.elementAt(j).getTotalWaitingTime() < min) {
            min = queues.elementAt(j).getTotalWaitingTime();
            index = j;
        }
    }
    queues.elementAt(index).enqueue(c);
}
```

The methods setPeakHour() and setAverageWaitingTime() are used to calculate the peak hour of the simulation at the given moment, and the average waiting time of all the queues. Because they compute these values at a given moment of time, both methods will be called at each second of the simulation.

The method listQueueStates() prints on the text area of the graphical user interface, a simple representation of the queues. On each line, we have the number of the queue, and following that, each client represented by a ‘ * ’. This method will also be called at each second of the simulation in order to display the state of the simulation at that given time.

The method run(), which overrides the method with the same name from the class inherited by my class, which is Thread. This method starts a new thread which will be executed in parralel to the main one. Firstly, we create all the clients by calling the specific method. Then, for each second from 1 to the simulation interval, I will print the current time on a text field. A separate thread will be started for each queue we have in the simulation, and we will keep track of these threads in an array list of threads. Also, we add the given number of queues to the vector of queues defined above. We start each thread, add the clients where they should be added, and then we use join() to get back to having one execution thread, only the initial one. After that, we calculate the peak hour and average waiting time, and display them on the specific labels. In the end, we call the Thread.sleep() method, for 1000 miliseconds, which means that the simulation will wait one second before doing all of these things all over again.

GraphicDesign class

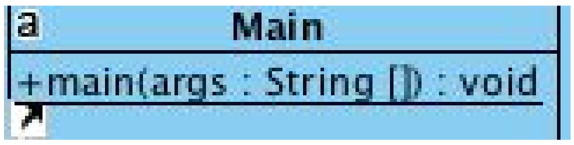


```
GraphicDesign
-serialVersionUID : long = 1L
-input1 : JTextField = new JTextField("1")
-input2 : JTextField = new JTextField("10")
-input3 : JTextField = new JTextField("5")
-input4 : JTextField = new JTextField("6")
-input5 : JTextField = new JTextField("1")
-input6 : JTextField = new JTextField("15")
-input7 : JTextField = new JTextField("10")
+textTime : JTextArea = new JTextArea()
+queueRepresentation : JTextArea = new JTextArea()
+GraphicDesign()
```

This class has the purpose to create the graphical user interface in a friendly way, and to be easy to use. As attributes, I have all the 7 inputs that the user needs to insert, in order for the application to start, the text area where the time will be printed, and the text area where the queues will be represented. Everything is done in the constructor. Here, I will also define a label for each text field, named after what the user must introduce in order to start the application, and also labels for the peak hour and average simulation time, which will be updated every second in the ApplicationController class.

More information about the way the introduced information is processed will be explained in the next chapter.

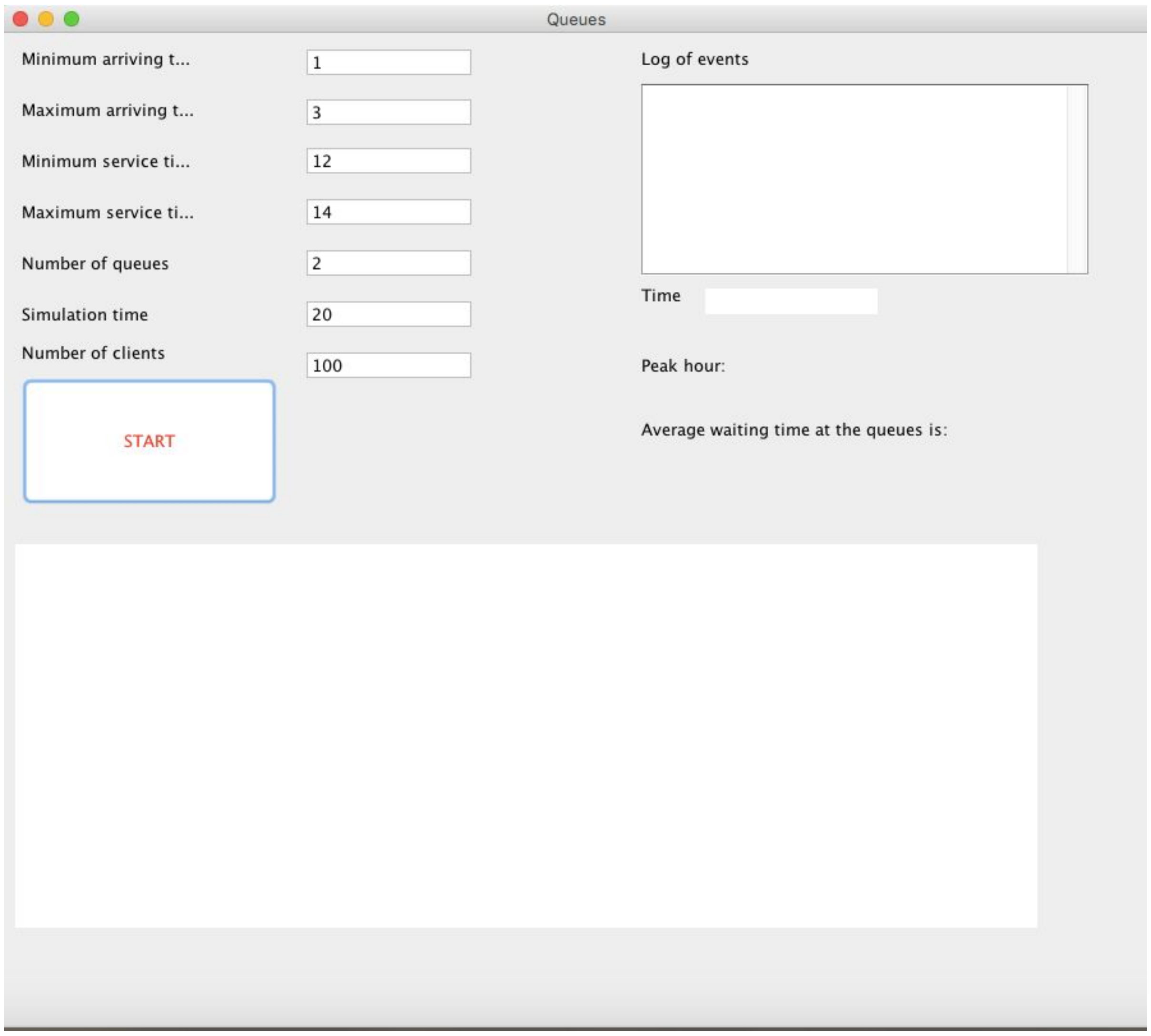
The Main class



```
Main
+main(args : String []) : void
```

The main class is a very simple, yet extremely important class. In the public static void method main(), I just create a new object of the type GraphicDesign, which will automatically open the GUI and start the application if the user introduces the required parameters.

User Interface


Queues

Minimum arriving t...	1
Maximum arriving t...	3
Minimum service ti...	12
Maximum service ti...	14
Number of queues	2
Simulation time	20
Number of clients	100

The graphical user interface is designed to help the user at simulating what he wants. Although it is not that complex, its simplicity is what gives it the desired efficiency. It has clear fields for the parameters the application needs from the user. If the user introduces wrong data, or insufficient data, a pop up window will appear, telling the user what the problem is.

The big button, when pressed, does the following things. It converts all the input strings of the user into integers, stores them into variables, and when it creates an object of type ApplicationController, it passes them as arguments. Then, at each second, a simple representation of the queues is done at the bottom. Also, there are two labels displaying the peak hour and average waiting time at the queues, at the time of the simulation.

Implementation and Testing

I have begun the implementation by creating the simple classes of Client and Queue, and the corresponding methods. By using the toString() method from the client, I have tested each method by printing in the console the computed information and by checking the printed results. After this, I proceeded with the ApplicationController class. Each method has been tested separately by printing in the console and checking if the information is as it should. Of course, all methods have been modified more times after this, for better optimization. The run method makes sure that at each second, I can print or do whatever I want, so I can see if the evolution of data is as it should.

After making sure that all the methods work as intended, I procedeed with the graphical user interface. I made sure that the space for input parameters is large enough to be visible and intuitive to use, but also small enough so that the actual simulation will be the most visible.

Results

The result is, after many hours of coding and debugging, an application that may look simple and extremely easy to implement for a more experienced programmer, but for which I have dedicated a lot of time and effort and is one of the biggest I have ever created. The application makes it very easy for the user to do what he wants, by clearly indicating what he / she should do.

Conclusions

What I learned

From this particular project, the main thing I have learned is the threads. I have understood how te concept works, and also how to implement it, in a simple way. As a result, each queue from my application runs in its separate thread, and by doing so, makes the program run faster and more efficient. I have found out that it is not that difficult to do what processes I want in parallel, and by doing so we greatly increases the efficiency of the application. I am sure that i will use what I have learned in further projects and application I develop.

Further possibilities for improvement

First of all, I can improve the design of the user interface, with more coloured and interactive ways to display the fields and the simulation. Also, I can make the queue number variable, so that if there are fewer clients than a threshold value, no new queue will be opened until that value is surpassed.

Bibliography

1. http://stackoverflow.com
2. https://ro.wikipedia.org/wiki/Pagina_principală
3. https://creately.com
4. https://wordcounter.net