

SIMULATION APP FOR QUEUING BASED SYSTEMS

~ Homework 2 ~

Papita Anda

Group 30421

Programming techniques

TABLE OF CONTENTS

1. Objective	3
2. Problem analysis, modeling, scenarios, use-cases	3
3. Design	5
4. Implementation.....	6
5. Results.....	11
6. Conclusions	12
7. Bibliography	12

1. Objective

The objective of this laboratory homework is the design and implementation of a simulation application meant to be analyzing queuing based systems and determining clients' waiting time. In other words, this application implements a real life issue, that of waiting in line. The problem is commonly met wherever people are going looking for a specific service. The application simulates a more efficient way of distributing clients/people to a line where the waiting time is shorter, thus earning valuable time not only to the benefit of customers, but also to the people providing the service.

Hence, the purpose of this app is minimizing the time people wait when they are in a queue. Such situations can be stressful from a psychological point of view and such a system would help making things quicker and more compatible with the nowadays needs of individuals. From an educational perspective, this assignment was meant to help the student get familiar with the concept of threads and concurrency in Java.

2. Problem analysis, modeling, scenarios, use-cases

In order to be able to track the evolution of a queue in real time, one must understand the concept of concurrency. In concurrent programming, there are two basic units of execution: *processes* and *threads*. In the Java programming language, concurrent programming is mostly concerned with threads. However, processes are also important. Multithreaded execution is an essential feature of the Java platform. Every application has at least one thread — or several, if you count "system" threads that do things like memory management and signal handling. But from the application programmer's point of view, you start with just one thread, called the *main thread*. This thread has the ability to create additional threads.

In this case, the application uses multiple threads, more specifically one for each queue. When the simulation starts, the threads are also starting and wait for customers to arrive. After all clients go past the queue, the threads stop, as does the simulation. The main use in creating lists was the ArrayList and Vector of different

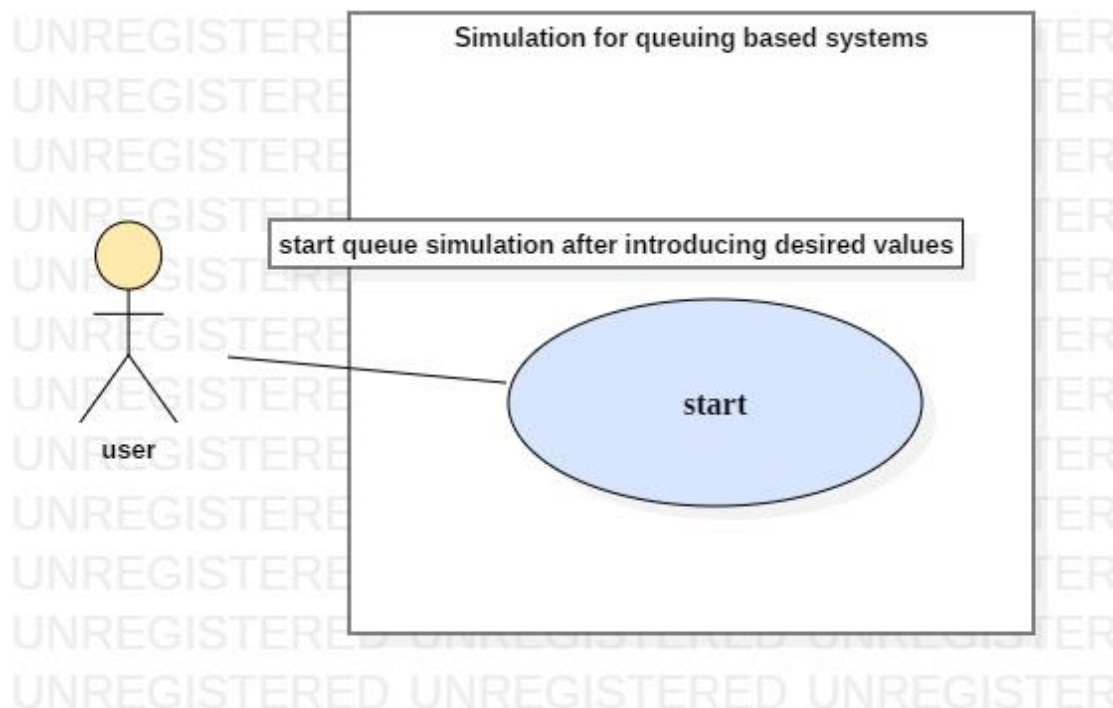
objects. Besides the threads used on each queue, there is a main thread that starts the simulations and helps to insert a recently arrived customer in the waiting line with the least possible waiting time. The thread ends when all customers are done.

Main success scenario:

The observer of the simulation must introduce:

- the total number of clients that will come for a service,
- the number of queues to which the clients will be distributed
- the minimum and maximum interval of arriving time between customers
- the minimum and maximum service time for each customer.
- Between those two sets of intervals a random number will be generated for each client.
- the user must press the start button and the output information will present
 1. the queue evolution (each queue with the specific client who comes to it)
 2. the total waiting time for each queue
 3. the average waiting time

The use case diagram is the following one:

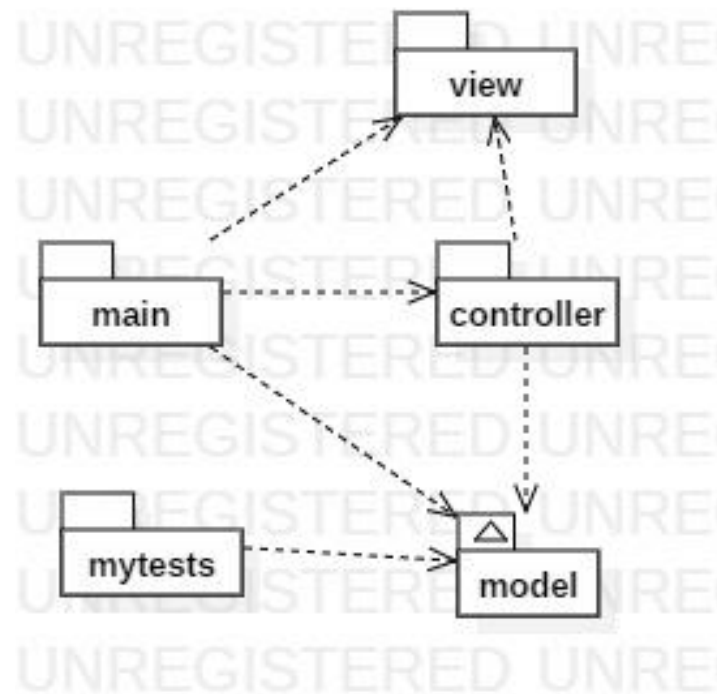


3. Design

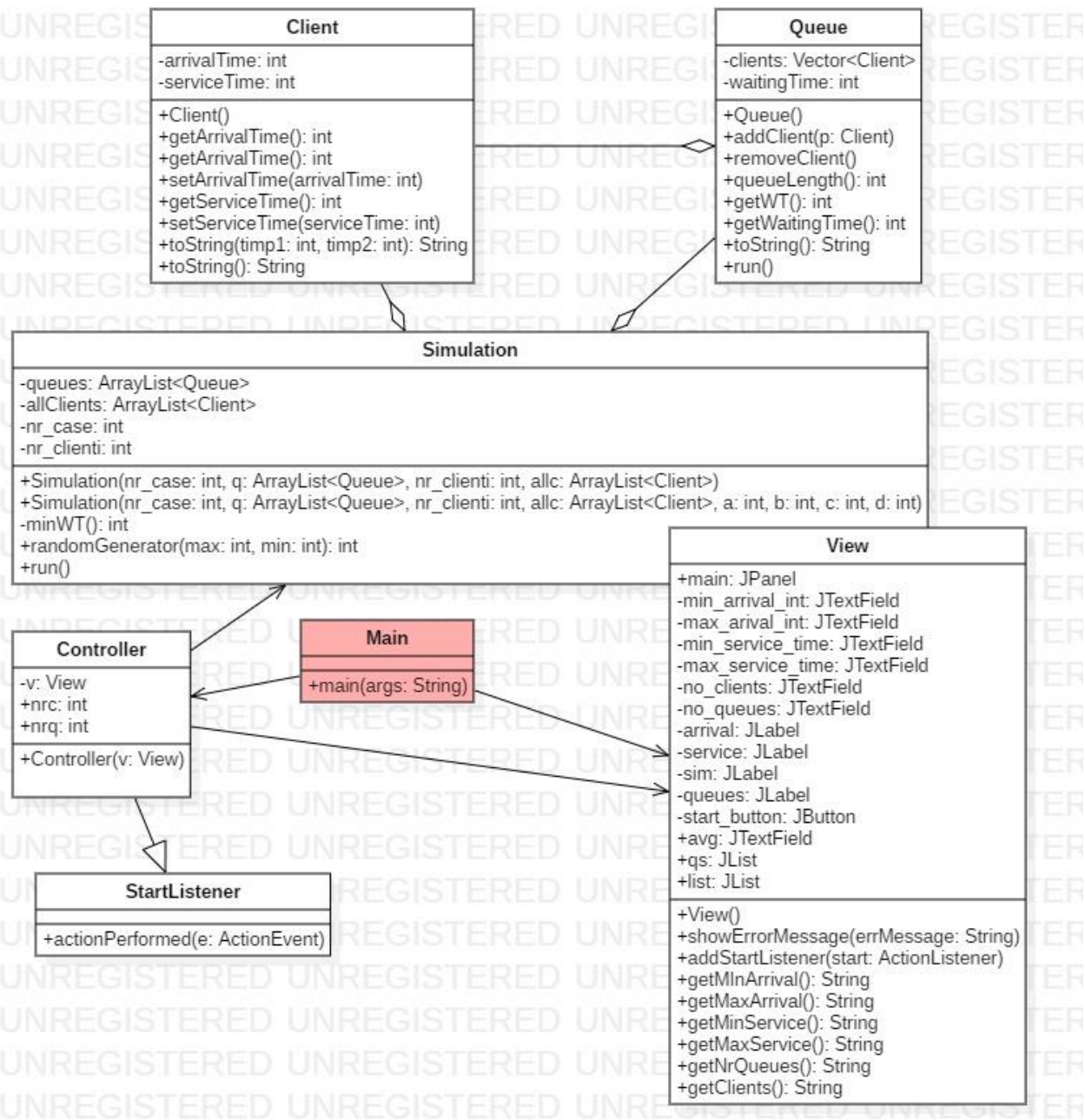
The design of this app follows the MVC pattern. Therefore, the model part is represented by the classes Client, Queue and Simulation, which are found in the model package. These classes are the ones which initialize the whole simulation by creating clients, then queues and then the simulation itself, using threads as mentioned before. The view part, represented by the class View from the view package is responsible for the graphic user interface design. It extends the JFrame class. Finally, the Controller part of the app acts on both model and view and it keeps them separate. It also controls the data flow into model object (simulation) and updates the view whenever data changes. More precisely, it coordinates each button the user can press to perform a specific action (start the simulation by pressing the “Start” button) and helps display the resulting queue evolution and the data that comes with it into a readable form to the user. In addition to these elements, the app contains the Main class from the main package which links the model, controller and view together and starts the execution.

The connections between packages can be seen in the following diagram:

Package relationships



The UML diagram of the project is the following one:



4. Implementation

Client class: this class represents the format of a client which comes in a queue. Each object of type `Client` will have as attributes the arrival time, which is the time the customer arrives for the service, and the service time, the time that the customer's desired service requires. Its methods are mostly getters and setters and two options of the `toString` function. The constructor simply assigns the specified times to the client.

Methods:

1. `public String toString(int timp1, int timp2)` -> returns a string (a valid format) of the client with its two characteristic times
2. `public String toString()` -> this is identical to the first method, but it has no parameters. It will access the attributes of the client

The rest of the methods will be consisting of just getters and setters;

Queue class: this class represents the type of a queue. Each object of type Queue will have a list of objects of type Client, which is represented with the help of Vector. Another attribute is the waiting time of the queue, which will be calculated with respect to the clients' service and arrival time. This class extends the Thread class, which makes the start of a thread on a queue possible. The run() method will be the one executed by the threads on this type of objects

Methods:

1. `public synchronized void addClient(Client p) throws InterruptedException` -> this method adds a client to the end of the vector; it is a synchronized method as it enables a simple strategy for preventing thread interference and memory consistency errors; if the object is visible to more than one thread, all reads or writes to that object's variables are done through synchronized methods. Thus, it contains the notify() method for threads.
2. `public synchronized void removeClient() throws InterruptedException` -> this methods removes the first customer from the line, which is the first element of the vector. As specified above, it is a synchronized method which prevents thread interference.
3. `public synchronized int queueLength() throws InterruptedException` -> this method returns the length of a queue, or more exactly the numbers of clients that are in that queue
4. `public int getWT()` -> this method gets the final waiting time of each queue, which will help calculate other statistics the user will be able to see in the interface
5. `public int getWaitingTime()` -> this method gets the waiting time of a queue with each new client that comes to the queue. In contrast to the previous function for getting the waiting time, this one also considers the arrival time of each client, which further helps to decide to which queue each client would better go (in order to minimize his waiting time)
6. `public String toString()` -> this method turns the contents of a queue into an appropriate way to be viewed by the user and displayed in the interface
7. `public void run()` -> perhaps one of the most important methods in the whole application, this method must be implemented due to the fact that the class

extends the Thread class. It represents an active thread on each queue. All threads that will function on an object of type Queue will implement this method. It will show the queue for a period of time equal with the sum of the arrival and service time for each client. Then it will decrement the service time, thus the waiting time, for each customer. If his service time is null, the client will leave (he will be removed).

8. `public Vector<Client> getClients()` -> function that will get the list of clients of a queue

Simulation class: this class contains a list with all the clients that will come to stand in queues, a list of all the queues, the number of clients and the number of queues. It also extends the Thread class. It represents the class that will implement the way a customer goes to the line with the minimum waiting time.

Methods:

1. `public Simulation(int nr_case, ArrayList<Queue> q, int nr_clienti, ArrayList<Client> allc)` -> this constructor gets the list of the randomly chosen clients based on the parameter allc which already contains that list
2. `public Simulation(int nr_case, ArrayList<Queue> q, int nr_clienti, int a, int b, int c, int d)` -> this constructor does the same thing, but the main difference is that it uses the values that are introduced in the graphic interface by the user. It is used in the Controller class
3. `private int minWT()` -> this method returns the index of the queue which has the minimum waiting time
4. `public int randomGenerator(int max, int min)` -> this method generates random numbers based on the interval given by its parameters [min, max]
5. `public void run()` -> this is the most important of the class, as it generates the way the whole simulation works. This method calls for the main thread. Again, the thread will repeat itself in the amount of the sum of the specified times of a client. The first client in the list of all clients will be inserted in the queue with the minimum waiting time. The reason the first client is chosen is because the list of clients is sorted based on each client's arrival time. Moreover, it contains the way the information is displayed in the graphic interface

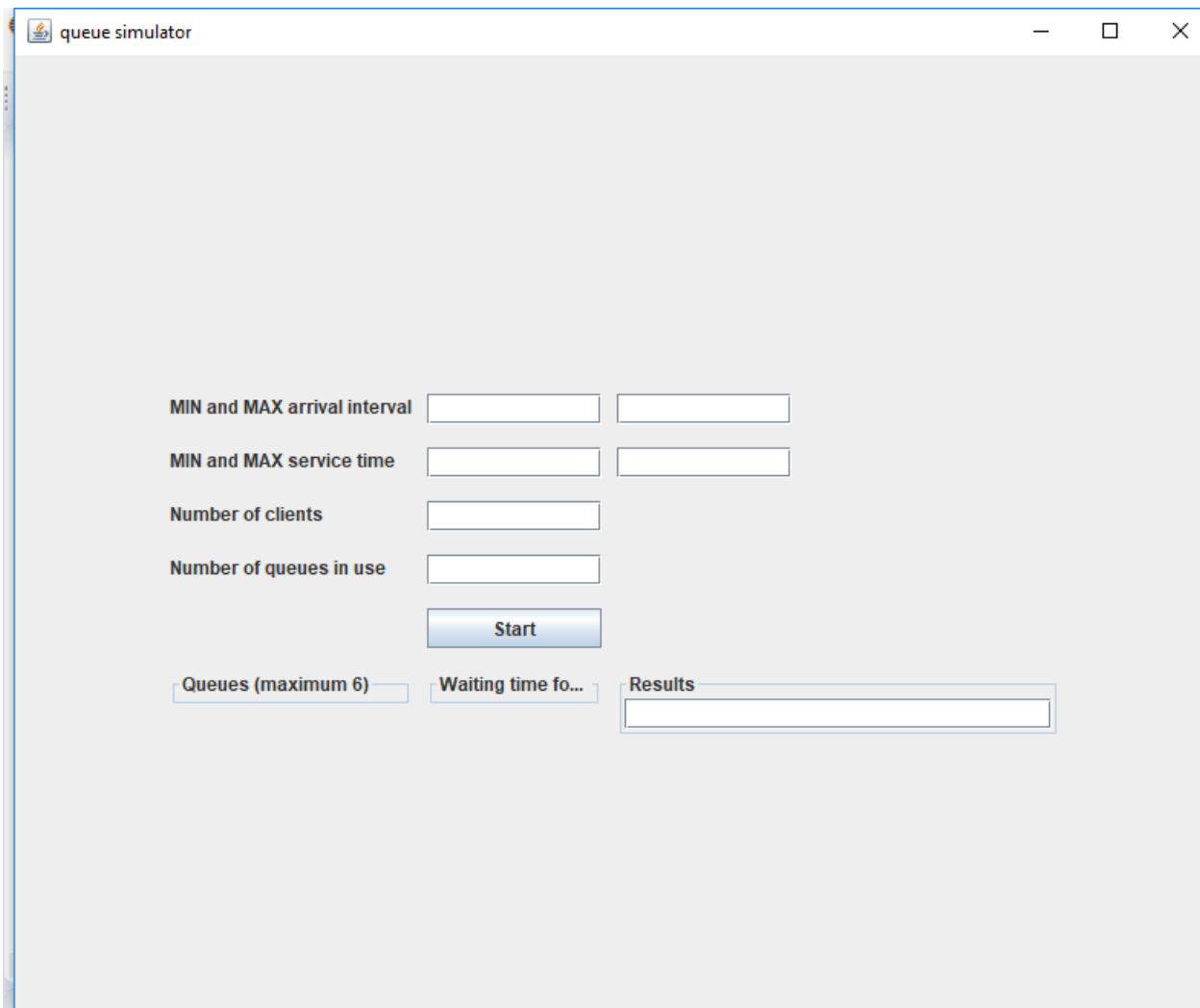
View class: this class creates the interface the user works with regarding to introducing the necessary and desired values for the simulation. It extends the JFrame class. Its components include labels, text fields and buttons. This time I

found the most appropriate and perhaps convenient to use the GridBagLayout with respect to the main panel organization. I set constraints to each component. They refer to the number of columns and lines of the panel. Thus, the panel can be seen as an imaginary matrix and each element is placed in one matrix element. Three more panels were created. They contain JList components which display the evolution of the queues and a text field for additional results. Everything is arranged within the constructor, View().

Additional methods:

1. `public void addStartListener(ActionListener start)` -> creates an event for the start button. All the other buttons are configured in this way and the methods will not be mentioned.
2. `public String getMinArrival()` -> these methods get the values introduced by the user regarding arrival time intervals, service time intervals and numbers of queues and clients

The interface has the following structure:



queue simulator

MIN and MAX arrival interval

MIN and MAX service time

Number of clients

Number of queues in use

Start

Queues (maximum 6) Waiting time fo... Results

Controller class: this class acts on both Simulation and View classes. It controls the data flow that goes into Simulation and updates the View whenever data changes. More precisely, it coordinates the button start which performs a specific task if the user chooses to press it. It consists of one constructor that implements view functions. The class also contains a listener class for the button start, for starting the simulation.

For example:

```
36     }
37     /**
38     * class that implements the action performed when button "start" is pressed
39     * @author anda
40     *
41     */
42     class StartListener implements ActionListener{
43     public void actionPerformed(ActionEvent e) {
44         int mina = Integer.parseInt(v.getMinArrival());
45         int maxa = Integer.parseInt(v.getMaxArrival());
46         int mins = Integer.parseInt(v.getMinService());
47         int maxs = Integer.parseInt(v.getMaxService());
48         int nrq = Integer.parseInt(v.getNrQueues());
49         int nrc = Integer.parseInt(v.getClients());
50         ArrayList<Queue> q = new ArrayList<Queue>(nrq);
51
52         for(int i = 0; i < nrq; i++) {
53             q.add(new Queue());
54             q.get(i).start();
55         }
56         Simulation s = new Simulation(nrq, q, nrc, maxa, mina, maxs, mins);
57         s.start();
58     }
59 }
60 }
61
```

Main class: this is the class that starts the application. It only contains a main() function and it creates a model, a view and a controller.

5. Results

The screenshot shows a window titled "queue simulator" with the following elements:

- Input Parameters:**
 - MIN and MAX arrival interval: 2 and 50
 - MIN and MAX service time: 50 and 100
 - Number of clients: 5
 - Number of queues in use: 3
- Start Button:** A blue button labeled "Start".
- Queues (maximum 6):**
 - QUEUE 0: C-7-73
 - QUEUE 1: C-10-65
 - QUEUE 2: C-19-128
 - QUEUE 1: C-27-132
 - QUEUE 0: C-45-131
- Waiting time for queues:**
 - queue 0 has the total waiting time of: 196
 - queue 1 has the total waiting time of: 190
 - queue 2 has the total waiting time of: 125
 - queue 0 has the total waiting time of: 196
 - queue 1 has the total waiting time of: 190
 - queue 2 has the total waiting time of: 125
 - queue 0 has the total waiting time of: 196
 - queue 1 has the total waiting time of: 190
 - queue 2 has the total waiting time of: 125
- Results:**
 - Average waiting time: 170

As we can see in this picture, the results appear to be correct. All 3 queues are initially empty. The first client arrived, C-7-73 is directed to the queue 0. The two following clients are directed to the next two queues because they are free. The fourth client, however, is directed to the second queue, as it is the queue with the smallest waiting time of all other queues. The same thing happens for the next client. In the middle, *the total waiting time for each queue is displayed. In the results section we can observe the average waiting time for each queue.

*the total waiting time is displayed 3 times by mistake

6. Conclusions

In conclusion, this application should be in high demand in supermarkets should it be more detailed. The client alone cannot estimate how the waiting time for the queues he is waiting in and such system would save a lot of important time. The program successfully simulates a more efficient client distribution for waiting in line.

7. Bibliography

1. <https://docs.oracle.com/javase/tutorial/essential/concurrency/threads.html>
2. <https://www.math.uni-hamburg.de/doc/java/tutorial/uiswing/components/textarea.html>
3. <https://www.geeksforgeeks.org/java-swing-jlist-with-examples/>