Fundamental Programming Techniques 2021

Assignment 2

Queues simulator

Java application

Chereji Iulia-Adela

Group 30424-1

CONTENTS

1. ASSIGNMENT OBJECTIVE

1.1 Main objective

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

The application should simulate (by defining a simulation time $tsimulation$) a series of N clients arriving for service, entering Q queues, waiting, being served and finally leaving the queues. All clients are generated when the simulation is started, and are characterized by three parameters: ID (a number between 1 and N), $tarrival$ (simulation time when they are ready to go to the queue; i.e. time when the client finished shopping) and $tservice$ (time interval or duration needed to serve the client; i.e. waiting time when the client is in front of the queue). The application tracks the total time spent by every client in the queues and computes the average waiting time. Each client is added to the queue with minimum waiting time when its $tarrival$ time is greater than or equal to the simulation time ($tarrival \geq tsimulation$).

The following data should be considered as input data for the application that should be inserted by the user in the application's user interface: - Number of clients (N); - Number of queues (Q); - Simulation interval ($tsimulation\ MAX$); - Minimum and maximum arrival time ($tarrival\ MIN \leq tarrival \leq tarrival$); - Minimum and maximum service time ($tservice\ MIN \leq tservice \leq tservice\ MAX$);

A number of Q threads will be launched to process in parallel the clients. Another thread will be launched to hold the simulation time $tsimulation$ and distribute each client i to the queue with the smallest waiting time when $tarrival\ i \geq tsimulation$.

The log of events saved in a .txt file contains the status of the pool of waiting clients and the queues as the simulation time $tsimulation$ goes from 0 to $tsimulation\ MAX$.

1.2 Secondary objectives

1.2.1 Analyze the problem and identify requirements

The modeling of the problem will be performed (i.e. identify the functional and non-functional requirements, identify the scenarios, detail and analyze the use cases). – Ch. 2

1.2.2 Design the queues simulator

The OOP design decisions will be presented (i.e. UML diagrams, data structures, class design, relationships, packages, user interfaces). –Ch. 3

1.2.3 Implement the queues simulator

The implementation of the classes (i.e. fields and important methods) and of the graphical user interface will be presented. –Ch. 4

2. PROBLEM ANALYSIS

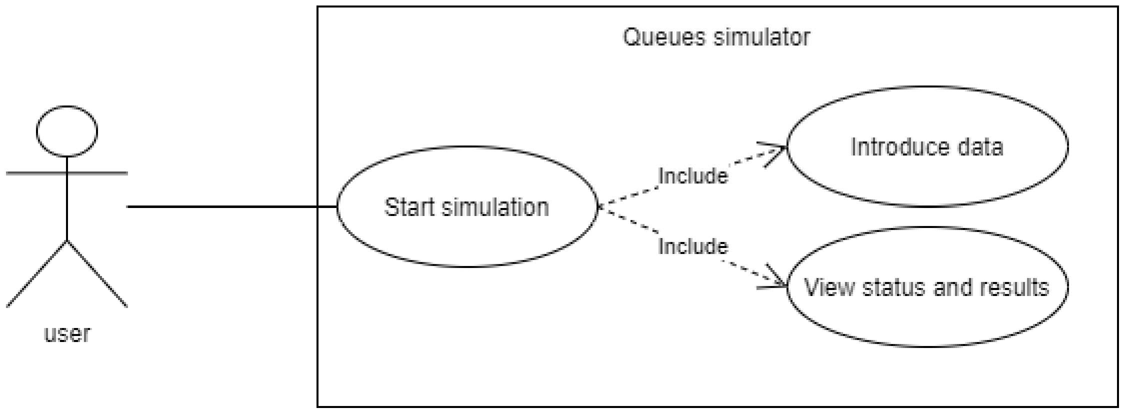
2.1 Functional requirements

- The queues simulator should allow users to insert the number of clients, number of queues, simulation time, minimum and maximum arrival time for the clients and minimum and maximum service time for the clients
- The queues simulator should allow users to start the simulation
- The queues simulator should display the real time log of events (current time, waiting clients, queues status) in the user interface
- The queues simulator should display after the simulation is finished, the average waiting time, average service time and peak hour in the user interface
- The queues simulator should display the real time log of events (current time, waiting clients, queues status) in a .txt file
- The queues simulator should display after the simulation is finished, the average waiting time, average service time and peak hour in a .txt file

2.2 Non-functional requirements

- The queues simulator should be intuitive and easy to use by the user
- The queues simulator should display an error message if one of the inputs is incorrect or in the wrong format
- The queues simulator should display a message if the user tries to modify the simulation data or to start a new simulation while the previous one is still running
- The queues simulator should be able to display the current status of the situation no matter how long (text might not fit in the original frame)
- The queues simulator should generate random clients
- The queues simulator should allocate clients to the queue with the smallest current waiting time (the sum of the service time of all the clients which are already in the queue)
- The queues simulator should display the current status of the simulation in a readable and understandable manner i.e. clients as (ID, arriveTime, serviceTime), queues as Queue nr: clients

2.3 Use cases



2.3.1 Use Case: Start simulation

Primary Actor: user

Main Success Scenario:

1. The user inserts the number of clients, number of queues, simulation time, minimum arrival time, maximum arrival time, minimum service time and maximum service time in the graphical user interface.
2. The user clicks on the “SAVE” button and then on the “START” button.
3. The simulation begins and the current status is displayed in the graphical user interface at every second.
4. The simulation ends and the final results are displayed.
5. The queues simulator is ready to receive new data or to start a new simulation.

Alternative Sequence 1: Incorrect input data

1. The user inserts incorrect data (e.g. other characters besides digits, minimum arrival time > maximum arrival time, minimum service time > maximum service time).
2. The user clicks on the “SAVE” button.
3. The queues simulator displays an error message.
4. The scenario returns to step 1.

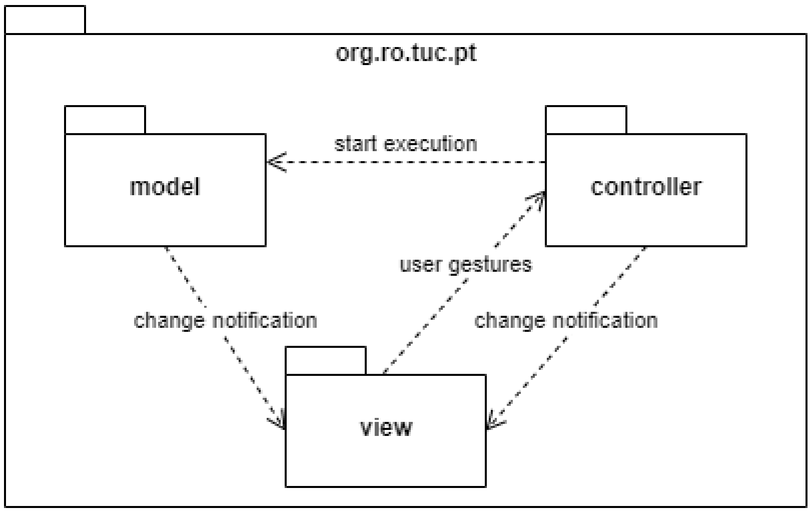
Alternative Sequence 2: The user attempts to start a new simulation or to modify the current simulation while the previous simulation is still running

1. The user clicks on the “SAVE” or “START” buttons while the previous simulation is still running.
2. The queues simulator displays an error message and the current simulation goes on.
3. After the simulation is finished the application is ready to receive new input and to start a new simulation.

3. DESIGN DECISIONS

3.1 Division into packages

The Model View Controller Architectural Pattern was used.



- model package encapsulates the functionality and the data models, and sends information to be printed in the view and in the log of events file.
- view package receives input from the user and displays the results given by the controller and the model.
- controller package handles the user requests, sends data to be processed to the model and sends information to be printed in the view.

3.2 Division into classes

3.2.1 CRC cards

App	
• Creates the View object • Creates the Controller object	• View • Controller

Controller	
• Adds Buttons Listeners • Processes input • Sends information to be printed in the view • Starts the main Thread	• View • SimulationManager

Client	
• Holds the client information • Retrieves the client information	

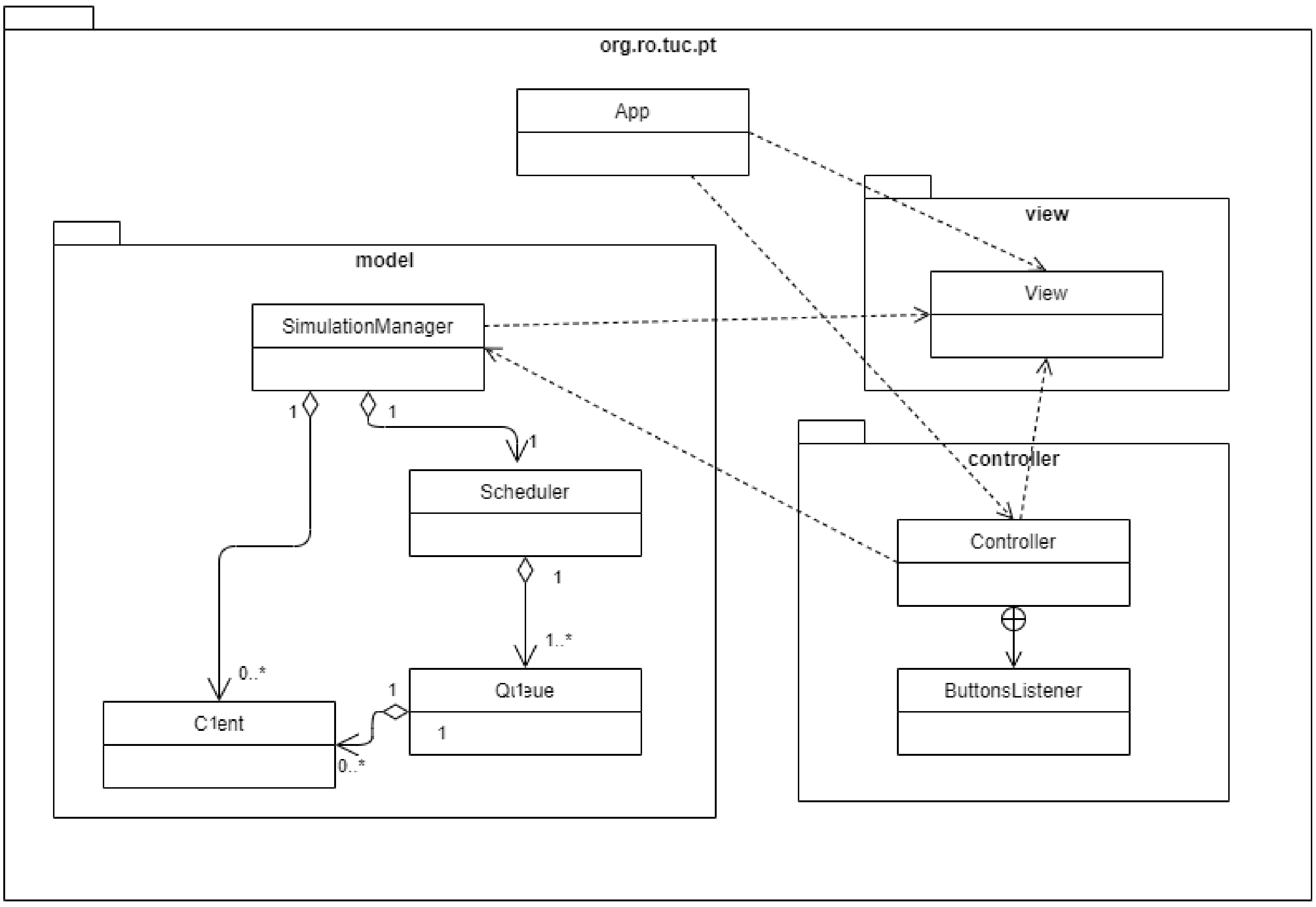
Queue	
• Receives and processes a number of Clients at a time	• Client

Scheduler	
• Holds the queues • Allocates clients to queues	• Queue • Client • SimulationManager

SimulationManager	
• Holds the input from the user • Generates clients • Creates the scheduler • Starts the queues threads • Writes to the log events file and to the user interface	• Client • Scheduler • Controller • View • Queue

View	
• Displays status and messages • Gets input from the user	• Controller • SimulationManager

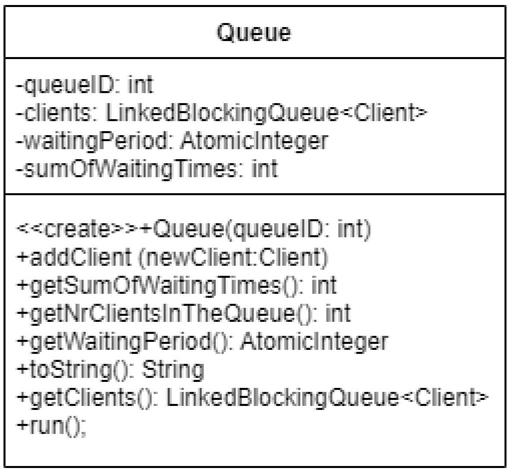
3.2.2 Class diagram



4. IMPLEMENTATION

4.1 Class descriptions

4.1.1 Class Queue


Queue
-queueID: int
-clients: LinkedBlockingQueue<Client>
-waitingPeriod: AtomicInteger
-sumOfWaitingTimes: int
<<create>>+Queue(queueID: int)
+addClient (newClient:Client)
+getSumOfWaitingTimes(): int
+getNrClientsInTheQueue(): int
+getWaitingPeriod(): AtomicInteger
+toString(): String
+getClients(): LinkedBlockingQueue<Client>
+run();

Class Queue implements Runnable. It has a list of Client objects (which are currently in the queue), that is of the type LinkedBlockingQueue because a process will add clients to the queue (scheduler), and another process will extract the clients from the queue (Queue), so it ensures thread safety.

Also the waitingPeriod is an Atomic integer because it is updated in the Queue process and it must be read in the Scheduler process in real time.

The sumOfWaitingTimes field is the sum of all the client's time spent in the queue. It will be used when the simulation ends to compute the average waiting time. It is set to 0 in the constructor of the class.

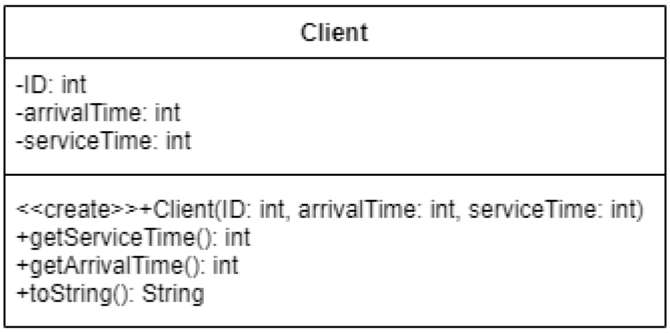
The addClient method adds a new client to the list of clients. It modifies the waitingPeriod of the queue by adding the new client's serving time, and also adds the current waiting period to the sumOfWaitingTimes because that is the period that the new added client will be in the queue.

```
public void addClient (Client newClient)
{
    clients.add(newClient);
    waitingPeriod.addAndGet(newClient.getServiceTime());
    sumOfWaitingTimes+=waitingPeriod.get();
}
```

The run method executes forever, until the application is closed, or a new simulation begins, because at that moment a new SimulationManager is created, and a new Scheduler with new Queues, so the previous queues will be taken by the garbage collector aand their threads will be stopped. The method takes the next client from the queue (if there is one), and sleeps for the client' service time (but every second updates queue's waitingPeriod by subtracting the second that passed), and then polls out the client from the queue, sending it to the garbage collector.

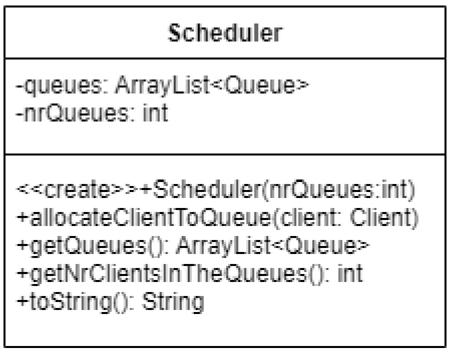
```
@Override
public void run() {
    while(true)
    {
        Client next=clients.peek();
        if (next!=null) //there is a client
        {
            try
            {
                for(int i=0;i< next.getServiceTime();i++)
                {
                    sleep(1000);
                    waitingPeriod.addAndGet(-1);
                }
                clients.poll();
            }
            catch (InterruptedException e) { }
        }
    }
}
```

4.1.2 Class Client



The class Client is used to hold information about Clients. It has and ID, arrivalTime and serviceTime being integer values.

4.1.3 Class Scheduler


Scheduler
-queues: ArrayList<Queue>
-nrQueues: int
<<create>>+Scheduler(nrQueues:int)
+allocateClientToQueue(client: Client)
+getQueues(): ArrayList<Queue>
+getNrClientsInTheQueues(): int
+toString(): String

The class Scheduler holds the application's Queues in a list. It allocates the clients to the queue with the smallest waiting time at that given moment. The constructor of the class creates nrQueues empty queues and adds them to the list.

The allocateClientToQueue method iterates through the queues and finds the one with the minimum waiting time, and calls that queue's addClient method with the client that it received as paramether.

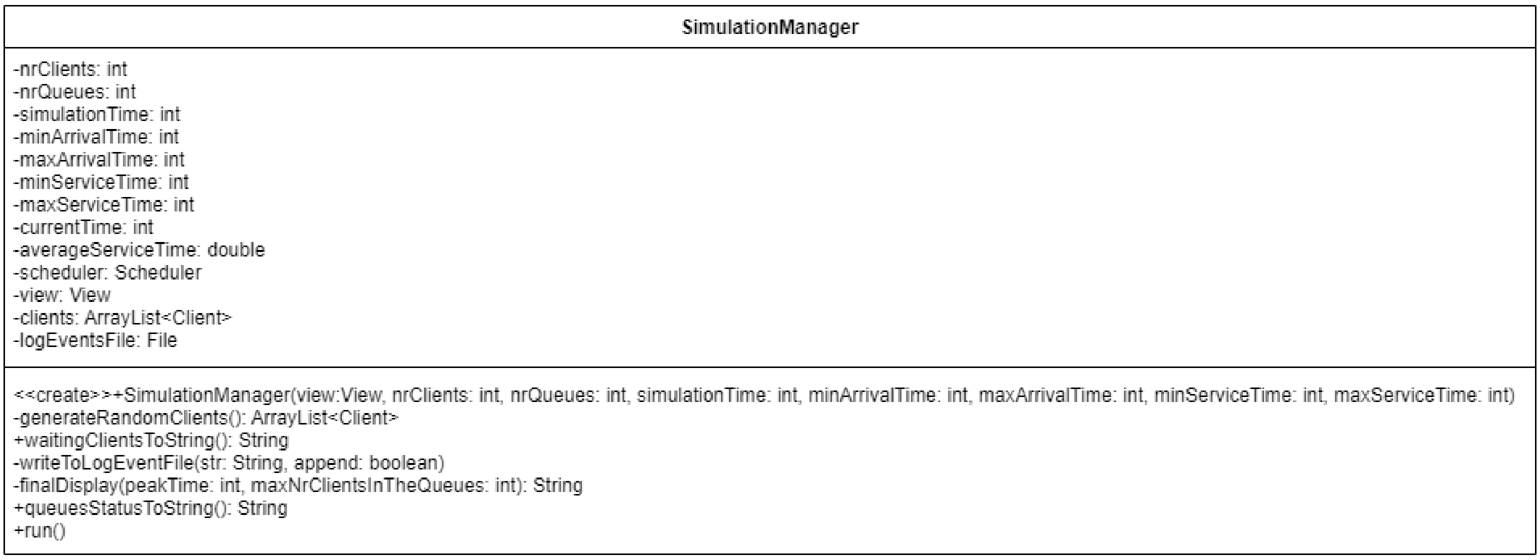
```
public void allocateClientToQueue(Client client)
{
    int i=0,min=0;
    while(i<nrQueues)
    {

if(queues.get(i).getWaitingPeriod().get()<queues.get(min).getWaitingP
eriod().get())
```

```
                min=i;
            i++;
        }
        queues.get(min).addClient(client);
    }
```

The getNrClientsInTheQueues method computes the total number of clients that are in the queues at the current moment. This will be used in SimulationManager to compute the peak hour.

```
public int getNrClientsInTheQueues()
{
    Iterator<Queue> iterator = queues.iterator();
    int sum=0;
    while(iterator.hasNext())
    {
        sum += iterator.next().getNrClientsInTheQueue();
    }
    return sum;
}
```

4.1.4 Class SimulationManager



SimulationManager

-nrClients: int
-nrQueues: int
-simulationTime: int
-minArrivalTime: int
-maxArrivalTime: int
-minServiceTime: int
-maxServiceTime: int
-currentTime: int
-averageServiceTime: double
-scheduler: Scheduler
-view: View
-clients: ArrayList<Client>
-logEventsFile: File

<<create>>+SimulationManager(view:View, nrClients: int, nrQueues: int, simulationTime: int, minArrivalTime: int, maxArrivalTime: int, minServiceTime: int, maxServiceTime: int)
-generateRandomClients(): ArrayList<Client>
+waitingClientsToString(): String
-writeToLogEventFile(str: String, append: boolean)
-finalDisplay(peakTime: int, maxNrClientsInTheQueues: int): String
+queuesStatusToString(): String
+run()

The class SimulationManager implements the Runnable interface. It holds the information given by the user in the user interface (i.e. nrClients, nrQueues, simulationTime, minArrivalTime, maxArrivalTime, minServiceTime, maxServiceTime) and the View of the application. The logEventsFile is the file where the log of events will be written during the simulation.

In the constructor the currentTime field is set to 0, a new Scheduler object is created (which creates the queues), and nrClients random clients are created using the generateRandomClients private method.

The method generates random numbers in the input intervals and adds the newly created clients to the clients list. It also computes the averageServiceTime.

```
private ArrayList<Client> generateRandomClients()
{
    ArrayList<Client> clientList=new ArrayList<Client>();
    Random random = new Random();
    int arriveTime, serviceTime;
    averageServiceTime=0;
    for(int i=1;i<=nrClients;i++)
    {
        arriveTime= random.nextInt(maxArrivalTime+1-
minArrivalTime)+minArrivalTime;
        serviceTime= random.nextInt(maxServiceTime+1-
minServiceTime)+minServiceTime;
        averageServiceTime+=serviceTime;
        clientList.add(new Client(i,arriveTime,serviceTime));
    }
    averageServiceTime=averageServiceTime/nrClients;
    return clientList;
}
```

The writeToLogEventFile method writes the string given as parameter in the logEventsFile. If append is true it appends the string to the end of the file, otherwise it deletes the existing content of the file and writes the string instead.

```
private void writeToLogEventFile(String str, boolean append)
{
    try
    {
        FileWriter fw = new FileWriter(logEventsFile,append);
        PrintWriter pw = new PrintWriter(fw);
        pw.append(str);
        pw.close();
    }
    catch (IOException e)
    {
    }
}
```

The finalDisplay method creates the String containing the final results of the simulation (with the average waiting time, average service time and peak hour). It displays the double numbers with a maximum of 2 decimal places.

```
private String finalDisplay(int peakTime, int
maxNrClientsInTheQueues)
{
    ArrayList<Queue> queues = scheduler.getQueues();
    String str = waitingClientsToString();  str=str+ "\nAverage
waiting time: ";
    double avrNumber=0;
    Iterator<Queue> iterator = queues.iterator();
    while(iterator.hasNext())
        avrNumber+=iterator.next().getSumOfWaitingTimes();
    avrNumber=avrNumber/nrClients;
    String avr=""+avrNumber;
    int point=avr.indexOf('.');
    if(avrNumber == (int)avrNumber)
        avr=avr.substring(0,point);
    else if(avr.length()>point+3)
        avr=avr.substring(0,point+3);
    str = str + avr + "\nAverage service time: ";
    String avrS=""+ averageServiceTime;
    point=avrS.indexOf('.');
    if(averageServiceTime == (int)averageServiceTime)
        avrS=avrS.substring(0,point);
    else if(avrS.length()>point+3)
        avrS=avrS.substring(0,point+3);
    str = str + avrS + "\nPeak hour: "+peakTime+"\nClients in the 
queues at peak hour: "+maxNrClientsInTheQueues;
    view.clientsWaitingDisplay.setText(str);
    return str;
}
```

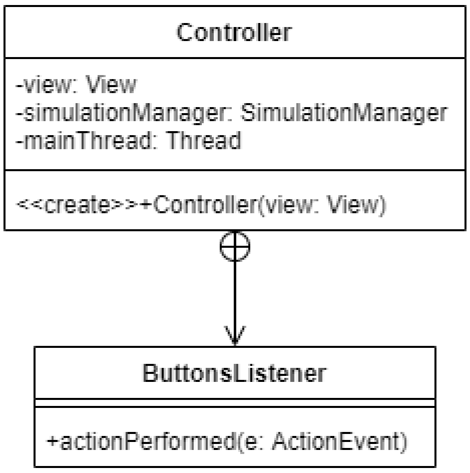
The run method starts the threads of all the queues and clears the logEventsFile. Then while the current time is smaller than the simulation time given by the user through the user interface, it takes all the clients from the clients list and sends to the Scheduler the ones with the arrival time equal to the current time to be assigned to queues, then it deletes them from its list of clients. The method then displays the current status in the user interface (view) and to the logEventsFile, increases the current time, and sleeps for 1 second. During this it saves the peak time and the maximum number of clients that were in the queues at peak time.

When the current time has reached the simulation time, the method displays the final results then ends.

```
@Override
public void run() {
    ArrayList<Queue> queues = scheduler.getQueues(); Iterator<Queue> 
iterator = queues.iterator();
    while(iterator.hasNext()) new Thread(iterator.next()).start();
    Iterator<Client> clientIterator; Client client; int peakTime=-1, 
maxNrClientsInTheQueues=0; String waitingClients, queuesString;
    writeToLogEventFile("", false);
    while(currentTime<simulationTime) {
        clientIterator=clients.iterator();
        while(clientIterator.hasNext()) {
            client=clientIterator.next();
            if(client.getArrivalTime()==currentTime) {
                scheduler.allocateClientToQueue(client);
                clientIterator.remove(); } }
        waitingClients=waitingClientsToString();
        queuesString=scheduler.toString();
        view.clientsWaitingDisplay.setText(waitingClients);
        view.queueDisplay.setText(queuesString);

writeToLogEventFile(waitingClients+"\n"+queuesString+"\n",true);
        int nrClientsInTheQueues=
scheduler.getNrClientsInTheQueues();
        if(nrClientsInTheQueues>maxNrClientsInTheQueues) {
            maxNrClientsInTheQueues=nrClientsInTheQueues;
            peakTime=currentTime;
        }
        currentTime++;
        try { sleep(1000); }
```

```
            catch (InterruptedException ex) { }
        }
        String str=finalDisplay(peakTime,maxNrClientsInTheQueues);
        view.clientsWaitingDisplay.setText(str);
        writeToLogEventFile(str,true);
    }
```

4.1.5 Class Controller


Controller
-view: View
-simulationManager: SimulationManager
-mainThread: Thread
<<create>>+Controller(view: View)
ButtonsListener
+actionPerformed(e: ActionEvent)

The class Controller takes the input from the view, adds action listeners to the view's buttons, displays messages and holds the application's SimulationManager and main Thread.

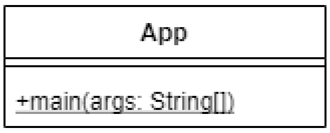
4.1.6 Class ButtonsListener

The class ButtonListener implements the ActionListener interface and is an inner class of the Controller class. In the method actionPerformed, it validates the input received from the view, displays error messages, creates the SimulationManager and starts the main thread of the application.

```
@Override
public void actionPerformed(ActionEvent e) {
    if(e.getSource()==view.buttons.get(0)){ //save
        int input[] = new int[7]; int i=0;
        try {
            while(i<7) { //read from user interface the 7 fields

input[i]=Integer.parseInt(view.fields.get(i).getText());
                view.setWrongLabelVisible(false,i);
                i++; }
            if(input[4]<input[3]) {
                view.setWrongLabelVisible(true,3);
                view.setWrongLabelVisible(true,4);
                i=8; }
            if(input[6]<input[5]) {
                view.setWrongLabelVisible(true,6);
                view.setWrongLabelVisible(true,5);
                i=8; } } catch(NumberFormatException ex) {
view.setWrongLabelVisible(true,i); }
        if(i==7){ //all ok
            if(mainThread!=null && mainThread.isAlive())
                view.setWrongLabelVisible(true,7);
            else {
                view.setWrongLabelVisible(false,7);
                simulationManager=new SimulationManager(view,
input[0], input[1], input[2], input[3], input[4], input[5],
input[6]);

view.clientsWaitingDisplay.setText(simulationManager.waitingClientsTo
String());

view.queueDisplay.setText(simulationManager.queuesStatusToString());
} }
```

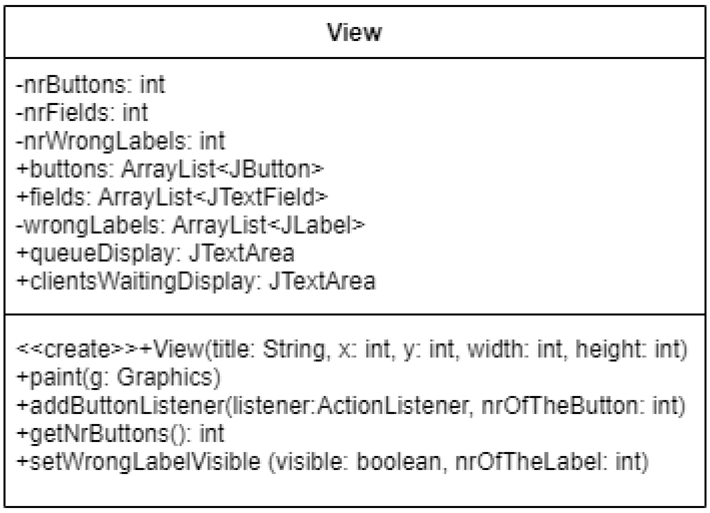
```
            view.revalidate(); view.repaint(); }
        else if(e.getSource()==view.buttons.get(1)){ //start
            if(mainThread!=null && mainThread.isAlive())
    view.setWrongLabelVisible(true,7); //there is a thread running
            else {
                view.setWrongLabelVisible(false,7);
                mainThread= new Thread(simulationManager);
                mainThread.start(); } view.revalidate(); view.repaint();
    } }
```

4.1.7 Class App



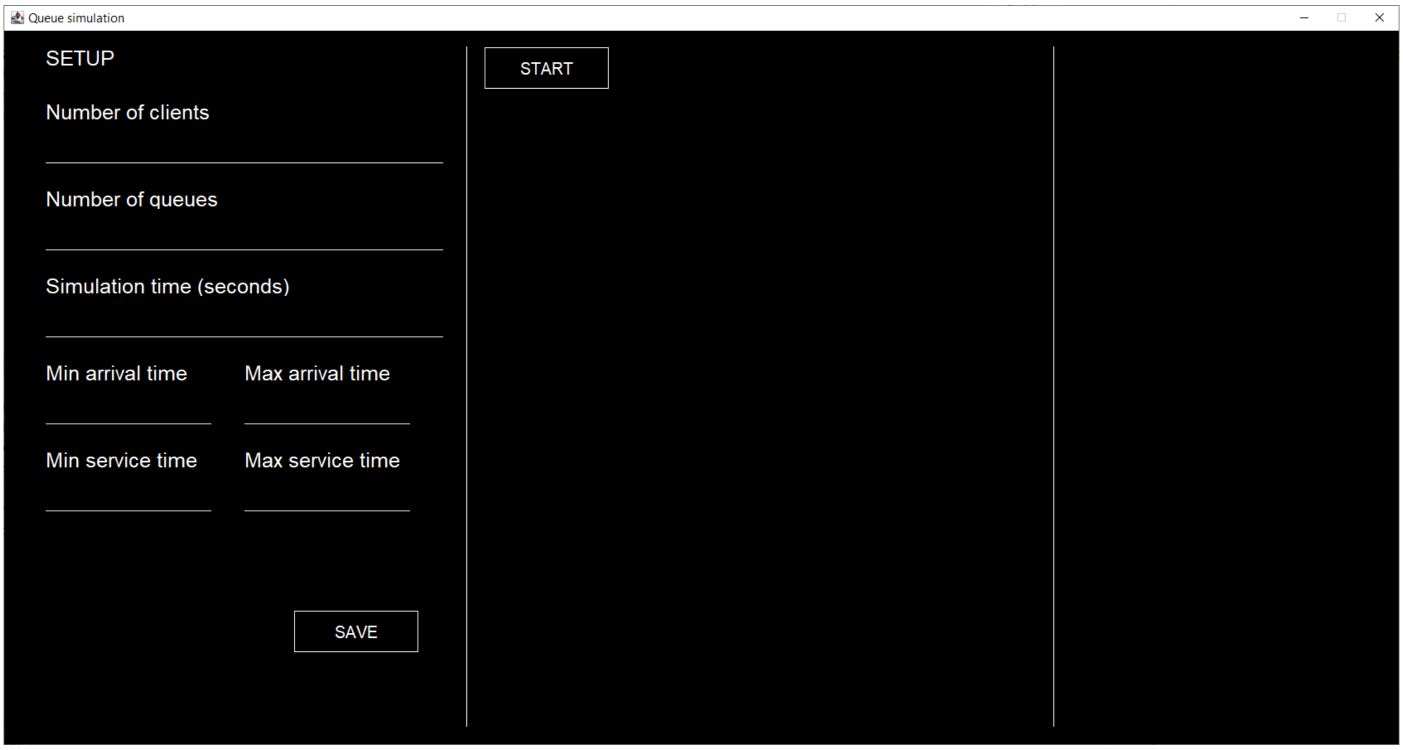
The App class contains the main method which Creates the View and the Controller objects.

4.1.8 Class View


View
-nrButtons: int
-nrFields: int
-nrWrongLabels: int
+buttons: ArrayList<JButton>
+fields: ArrayList<JTextField>
-wrongLabels: ArrayList<JLabel>
+queueDisplay: JTextArea
+clientsWaitingDisplay: JTextArea
<<create>>+View(title: String, x: int, y: int, width: int, height: int)
+paint(g: Graphics)
+addButtonListener(listener:ActionListener, nrOfTheButton: int)
+getNrButtons(): int
+setWrongLabelVisible (visible: boolean, nrOfTheLabel: int)

The class View creates the GUI of the application. The queueDisplay is the area where the queues status will be displayed, and clientsWaitingDisplay is the area where the clients that aren't yet in any queue will be displayed.
The wrongLabels are used to display specific error messages.

4.2 GUI description


Queue simulation

SETUP

Number of clients

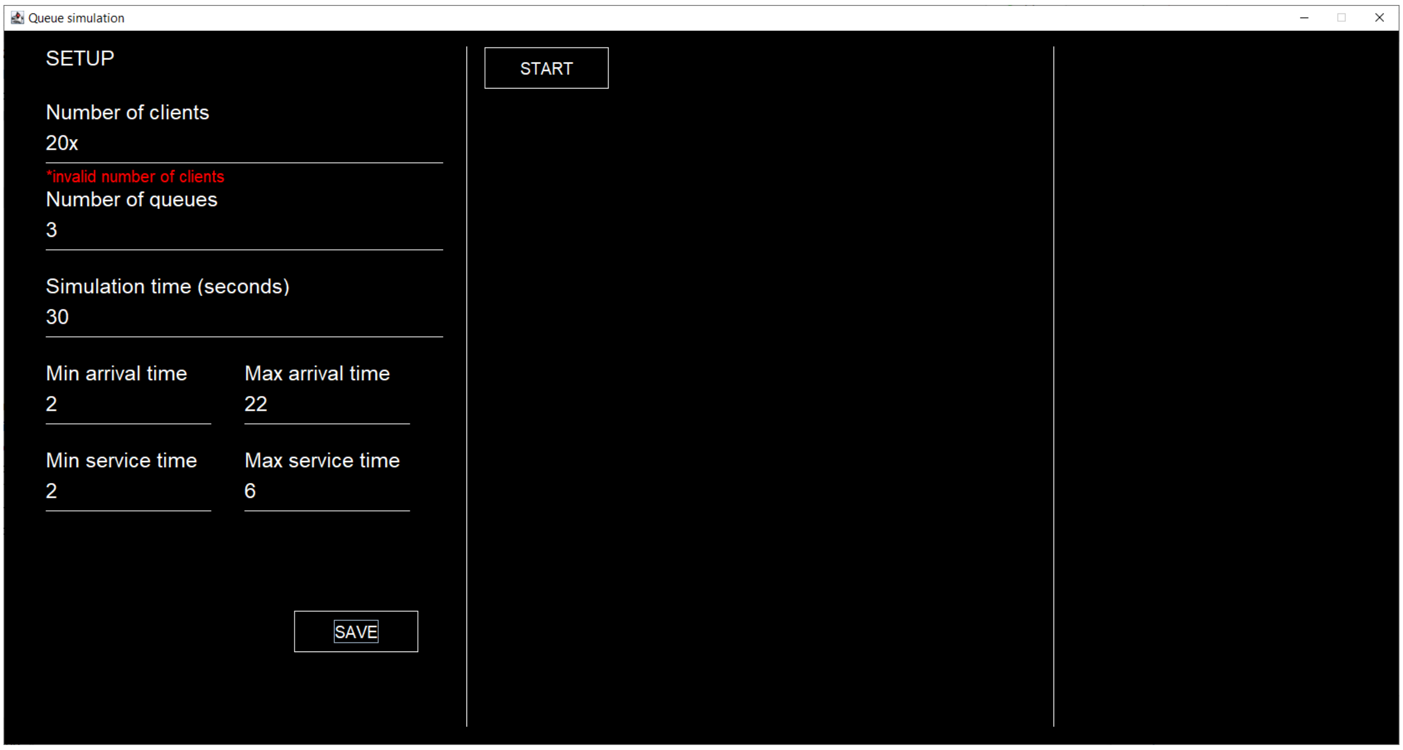
Number of queues

Simulation time (seconds)

Min arrival time | Max arrival time

Min service time | Max service time

The graphical user interface has 7 field where the user has to insert the data, and 2 areas used for displaying the status of the simulation.


Queue simulation

SETUP

Number of clients
20x

*invalid number of clients

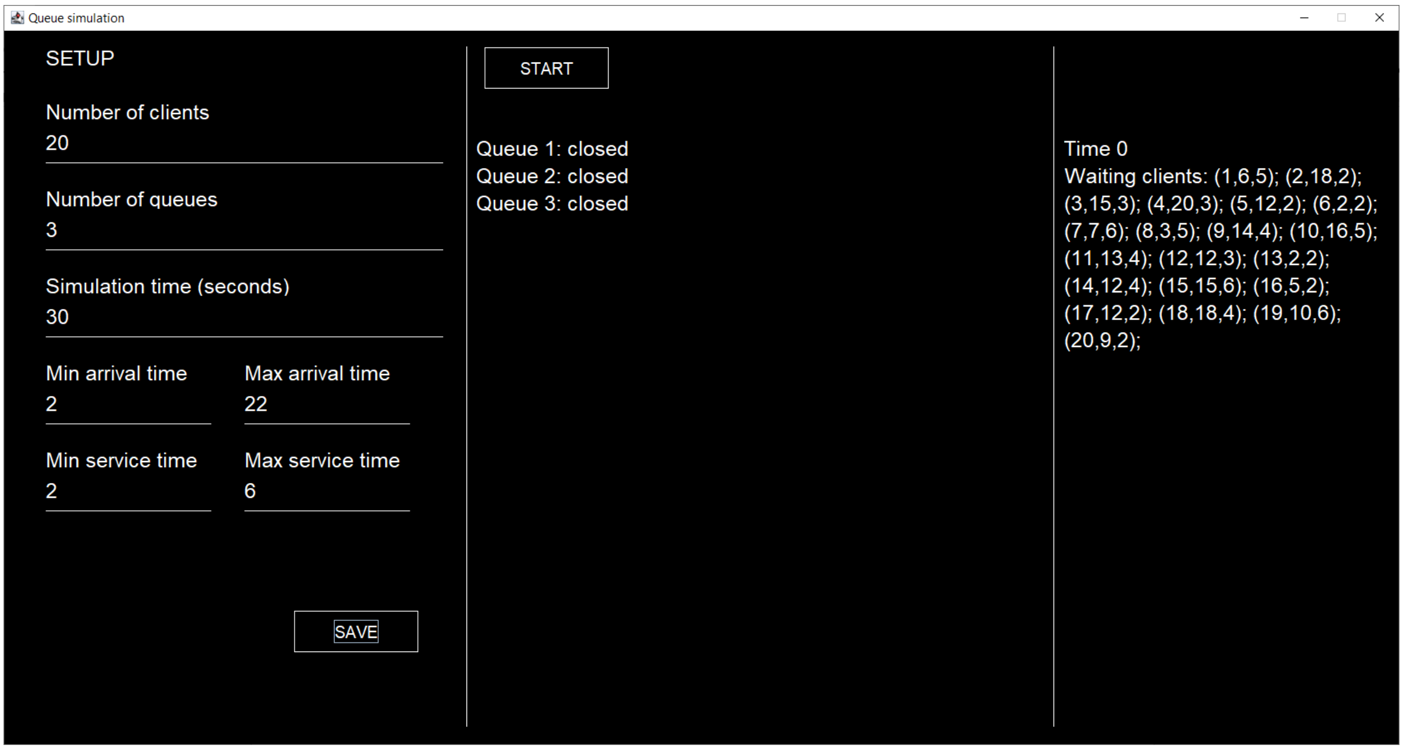
Number of queues
3

Simulation time (seconds)
30

Min arrival time: 2 | Max arrival time: 22

Min service time: 2 | Max service time: 6

In case of a wrong input an error message is displayed.



Number of clients
20

Number of queues
3

Simulation time (seconds)
30

Min arrival time
2

Max arrival time
22

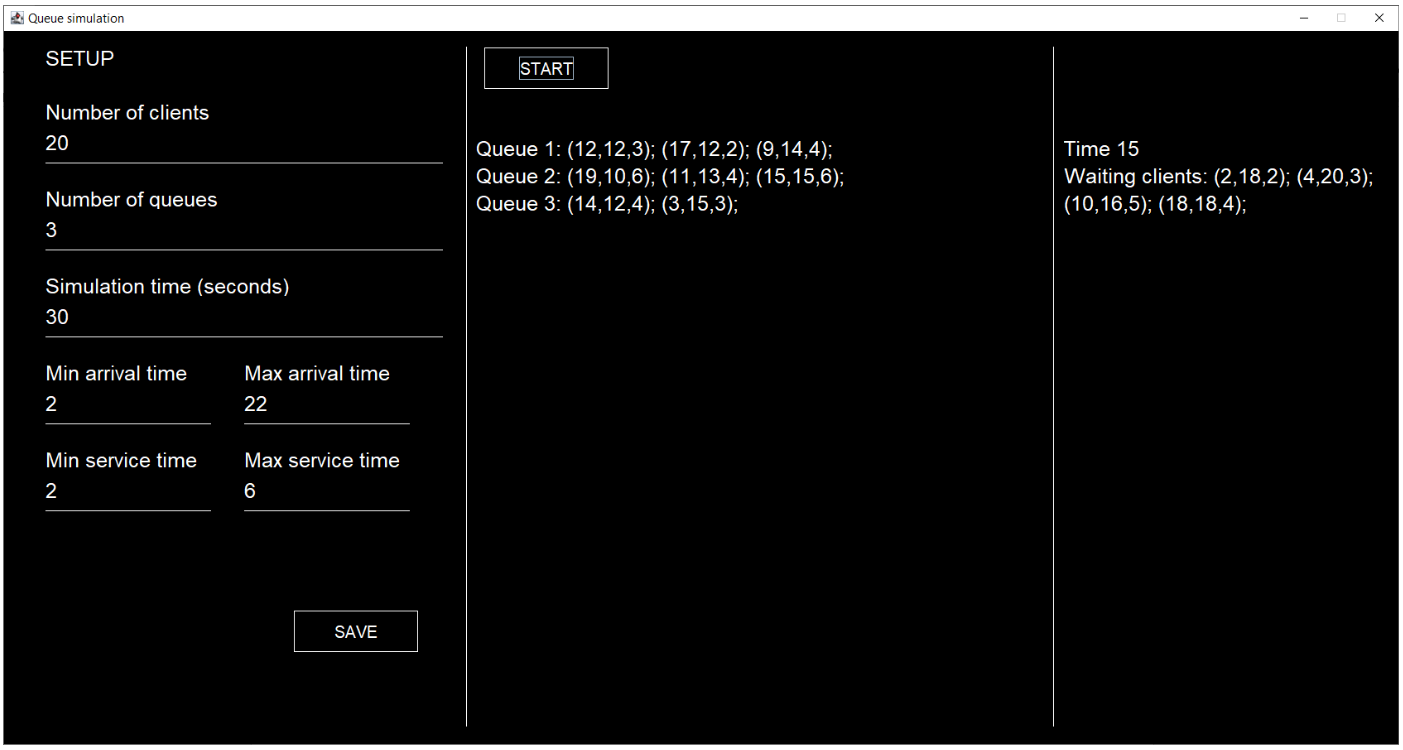
Min service time
2

Max service time
6

```
Queue 1: closed
Queue 2: closed
Queue 3: closed
```

```
Time 0
Waiting clients: (1,6,5); (2,18,2);
(3,15,3); (4,20,3); (5,12,2); (6,2,2);
(7,7,6); (8,3,5); (9,14,4); (10,16,5);
(11,13,4); (12,12,3); (13,2,2);
(14,12,4); (15,15,6); (16,5,2);
(17,12,2); (18,18,4); (19,10,6);
(20,9,2);
```

If the input is correct, when the SAVE button is pressed, the initial status of the simulation is displayed.



Number of clients
20

Number of queues
3

Simulation time (seconds)
30

Min arrival time
2

Max arrival time
22

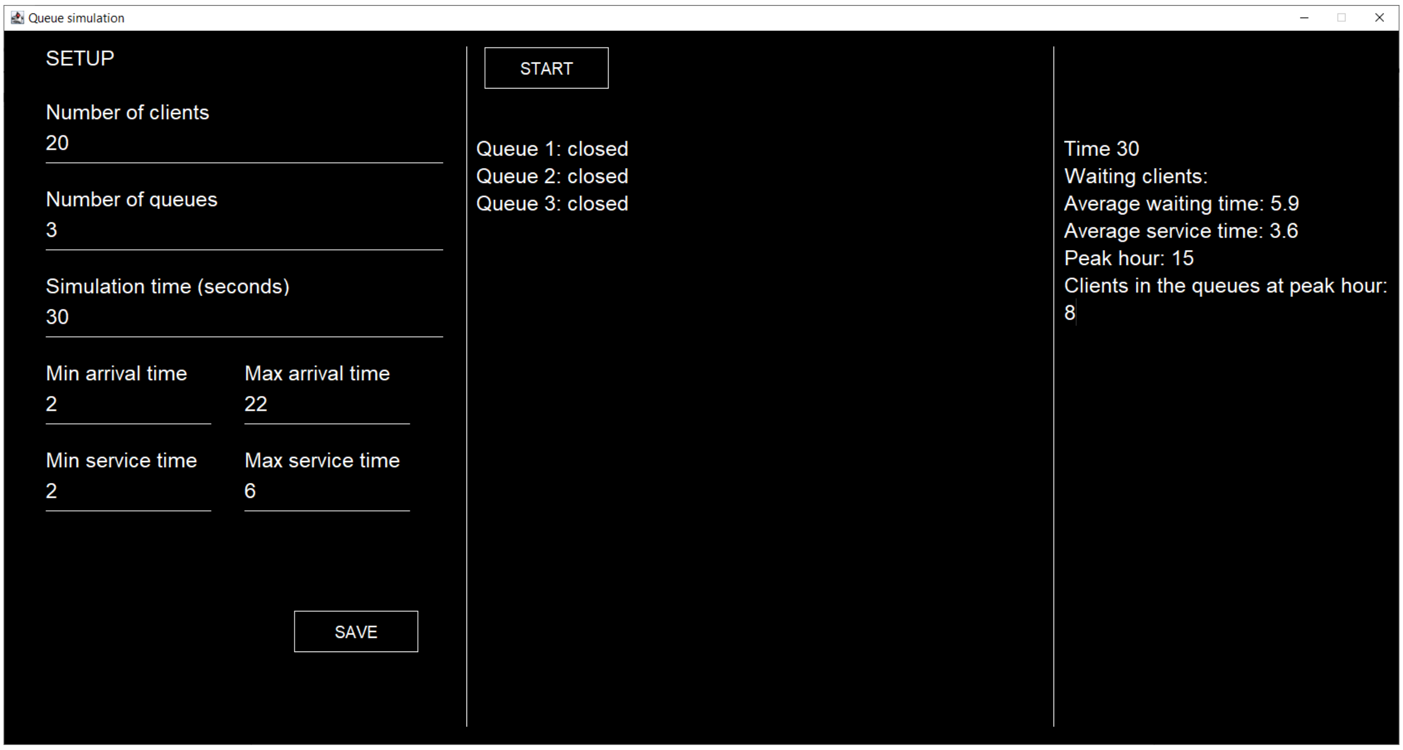
Min service time
2

Max service time
6

```
Queue 1: (12,12,3); (17,12,2); (9,14,4);
Queue 2: (19,10,6); (11,13,4); (15,15,6);
Queue 3: (14,12,4); (3,15,3);
```

```
Time 15
Waiting clients: (2,18,2); (4,20,3);
(10,16,5); (18,18,4);
```

When the START button is pressed the simulation begins and the real time evolution is displayed every second.


Queue simulation

SETUP

Number of clients
20

Number of queues
3

Simulation time (seconds)
30

Min arrival time
2

Max arrival time
22

Min service time
2

Max service time
6

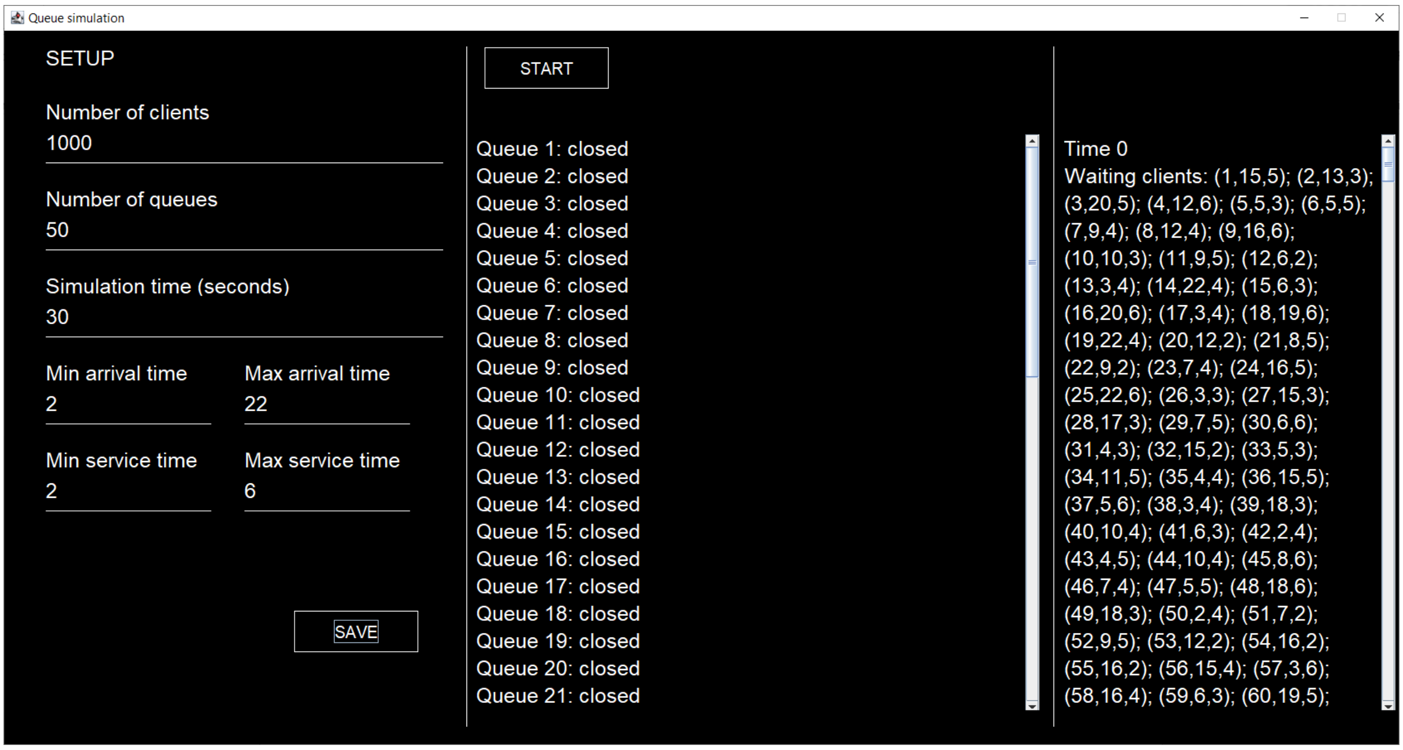
SAVE

START

```
Queue 1: closed
Queue 2: closed
Queue 3: closed
```

```
Time 30
Waiting clients:
Average waiting time: 5.9
Average service time: 3.6
Peak hour: 15
Clients in the queues at peak hour:
8
```

The final results are displayed after the simulation is finished (i.e. average waiting time, average service time and peak hour).


Queue simulation

SETUP

Number of clients
1000

Number of queues
50

Simulation time (seconds)
30

Min arrival time
2

Max arrival time
22

Min service time
2

Max service time
6

SAVE

START

```
Queue 1: closed
Queue 2: closed
Queue 3: closed
Queue 4: closed
Queue 5: closed
Queue 6: closed
Queue 7: closed
Queue 8: closed
Queue 9: closed
Queue 10: closed
Queue 11: closed
Queue 12: closed
Queue 13: closed
Queue 14: closed
Queue 15: closed
Queue 16: closed
Queue 17: closed
Queue 18: closed
Queue 19: closed
Queue 20: closed
Queue 21: closed
```

```
Time 0
Waiting clients: (1,15,5); (2,13,3);
(3,20,5); (4,12,6); (5,5,3); (6,5,5);
(7,9,4); (8,12,4); (9,16,6);
(10,10,3); (11,9,5); (12,6,2);
(13,3,4); (14,22,4); (15,6,3);
(16,20,6); (17,3,4); (18,19,6);
(19,22,4); (20,12,2); (21,8,5);
(22,9,2); (23,7,4); (24,16,5);
(25,22,6); (26,3,3); (27,15,3);
(28,17,3); (29,7,5); (30,6,6);
(31,4,3); (32,15,2); (33,5,3);
(34,11,5); (35,4,4); (36,15,5);
(37,5,6); (38,3,4); (39,18,3);
(40,10,4); (41,6,3); (42,2,4);
(43,4,5); (44,10,4); (45,8,6);
(46,7,4); (47,5,5); (48,18,6);
(49,18,3); (50,2,4); (51,7,2);
(52,9,5); (53,12,2); (54,16,2);
(55,16,2); (56,15,4); (57,3,6);
(58,16,4); (59,6,3); (60,19,5);
```

In case the data doesn’t fit in the window the areas can expand.

5. CONCLUSIONS

5.1 What I have learned

- How to use threads
- How to make threads sharing data safe

5.2 Future development

- The application could have a more friendly user interface with images or icons.
- The option to pause and resume or quit a simulation could be implemented.
- The application could be transformed into a game where the user can open and close queues, the clients could keep coming, they could give a rating to the store based on how much time they had to wait or they could leave the queue angry and don't buy their stuff anymore if the time they waited is too long, and the application could keep track of the profit you make when people buy things.

6. BIBLIOGRAPHY

- https://stackoverflow.com/questions/13746298/java-line-drawn-disappeared-in-some-points?fbclid=IwAR0seACw2zbDzM7BnJQfuYY-XUfwJnNPzhFNkTYZwxbkO2pQPfSZbQyQeDw
- https://www.geeksforgeeks.org/arraylist-get-method-java-examples/
- https://stackoverflow.com/questions/15995458/how-to-find-the-minimum-value-in-an-arraylist-along-with-the-index-number-jav?fbclid=IwAR1Q9NWZDDXsiU4P8vTb9ePJxMxRrNooA57IVc7ps1cCTg9AUFQpWNeH5X0
- https://stackoverflow.com/questions/42690425/jtextarea-border-in-java-swing?fbclid=IwAR2coXUGE3uP3faWLuAMc_kMm4kUp2XIel0de-y7jTyzXK9RrLlSAsBDW2c
- https://www.baeldung.com/java-generating-random-numbers-in-range
- https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/LinkedBlockingQueue.html?fbclid=IwAR0ixt8OGuXyfNISgCd0wo7wB9v6b6DwIV6u75GrNHdia9_cr1wwRRWQnz4
- https://www.geeksforgeeks.org/atomicinteger-addandget-method-in-java-with-examples/
- https://books.trinket.io/thinkjava/appendix-b.html?fbclid=IwAR1Bv4vKvxJRvByVRNjqS1st9To6K0Arm6zxL8Zl1frobkafMw7N7lm5pWU