

Laboratory Work - Homework 3

Name: Bologa Marius Vasile
Group: 30425

1. Project objectives

a) Objective

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

b) 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 "client" to wait before receiving a "service". The management of queue based systems is interested in minimizing the time amount its "clients" 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 clients 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 clients 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 other clients in the queue and their service needs.

Input data:

- Minimum and maximum interval of arriving time between clients;
- 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 , modeling scenarios , use cases

Analysis and Modeling

A few words about threads: Java threads are objects like any other Java objects. Threads are instances of class `java.lang.Thread`, or instances of subclasses of this class. In addition to being objects, java threads can also execute code. There are 2 ways in which you can declare a thread:

1. The first way to specify what code a thread is to run, is to create a subclass of `Thread` and override the `run()` method. The `run()` method is what is executed by the thread after you call `start()`. Here is an example of creating a Java Thread subclass:

2. The second way to specify what code a thread should run is by creating a class that implements `java.lang.Runnable`. The `Runnable` object can be executed by a `Thread`.

The application consists of 3 packages: views, controller, models. All packages contain a number of classes that model the given problem. Such that in the package models is the class **Task** represents the task in the application and contains all attributes related to a task (such as arrival time, waiting time, service time etc.). The class `Server` is responsible for the representation of queues and contains a `ArrayBlockingQueue` of tasks. The class **TaskScheduler** is the class that is responsible for the distribution of clients in queues, as well as giving the time periods for the simulation. The class `TaskGenerator`, generates tasks with all the needed information.

Scenarios:

Scenarios consist of modeling the behaviour of the application in a real-life situation. I will exemplify a scenario that tests the functionality of the application. We suppose the user inputs correct information.

The user would like to simulate a situation in which 3 queues are active and the simulation lasts 40 seconds. He inputs the corresponding parameters into their respective fields, set an interval for service time and an interval for arriving time between customers and presses **START SIMULATION**.

The application processes data and constructs the queues. Tasks are being added and removed from the server, taking into consideration the values for service time and arriving time between each task.

The user views the results of the simulation: the first tab shows the final state of the system (using graphics) and the second tab shows the dynamic evolution of the system as well as the values outputted at the end of every simulation.

The application can be used to simulate a real-life situation in one of the crowded supermarkets (it is quite hypothetical, since nowadays, no supermarkets use all of their available cash registers).

There are a few cases in which the application fails to perform any actions. These cases consist of, but are not restricted to:

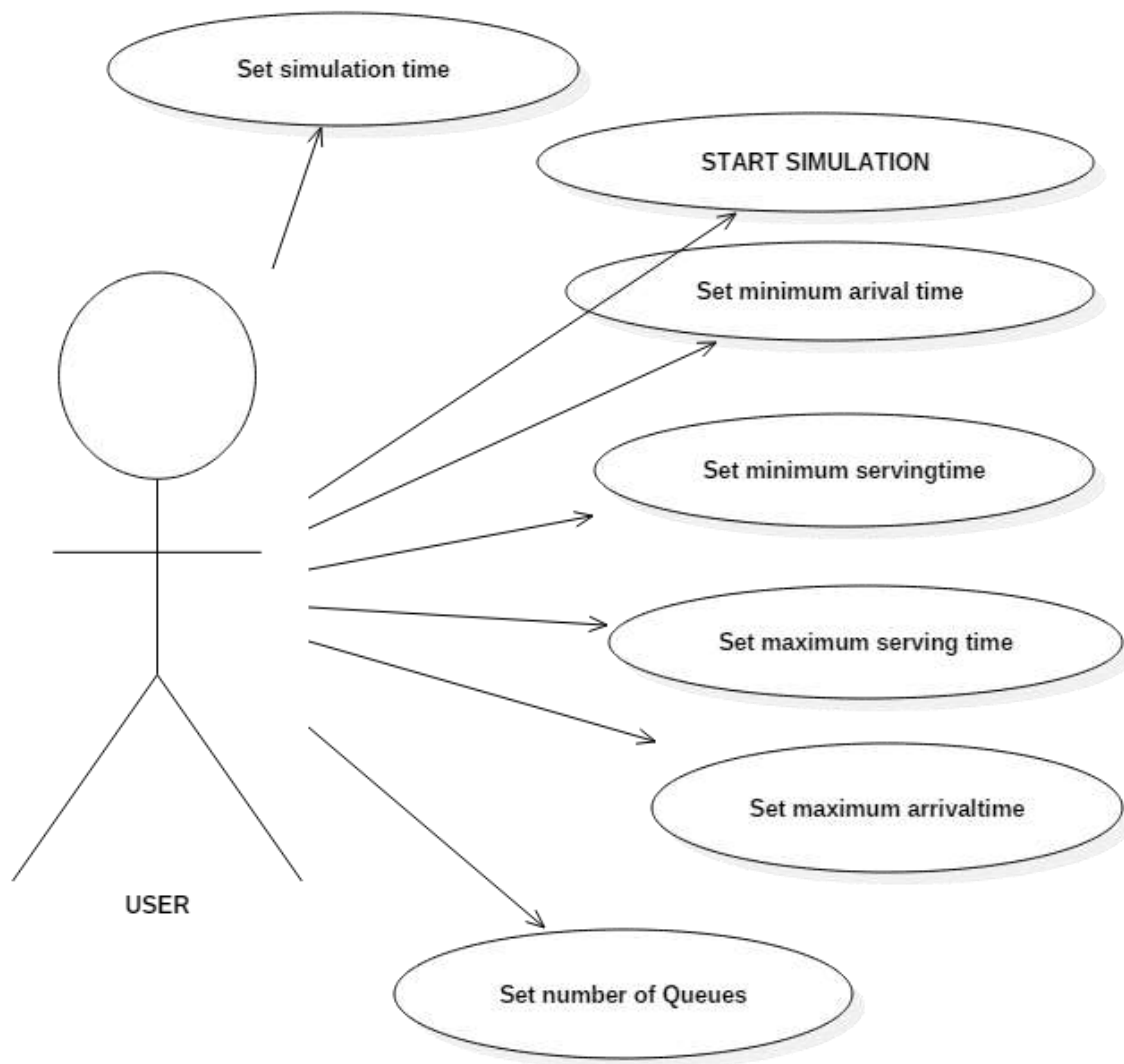
- User inserting a non-number character anywhere as input
- User doesn't input all the necessary values

Sometimes the log behaves a bit odd: when multiple lines of text need to be displayed simultaneously, a line of text is sometimes displayed at the very beginning of the log instead of where it's supposed to be. If the user input incorrect information, there will be shown specific error messages, and the application will not run.

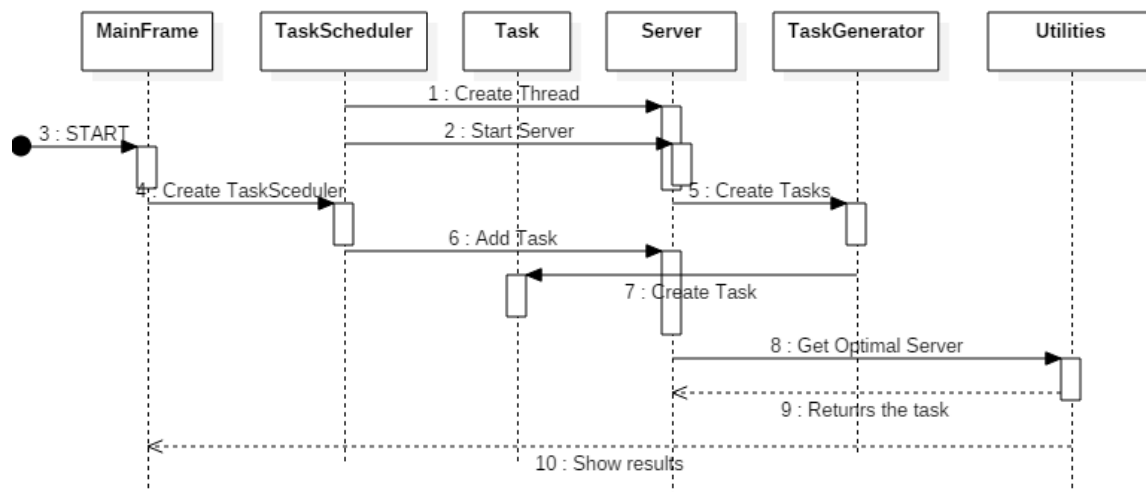
3. Projection and Implementation

3.1 UML diagrams

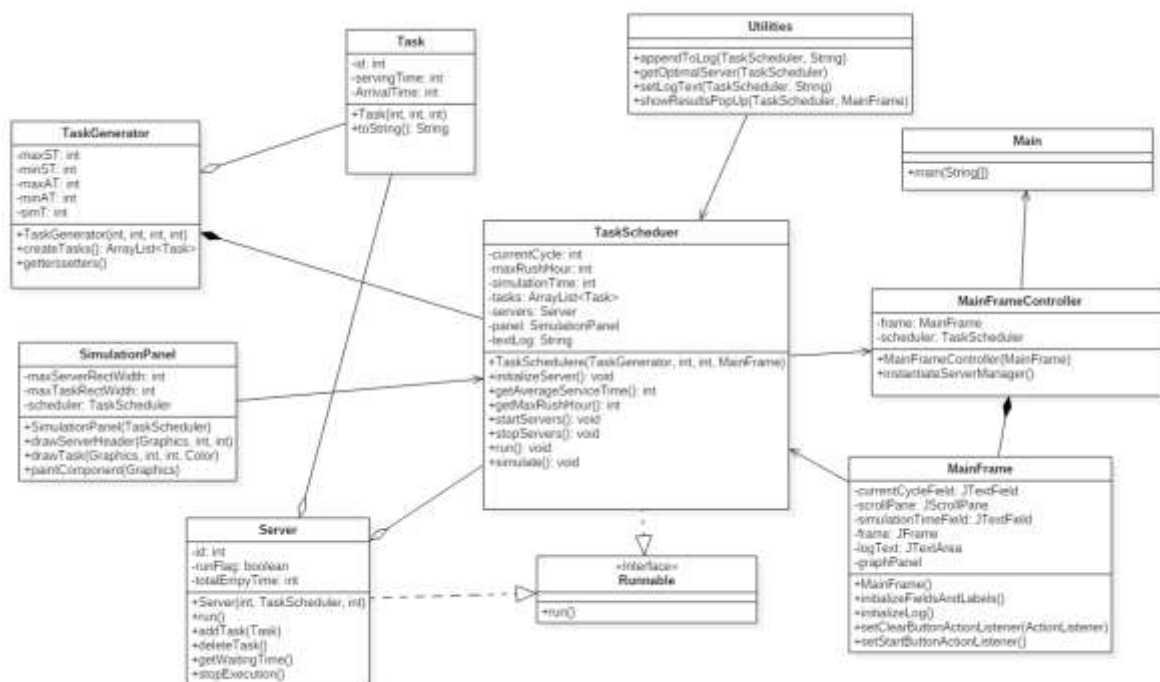
3.1.1 Use case diagram



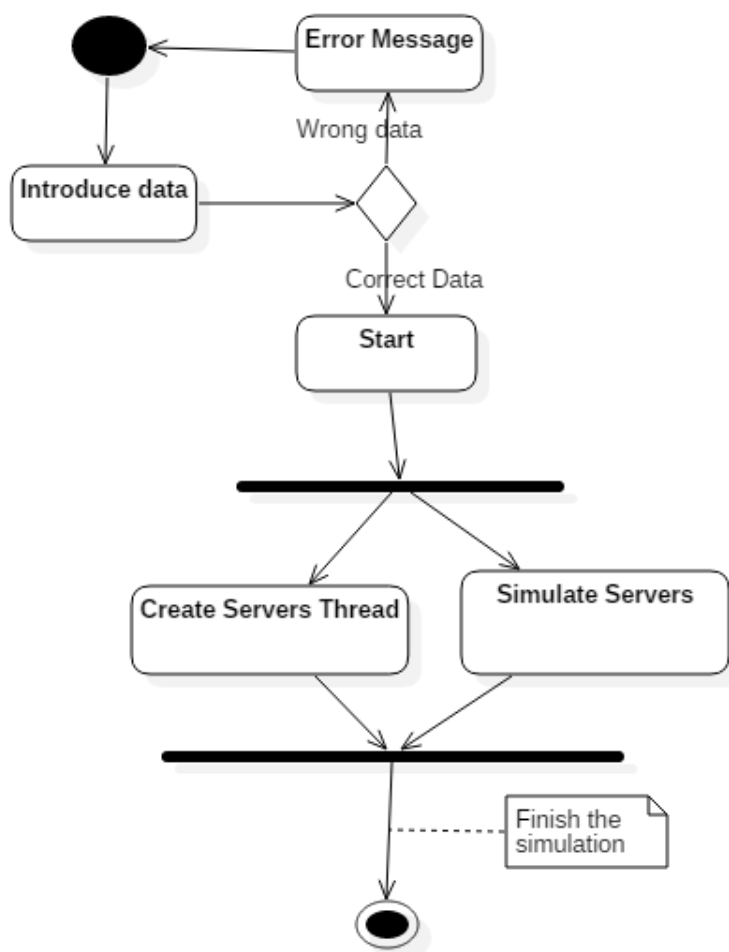
3.1.2 Sequence diagram



3.1.3 Class diagram



3.1.4 Activity diagram



3.2 Data structures

In computer science, a queue is a particular kind of abstract data type or collection in which the entities in the collection are kept in order and the principal (or only) operations on the collection are the addition of entities to the rear terminal position, known as *enqueue*, and removal of entities from the front terminal position, known as *dequeue*. This makes the queue a First-In-First-Out (FIFO) data structure.

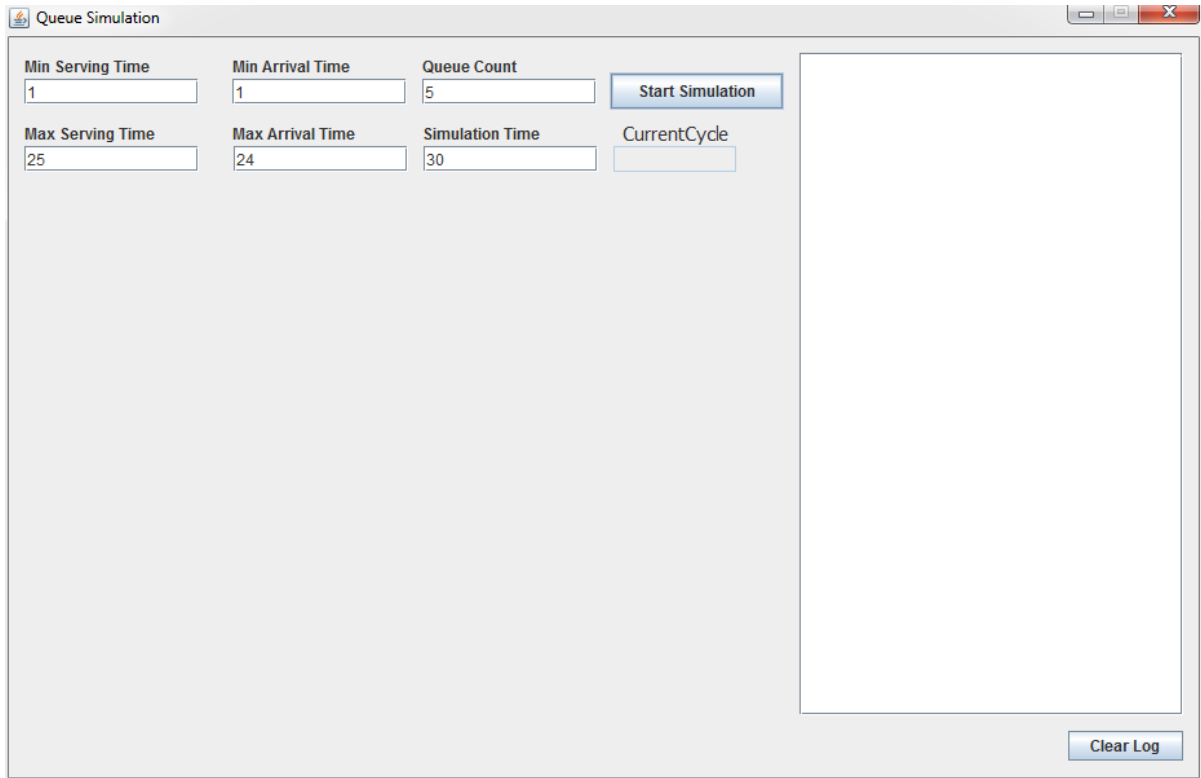
The data structures used at this problem are either primitive data types such as integers or floats or more complex objects such as array of servers or ArrayLists and ArrayBlockingQueue.

The ArrayList is used to store the newly created tasks. For implementing the servers I have used ArrayBlockingQueue. ArrayBlockingQueue is a bounded, blocking queue that stores the elements internally in an array. That it is bounded means that it cannot store unlimited amounts of elements. There is an upper bound on the number of elements it can store at the same time. You set the upper bound at instantiation time, and after that it cannot be changed. I have chosen to use this data structure because it is thread safe and it is easy to simulate the FIFO concept of the queue. I have used the predefined functions peek() to get the first element in the queue and also put() to add a new task to the queue.

3.3 Class projection

Class projection refers mainly to how the model was thought, how the problem was divided in sub-problems, each sub-problem representing more or less the introduction of a new class. First I will start by mentioning exactly how my problem was divided into packages and afterwards each package with its own classes. I begin by creating the three packages I used: the first one being called „views” the second one being called „models”, and the third one „controllers”. I named them intuitively because the first one handles the interface part, the part that deals with the user, the second one handles the implementation part, the third package is the one that provides useful functions, in order to make the algorithms in a modularized way and the fourth one, is the one that contains the main class, and the controller of the interface. I will begin with the interface, then I will continue with the models and controller part.

3.4 Interface(view) **MainFrame.java**



- extends JFrame class , with which we made the GUI of the application
- some elements of the interface are: labels , text fields, buttons and a scroll panel with a text area for printing the each step of the simulation
- private** JTextArea **logText**-used for printing out the log of events and main system data
- private** JTextField **currentCycleField**-used for displaying the current cycle of the simulation
- private** JTextField **queueCountField**-used as an input for getting the number of queues needed for simulation
- private** JTextField **simulationTimeField**- used as an input for getting the simulation time of the application
- private** JButton **startButton**- used for running the application
- private** JButton **btnClearLog** – used to delete the information from the text area
- private** JPanel **graphPanel** – used to present the actual simulation of the queue

public void initializeFieldsAndLabels() –method used to instantiate all the field and the labels from the frame

public void setGraphPanel(SimulationPanel **gPanel**)- method used to instantiate the simulation panel, where will be displayed the evolution of the simulation

public void initializeLog() - method used to initialize the text area, and the log

Also the class contains getters and setters for all the fields and components.

SimulationPanel.java

This class is represents the way in which the simulation is made and shown to the user.

The class extends JPanel, and this panel is later added to the main frame. The constructor of this class is presented below:

```
public SimulationPanel(TaskScheduler scheduler) {
    this.scheduler = scheduler;
```

}

public void drawServerHeader(Graphics g, **int** x, **int** y)- this method draws the actual representation of the server, as a gray rectangle.

public void drawTask(Graphics g, **int** x, **int** y, Color c)- this method draws the actual representation of the task, as a rectangle but with different color, differentiated by their service time.

public void paintComponent(Graphics g)- this method is the most important one in this class, and takes all the tasks from each server and draws it into the corresponding place and gives a specific color to the task, depending on their service time.

3.5 Models

Task.java

As I have already mentioned above, this is the class responsible for the modeling of the task in the application. It contains all information that the system needs to know about the task (arrival time, service time, client number etc) and contains basic methods to return or set these values (**getters** and **setters**). This is the class where the method **toString()** is being overridden. This method is used to display the characteristic of each individual client into the dynamic activity log in the GUI. We need to override this method, because it is a method of class Object that is extended by every other class.

private int servingTime = 0;

private int arrivalTime = 0;

private final int taskID;

The constructor of the class:

```
public Task(int taskID, int servingTime, int arrivalTime) {
    this.servingTime = servingTime;
    this.arrivalTime = arrivalTime;
    this.taskID = taskID;
```

}

Utilities.java

The methods that are in this class are used in order to make the work with threads and the computation of the operations made on threads easier. Also this class is used to show the information that is required at the end.

public static void showResultsPopUp(TaskScheduler scheduler, MainFrame f) – this method is used for showing up the results at the end, average waiting time, empty time for each servers, serving time for each server etc.

public static void setLogText(TaskScheduler scheduler, String s)
 synchronized (scheduler.textLog) {}

-this method is used to put in the text area the information that are required

First, it is not possible for two invocations of synchronized methods on the same object to interleave. When one thread is executing a synchronized method for an object, all other

threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object.

Second, when a synchronized method exits, it automatically establishes a happens-before relationship with *any subsequent invocation* of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads.

```
public static void appendToLog(TaskScheduler scheduler, String s)
    synchronized (scheduler.textLog) {}
```

this adds text to the text area.

```
public static Server getOptimalServer(TaskScheduler scheduler)
```

this method takes the server with minimum number of clients and returns it as an optimal server.

Server.java

Server class implements Runnable, which actually makes this class to be a thread. This class contains:

private ArrayBlockingQueue<Task> tasks- using for storing the tasks

private volatile int totalEmptyTime = 0- used to compute the empty time for each server

private TaskScheduler scheduler;- the class that handles the tasks from server

private volatile Boolean runFlag- this attribute shows if a servers is active or not

private int ID;- unique identifier for server

In this class there are some variables that have a keyword „volatile”, this keyword is used to mark a Java variable as "being stored in main memory". More precisely that means, that every read of a volatile variable will be read from the computer's main memory, and not from the CPU cache, and that every write to a volatile variable will be written to main memory, and not just to the CPU cache.

public void run()- implementing runnable the class must also implement the method run

public void addTask(Task t)-used for adding a task in the queue

public void deleteTasks()—used for deleting a task from the queue

```
public void stopExecution() {
    synchronized (runFlag) {
        runFlag = false;
    }
}
```

Stops the execution of the each thread by setting the running flag to false

TaskGenerator.java

Task generator class it is used for creating a task with some parameters taken as a random number from a given range

private int minServingTime;

private int maxServingTime;

private int minArrivalTime;

private int maxArrivalTime;

private int simulationTime;

public ArrayList<Task> createTasks() as the name of the class, the main purpose of this class is to generate tasks, just how this method does generating them with random attributes.

TaskScheduler.java

The most important class from the project is taskscheduler. This class takes each tasks and put it into the right place at the right time. Some of the instance variable would be:

private int serverNumber = 0;- the number of servers
private int simulationTime = 0;- simulation time
private int currentCycle = 0;- the current cycle of the simulation
private ArrayList<Task> tasks – the array of tasks
private SimulationPanel panel;- the panel where is represented the simulation
private Server[] servers; the array of servers.

public void startServers()- this method creates a thread for each server and starts the thread

public void stopServers() – this method stops the servers by setting the run flag to false

public void updateEmptyTime()- adds at the empty time of the each server the time when it does nothing

public void updateRushHour() – gets the current cycle where are the maximum number of tasks in the servers

public void simulate()- this method gets the optimal server where a task should be added, and put the task there, also handles the random closing of one of the servers.

Because the class implements Runnable it means that it should also implement run. This is how it looks:

```
public void run() {
    frame.setGraphPanel(panel = new SimulationPanel(this));
    startServers();
    simulate();
    stopServers();
    updateEmptyTime();
    Utilities.showResultsPopUp(this, frame);
}
```

3.6 Controllers

Main.java

This is the starting point of the application, the main thread of the application. The class contains the main functions and it is represented as below:

```
public static void main(String[] args) {
    new MainFrameController(new MainFrame());
}
```

MainFrameController.java

More than that, for the JFrame that we created in the class MainFrame, we now add ActionListener and we handle the events. To do that, we create classes StartButtonActionListener, ClearButtonActionListener, that implements ActionListener. This implies the implementation of the method actionPerformed.

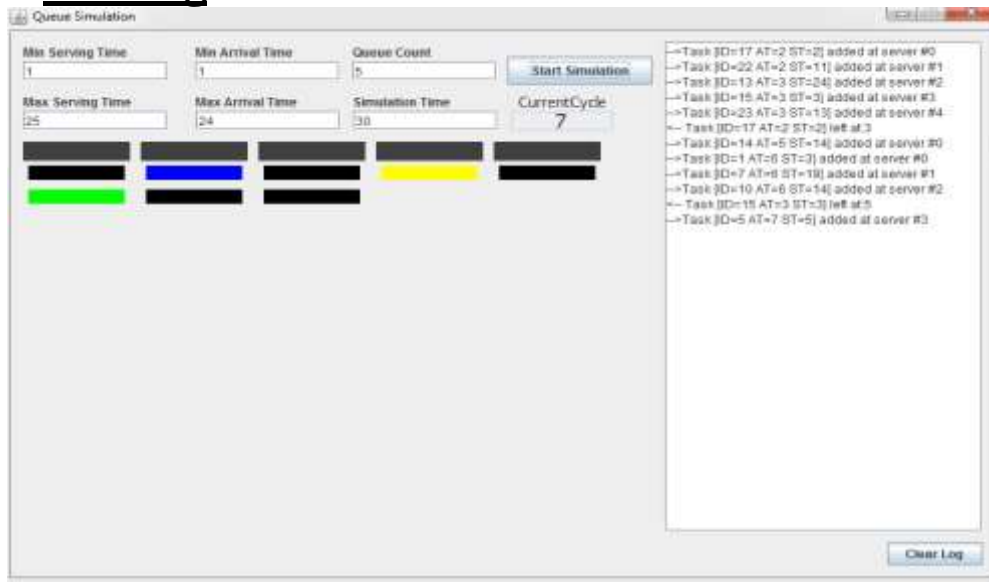
```
public class StartButtonActionListener implements ActionListener {
    @Override
```

```

    public void actionPerformed(final ActionEvent e) {
        instantiateServerManager();--takes all the information from the
        input/STDIN and builds up a scheduler, that helps us to simulate our application.
    }
}

```

4. Testing



For the data that I have introduced into the text fields the program works in a proper way. There can be noticed that in the right of the image it is written what happened in the queues, and in the left side is the graphical representation of what happens.

5. Results

The application is an user friendly and useful application to simulate and analyze queuing based systems for determining and minimizing clients' waiting time. Output results of the application are actually the calculation performed per each server. The information and the data are displayed both in written way but also in the graphical way, as can be seen from the picture that shows the graphical user interface. Even though being limited, this application can be considered a good and helpful tool in understanding and analyzing queuing based systems for determining and minimizing clients' waiting time.

6. Conclusion and future developments

All in all, I think this application pretty accurately simulates the behaviour of servers in a process where is given a certain priority to a task, although the tasks don't assess whether a certain server might yield a smaller waiting time, but blindly charge to the shortest server.

I have learned a great deal about threads and their functionality, as well as how to work with java Graphics, not only with Swing components as I've done so until now. Threads are not an easy concept to grasp, but they are essential when one has parts of a program that need to function at the same time (such were the queues in the case of this application).

Even though the application works as it is intended to, several improvements can be brought to it to make it even better. These improvements include, but are not limited to:

- A better queue selection algorithm, that makes the clients somewhat smarter (assess not only server length, but also the waiting time a server might bring)
- Error messages to notify the user that the input was incorrect and give reasons to why the program doesn't start running
- Implement dynamic queues: once a queue has reached full capacity, a new one should appear and tasks should be redistributed between the two servers
- A “fancier” graphical user interface instead of the dull rectangles
- The ability to run new simulations without the need of closing down the application and starting it up again

7. Bibliography

<http://stackoverflow.com/>

<https://docs.oracle.com/javase/tutorial/essential/concurrency/runthread.html>

<http://tutorials.jenkov.com/java-util-concurrent/arrayblockingqueue.html>

<http://www.java-examples.com/>