March 2016

Laboratory Assignment 3

Queuing based systems

Farcas Silviu Vlad

30425/1

Contents

1. Introduction

1.1 Task Objective

The task of this assignment is defined as follows:

TP Lab –Homework 3

Objective

Design and implement a simulation application aiming to analyze queuing based systems for determining and minimizing clients' 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 "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;

Our queuing based systems application should thus be able to graphically display queue evolution (identify customers and queues rectangles by their name) shown in real time, be able to be inputted data about number of queues, maximum load per queue, minimum and maximum of arriving time between clients and minimum and maximum service time for customers. At the press of the “Start” button, the button becomes disabled and the queuing process is displayed. When finished, “Start” button is again enabled.

1.2. Personal Approach

The aim of this paper is to present one way of implementing the required system, based on a simple and minimal GUI that provides the user with all possibilities described above. The UI will show in real-time the queue evolution with the help of a SwingWorker class which design is described later. The class will call publish() while the application is not

finished and the process() method will refresh the screen by calling the required method in the MainView class. Also, we have chosen the case in which a queue is shut down and its clients (with the exception of the one already served) will be redistributed to other queues. The UI also displays a Logging panel with all the events displayed in real-time. This will be accomplished by using the thread-safe method “append”, but more on that later. When pressed, the “Start button” will trigger the process, and the button will become inactive. When the process is finished, the button is re-enabled.

2. Problem Description

2.1. Problem Analysis

We all have been waiting at Billa, Lidl, or other supermarket chains and we know the golden rule: always position yourself at the queue with the least number of customers. Although, our brain does a little bit more, and also tells us to position at the queue with the customers who have the least number of products, that is, where the average service time is the least. Our application will not mimic in that detail the human brain; this could be a subject of future development. A queue is a basic FIFO data structure (First In First Out) meaning that the first client who enters the queue is the first who is served. Queuing systems are ubiquitous in the market: at banks, at supermarkets, at auto dealers etc. Hence, a queuing based system application would come in handy for many a corporate clients out there.

We shall henceforth call a client a Task and a queue a Server. A Server will be a running thread which will continuously process tasks until the end of the simulation or until the server is shut down by a random procedure.

The Task Generator is responsible for creating tasks at random time interval, between a max and a min time interval specified at runtime. The tasks are deferred to the TaskScheduler which will schedule the task to a specific server depending on the number of customers at each Server.

2.2 Modeling

A task is described by its arrivalTime, its finishTime which is initially -1, its serviceTime, its waiting time which is initially -1, its running server where it is waiting, its name and whether it was rescheduled or not.

The server is described by its tasks, by its name, and by the Boolean volatile variable isShutdown.

The TaskGenerator is described by its minServiceTime, maxServiceTime, minArrivalInterval, maxArrivalInterval, by the nrOfCustomers, by the server which is shutdown, by the serverSHutdownTime and by the thread which is runs the serverShutdown.

The TaskScheduler is completely described by its simulationTime, its startTime, its numebrOfServers, by the list of servers, of threads of runningServers and of custoemrsArrived, by a maxLoadPerServer, by the peakHour Date, by currentCustomers, by peakHourCustoemrs, by the serverShutdown, serverShutdownTime and Thread of the serverShutdown.

2.3 Scenarios

One can enter the desired data (number of queues, max load per queue etc.) and press Start. The program doesn’t check for data validity, i.e., the minArrivalTime should be less than the maxArrivalTime etc. This could be a subject of future developments.

Once the user presses Start, the button becomes inactive and the process starts. The UI is constantly refreshing and the logging screen is showing queue evolution in real time. After the process is ended, the user will see statistics about the simulation (peak hour, average waiting time, average service time) and the button “Start” will become enabled so that a new process could be generated.

2.4 Use cases

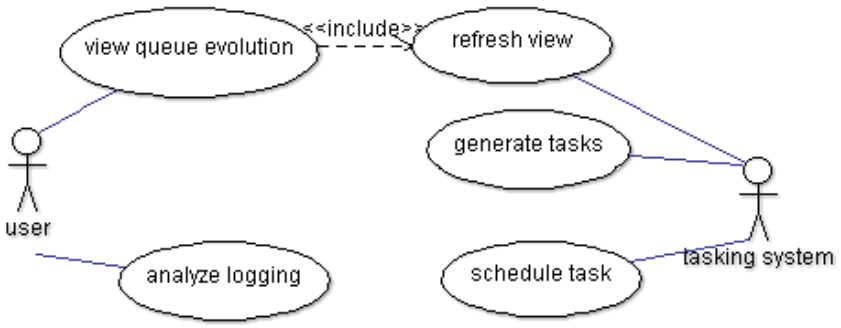
The user can graphically view queue evolution, and analyze logging in the logging pane. The system will refresh view, generate tasks and schedule tasks for appropriate servers.

3. Design

3.1. UML Diagrams

In the following we will present Use Case diagrams, Class Diagram, Sequence Diagram.

3.1.1. Use Case Diagrams



Use Case Description

The user can view queue evolution and analyze logging. The system will generate tasks, schedule task and refresh view.

Trigger

The user presses start.

Actors

1. The user
2. The tasking system

Preconditions

The user enters valid data.

Goals (Successful Conclusion)

User view queue evolution and queue logging.

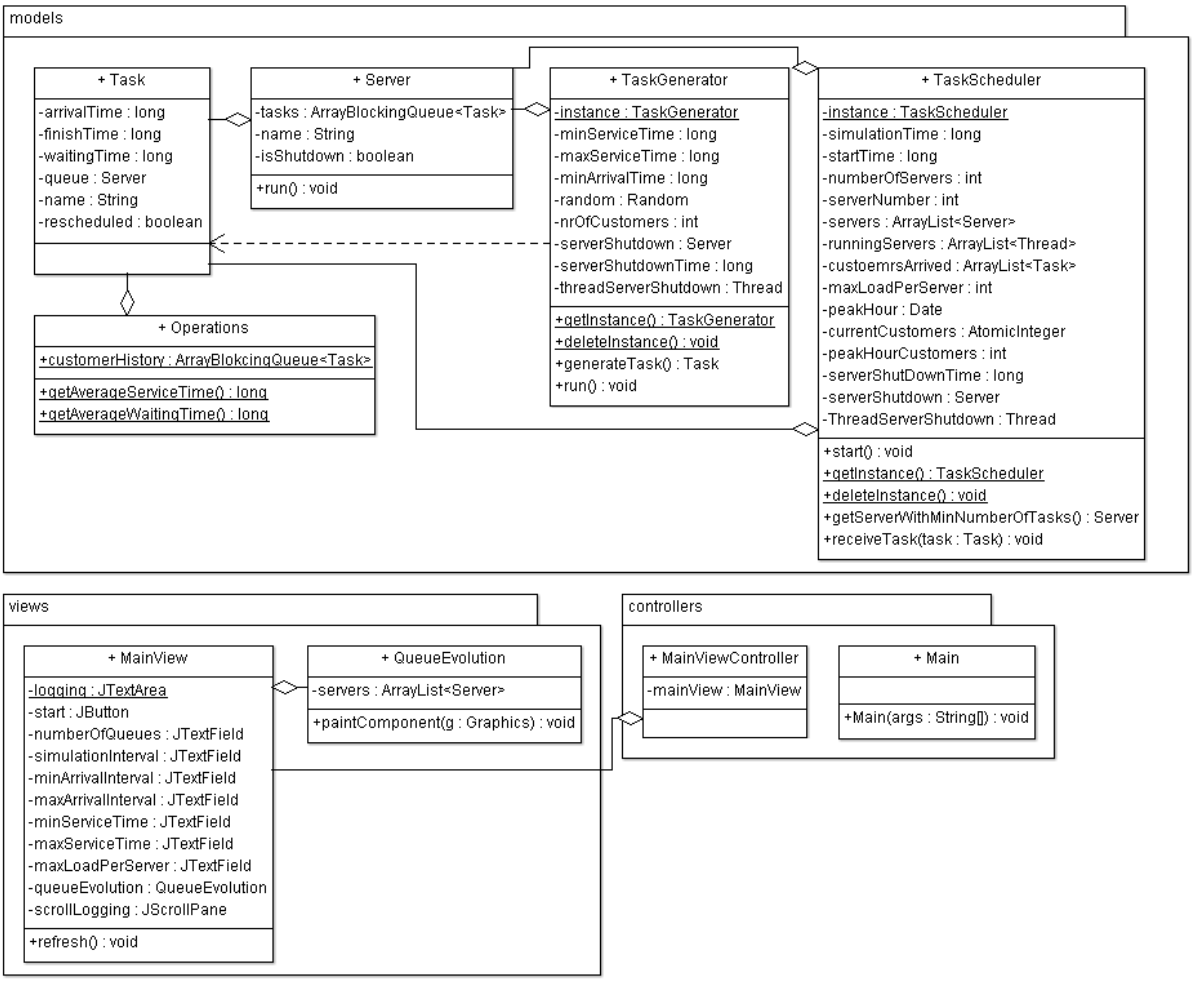
Failed Conclusion

Program throws exception.

Steps of execution

1. The user enters queuing data.
2. The user presses start.

3.1.2. Class Diagram (next page)



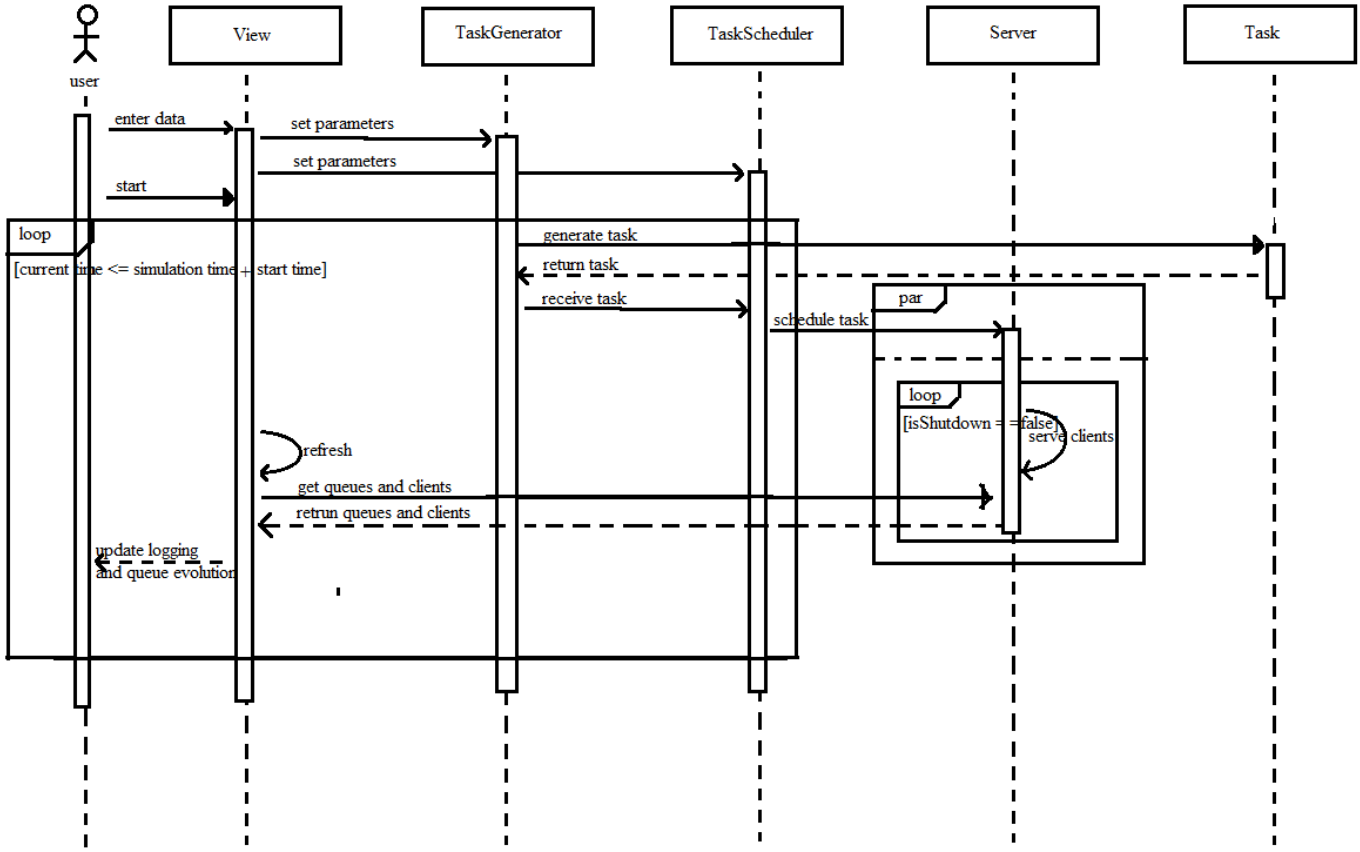
The class diagram shows the modelling of our problem into classes. There are 9 classes. One must know that not all fields and methods were represented on the diagram, only the ones that are important to our implementation (for example, getters/setters are not shown). Also, not all dependencies have been drawn, otherwise it would make the diagram unreadable. I have used aggregation in the following cases:

1) Server has many Tasks
2) TaskGenerator has a Server

3) Operations has many Tasks
4) TaskScheduler has many Servers
5) MainView has a QueueEvolution
6) MainViewController has a MainView

The rest of the relations are dependencies.

1.1.2. Sequence Diagrams



The sequence diagrams shows how the queuing process is working. First, we eneter data, then we set parameters, then we loop until the process is finished. In this loop, we generate tasks, the servers will rpocess tasks and the view will update the UI.

3.2. Data Structures

Besides usual primitive data types like int, long etc. and besides String, our program extensively makes use of ArrayList. These are useful in situations where multithreading is not present, like in TaskScheduler class for example. Also, in multithreading context, our application makes use of an implementation of BlockingQueue called ArrayBlockingQueue. In essence, this is a thread safe collection that is used in a FIFO manner, that is, the first element removed is the first element arrived. This mimics in detail the architecture of a real-life queue like the one at Lidl. ArrayBlockingQueue has a fixed maximum size which is set at construction time. ArrayBlockingQueue has a method called “put” which adds an element to the tail of the queue, or, if the queue is full, patiently waits for the queue to decrement in size. ArrayBlockingQueue has a method called “take” which removes an element from the head of the queue. If the queue is empty, it patiently waits for the queue to be incremented in size. There is also another method which we used in our implementation which is peek (returns the element at the head without removing it). These methods are used extensively in the implementation of the run() method of the Server thread, which is detailed later at Algorithms. Also, we use an AtomicInteger in TaskScheduler class. The AtomicInteger class offers thread-safety for incrementing and decrementing an integer, hence our interest in it.

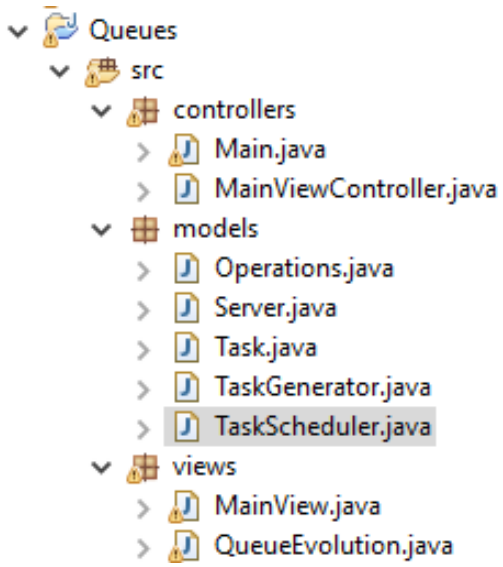
3.3. Class Design

The problem domain was decomposed into sub-problems, and for each sub-problem a solution was build. By assembling the solutions we obtain the general solution to our problem. The sub-problems were:

- how to correctly model an queue-based processing system?

- how will be implemented the user interface?
- how can we assure user interaction with our models?

For each of the above mentioned, we came up with a package that would support the features of order processing system, namely increasing/decreasing orders, adding/removing products, buying, searching and viewing history, the user interface and user interaction with our model. These packages follow the Model-View-Controller software architecture and they are as follows:



The **Task** class has an arrivalTime, a finishTime, a serviceTime, a waitingTime, a Server to which it belongs, a name and a Boolean variable to know whether the task has been rescheduled, that is, if the task was waiting at a queue which has closed, and thus was moved to another queue.

The **Server** class is the base for constructing server threads. It has an ArrayBlockingQueue of Tasks, a name, and a volatile Boolean isShutdown (in case it gets shutdown in the TaskGenerator thread by random means, and algorithm which we will show at the algorithms

section.) The run method() will run as long as the value of isShutdown is not changed by an outside process. Note that if there were clients served, the server thread will shutdown only after those clients were served.

In the **Operations** class, we will store each and every customer that has been created in our application. Thus, at the end, we will be able to compute the Average Service Time and the Average Waiting Time. The next two methods actually implement this, and there concrete details are mentioned in the Algorithms section.

The method getAverageServiceTime() returns the required time in miliseconds. The method getAverageWaitingTime() returns the required time in miliseconds. All these methods are static.

The **TaskScheduler** class is the most complex of our classes. We implemented it using a Singleton Pattern, since no more than one instance is needed/simulation. It contains the simulationTime, the startTime, the numberOfServers, the actual servers, an ArrayList of running Servers (which are threads), a detailed list of customers arrived, the maxLoadPerServer, the peak hour (stored using Date class), the currentCustomers, the peakHourCustomers, the server which is to be shutdown, the sevre shutdown time and the thread which runs the shutdown server.

The start() procedure initializes the servers, starts the server threads and decides for the random server to be shutdown, and at what time.

The getServerWithMinNumberOfTasks returns the server with the lowest number of customers waiting in line. The algorithm is better described in the Algorithms section.

The receiveTask(task: Task) method is a bit more complex. It analyzes the situation of current servers and determines what is the correct server where the task should be scheduled.

The **TaskGenerator** class is implemented using Singleton Pattern, again, since only one object is need/simulation. It has aminServiceTime, a maxServiceTime, a minArrivalInterval, a maxArrivalInterval, nrOfCustomers, the server which is to be shutdown, the time when is to be shutdown and the thread which runs the server.

The generateTask() method generates a task at a random interval between the specified limits. The run method implements the thread logic. Until the termination time, it constantly sends tasks to the Task Scheduler.

The **MainView** class has several JTextFields, like numberOfQueues, min & max ArrivalTime etc., a JButton start, a JTextAreaa Logging wrapped in a JScrollBar and a panel which is an object of type WueueEvolution. The refresh() method:

```
public void refresh(){
            queueEvolution.repaint();
            revalidate();
            repaint();
      }
```

assures constant refresh of the UI with the data in models.

The QueueEvolution has an ArrayList of servers and the paintComponent method:

```
public void paintComponent(Graphics g) {
            super.paintComponent(g);
            g.setColor(Color.BLUE);
            g.fillRect(0, 0, 800, 250);
            servers = TaskScheduler.getInstance().getServers();
            int i, j;
            for (i = 0; i < servers.size(); i++) {
                  g.setColor(Color.WHITE);
                  g.fillRect(i * 100 + 10, 10, 80, 10);
                  g.setColor(Color.BLACK);
                  g.drawString(servers.get(i).getName(), i * 100 + 20, 20);
                  j = 1;
                  for (Task t : servers.get(i).getTasks()) {
                        g.setColor(Color.CYAN);
                        g.fillRect(i * 100 + 10, j * 20 + 10, 80, 10);
                        g.setColor(Color.BLACK);
                        g.drawString(t.getName(), i * 100 + 20, j * 20 + 20);
                        j++;
                  }
```

```
            }
        }
```

The **MainViewController** class has a mainView and a isCancelled Boolean volatile variable to stop the refreshing when the simulation has ended. The StartButtonActionListener triggers the whole simulation process, and also starts a SWingWorker class, called Refresh, which with the help of the publish and process methods automatically refreshes the queueEvolution panel.

The **Main** class is the one which contains the main function.

This concludes a quick introduction into the aspect of our class design.

3.4. Interfaces

Our design does not make use of Interfaces.

3.5. Relations

In Object Oriented Programming, it is a good habit to design classes as loosely coupled as possible, in order to avoid error propagation. I tried to stick with this principle and, with few exceptions, I think I managed to fulfill this requirement. In this design, one can found the following relations: aggregation, dependency and innerness.

3.6. Packages

The design follows the Model-View-Controller (MVC) architecture. It has three packages: models, views, controllers.

3.7. Algorithms

In the following we will detail only the more intriguing algorithms used in our program.

First, let’s start with the Server’s run method, completely provided below:

```
public void run() {
            while (!(isShutdown)) {
                  try {
                        Task task = tasks.peek();
                        if (task != null) {
                              task.setWaitingTime(System.currentTimeMillis()
- task.getArrivalTime());
                              MainView.getLogging().append(task.getName() + "
is being served at " + name + " at time " + String.format("%tT", new
Date(System.currentTimeMillis())) + "\n");
                              Thread.sleep(task.getServiceTime());
                              task.setFinishTime(System.currentTimeMillis());
                              MainView.getLogging().append(task.getName() + "
has been served at time " + String.format("%tT", new
Date(System.currentTimeMillis())) + "\n");
                              tasks.poll();

      TaskScheduler.getInstance().getCurrentCustomers().getAndDecrement();
                        }
                  } catch (InterruptedException e) {
                        return;
                  }
            }
      }
```

If the server is not shutdown, continue processing. Then, we peek() at the top element. If there is a client, we set all the times accordingly and we append to the logging screen the required info. Then, we remove it.

Now, let's turn our attention to the randomization which takes place when choosing the server to shutdown:

```
serverShutdownTime = (long) (new Random().nextDouble() * (simulationTime) +
startTime + 1);
            i = new Random().nextInt(numberOfServers);
            serverShutdown = servers.get(i);
            threadServerShutdown = runningServers.get(i);
```

Then, in the TaskGenerator, we have:

```
if ((System.currentTimeMillis() - serverShutdownTime <= 1000)
                        && (System.currentTimeMillis() -
serverShutdownTime >= 0) && !(serverShutdown.isShutdown())) {
                        Thread t = new Thread(new ServerShutdownHandler());
                        t.start();
                  }
```

That is, we the error between the current time and the shut down time is less than 1000 milisec, do the shutdown. The Shutdown is performed by starting a new thread, provided by the ServerShutdownHAndler class, fully reproduced below:

```
public class ServerShutdownHandler implements Runnable{

		public void run(){
			serverShutdown.setShutdown(true);
			MainView.getLogging().append(
					serverShutdown.getName() + " is closing at time
" + String.format("%tT", new Date(System.currentTimeMillis())) + "\n");
			try {
				threadServerShutdown.join();
			} catch (InterruptedException e) {
				return;
			}
			MainView.getLogging().append(
					serverShutdown.getName() + " has been closed at
time " + String.format("%tT", new Date(System.currentTimeMillis())) + "\n");
			for (Task t : serverShutdown.getTasks()) {
				t.setRescheduled(true);
				TaskScheduler.getInstance().receiveTask(t);
			}

	TaskScheduler.getInstance().getServers().remove(serverShutdown);
		}

	}
```

Then, let's switch our attention at the getAverageWaitingTime, and getAverageServiceTime method. We will detail only one of them, since the other is symmetric.

```
public static long getAverageServiceTime() {
		long totalServiceTime = 0;
		long numberOfCustomers = 0;
		long startTime = TaskScheduler.getInstance().getStartTime();
		long finishTime = TaskScheduler.getInstance().getSimulationTime()
+ startTime;
		for (Task t : customerHistory) {
			if ((t.getArrivalTime() > startTime) && (t.getWaitingTime()
+ t.getArrivalTime() < finishTime)
					&& (t.getWaitingTime() != -1)) {
				totalServiceTime += t.getServiceTime();
				numberOfCustomers++;
			}
		}
		return totalServiceTime / numberOfCustomers;
	}
```

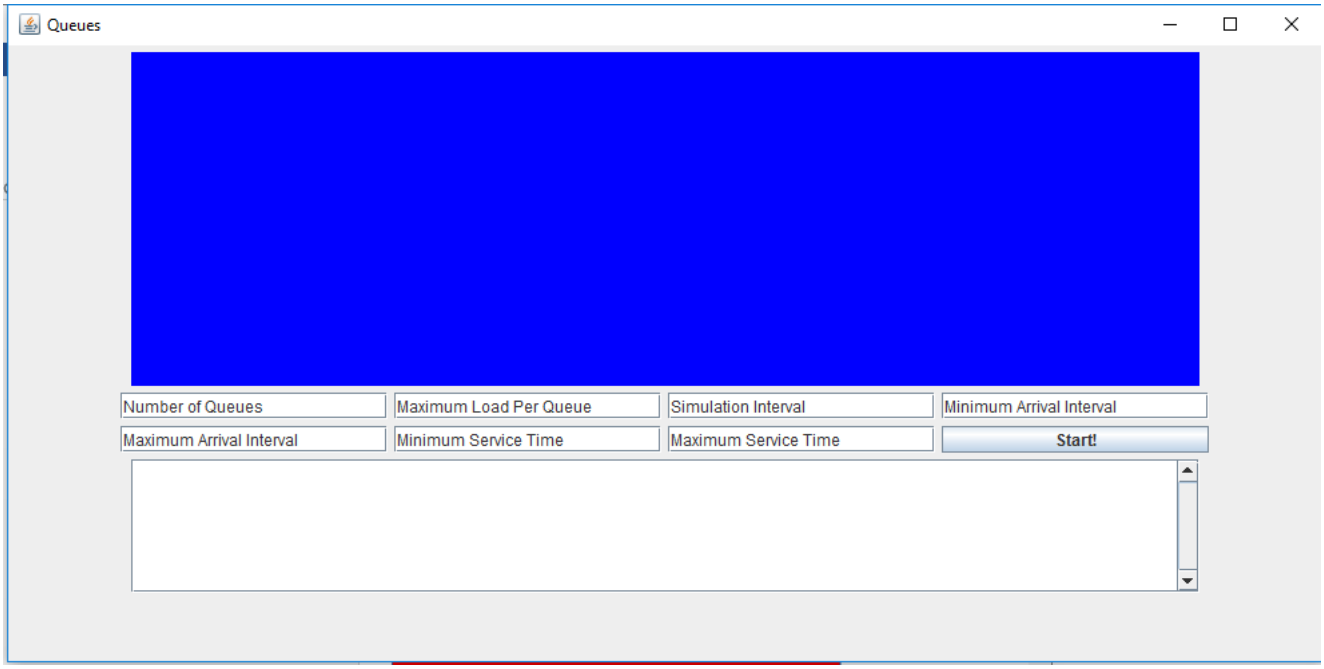
Now, let's switch our attention to the getServerWithMinNumebrOfTasks method. The mthod is fully reproduced below and no other commentations are necessary:

```
public Server getServerWithMinNumberOfTasks() {
		int min = maxLoadPerServer;
		Server minServer = null;
```

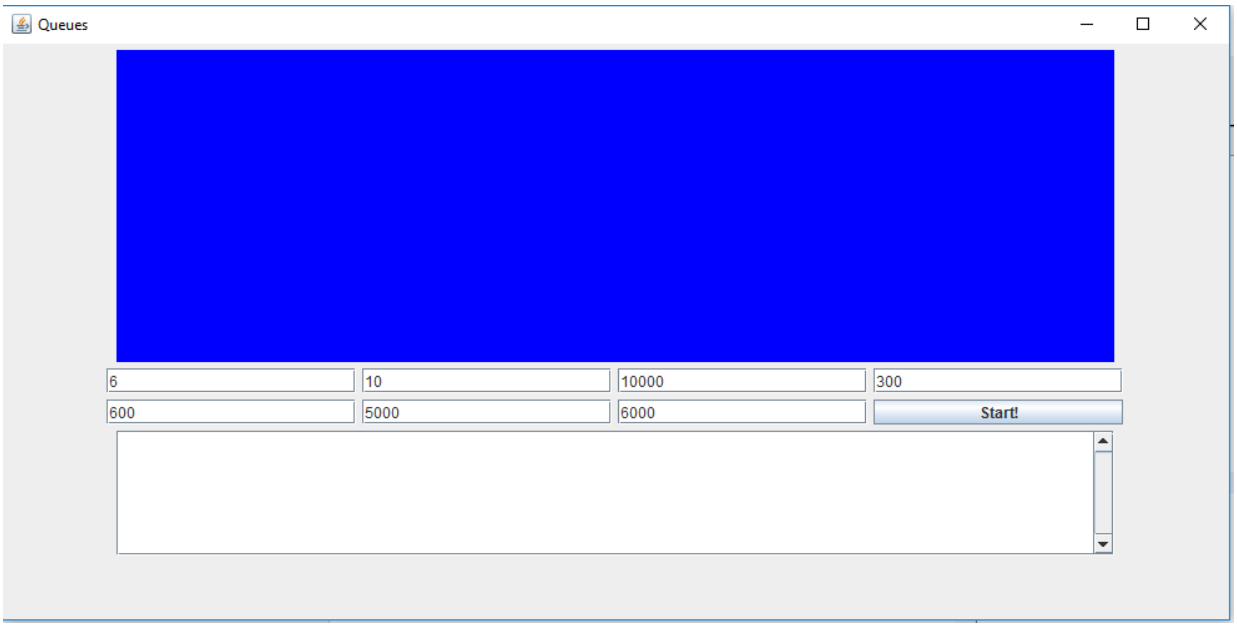
```
            for (Server s : servers) {
                  int size = s.getTasks().size();
                  if (size < min) {
                        min = size;
                        minServer = s;
                  }
            }
            return minServer;
      }
```

3.8. User Interface

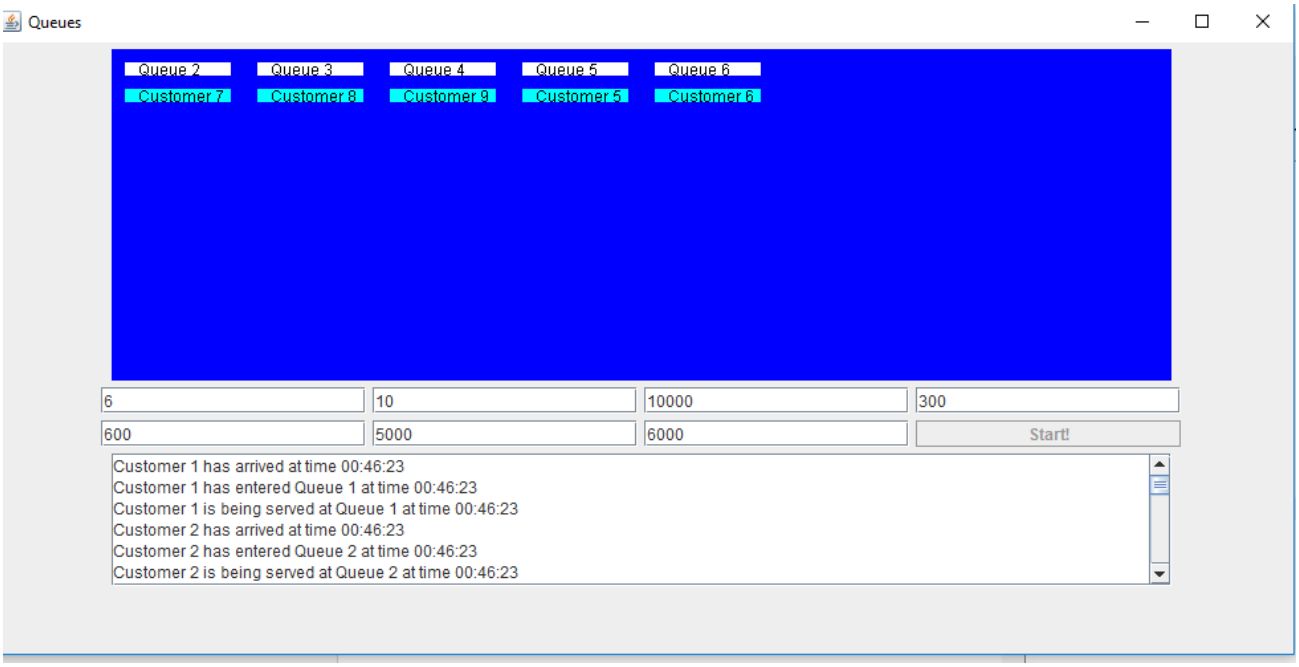
The base screen:



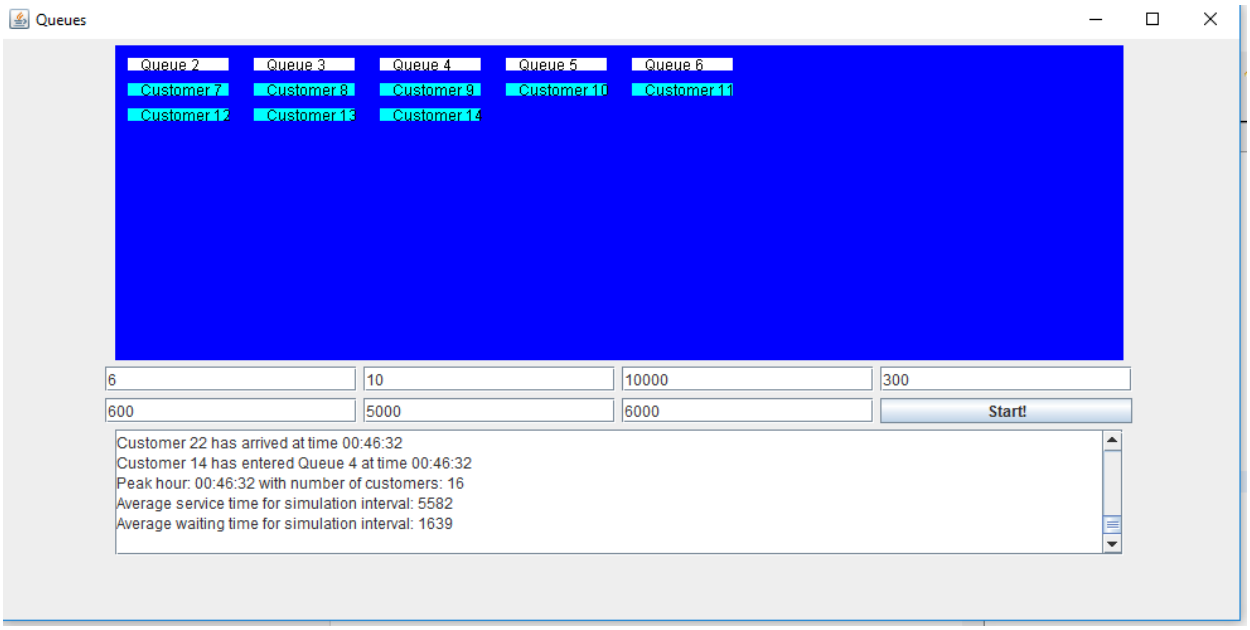
The view after entering data:



The view during execution:



The final view with peak hour, avgService&waitingTime:



4. Implementation and Testing

The project was developed in Eclipse IDE using Java 8, on a Windows 10 platform. It should maintain its portability on any platform that has installed the SDK. It was heavily tested, but new bugs could be discovered in the near future. One of the main inconveniences are that the program is not protected against all kinds of illegal input data, only to some. This could be the subject of future development.

5. Results

The application is a user-friendly, helpful app that can perform queue-based systems simulations. It is particularly useful in the hnads of corporate clients which would want to know the exact flow of cutomers

at a certain time in order to optimize the queuing process by opening/closing different servers.

6. Conclusions

6.1. What I've Learned

TIME IS PRECIOUS! I had to solve complicated problems of time management which taught me good organization skills. I learned that a good model is always a key to a successful project and that a bad model could ruin your project in the end. I learned never to get stuck on one little bug, and if I do, ask for help, either from the TA or from colleagues or on different forums. I am looking forward to apply what I have learned in future projects.

6.2. Future Developments

The app could be made more user-proof. That is, if the user enters a String at a numerical field, an error dialog pops-up. Or, if the lower bounf of, say, ArrivalTime is greater than the upper bound, an error window pops-up. Also, we could make the screen larger to accommodate more queues. Also, we could try to make the view more user-friendly by replacing squares with little people of different colors and queues by some sort of checkout image. We could also make the app open more queues if the queues already functioning are overcrowded. Also, we could close some queues if there are little to no clients there.

7. Bibliography

[1] http://www.uml.org/

[2] Barry Burd, *Java For Dummies*, 2014

[3] Kathy Sierra, Bert Bates, *Head First Java*, 2005

[4] Steven Gutz, Matthew Robinson, Pavel Vorobiev, *Up to Speed with Swing*, 1999

[5] Joshua Bloch, *Effective Java*, 2008

[6] http://www.stackoverflow.com/

[7] http://docs.oracle.com/

[8] http://www.coderanch.com/forums

[9] Derek Banas' Channel on YouTube
https://www.youtube.com/channel/UCwRXb5dUK4cvsHbx-rGzSgw

[10] Cave of Programming Channel on YouTube

https://www.youtube.com/user/caveofprogramming/playlists