MINISTRY OF EDUCATION AND SCIENTIFIC RESEARCH

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA

Simulation of queue based system

- Project #3 -

Student: Dariana Lupea

Teacher: Delia Balaj

Group: 30425

Deadline: April 15, 2016

Table of contents

1. **Introduction**
  1.1 Problem specification 3
  1.2 Personal interpretation 3
2. **General aspects**
  2.1 Problem analysis 4
  2.2 Modeling 4
  2.3 Scenarios and use cases 4

3. **Projection**
  3.1 UML diagrams 5
    3.1.1 Use-case diagram 5
    3.1.2 Class diagram 6
    3.1.3 Sequence diagram 6

  3.2 Data structures 6
  3.3 Class projections 7
  3.4 Packages 7
    3.4.1 Model package 7
    3.4.2 GUI package 8
    3.4.3 View package 9
  3.5 Algorithms 10
  3.6 Interface 13

4. **Implementation and testing** 15

5. **Results** 16

6. **Conclusions** 16

7. **References** 16

1. Introduction

1.1 Objective

The objective of this project is to design and implement a simulation application aiming to analyze queuing based systems for determining and minimizing client's waiting time.

1.2 Problem specification

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;

1.3 Personal interpretation

This application is, in my opinion, a great opportunity of learning how to deal with threads and also to understand the main logical aspects imposed by such an approach. In order to simulate in real time the queue based system, I worked with threads. I will shortly describe a few theoretical aspects concerning them:

Multithreading refers to two or more tasks executing concurrently within a single program. A thread is an independent path of execution within a program. Many threads can run concurrently within a program. Every thread in Java is created and controlled by the **java.lang.Thread class**. A Java program can have many threads, and these threads can run concurrently, either asynchronously or synchronously.

Multithreading has several advantages over Multiprocessing such as;

- Threads are lightweight compared to processes
- Threads share the same address space and therefore can share both data and code
- Context switching between threads is usually less expensive than between processes
- Cost of thread intercommunication is relatively low that that of process intercommunication
- Threads allow different tasks to be performed concurrently.

There are two ways to **create** a thread in Java:

- **Implement the Runnable interface (java.lang.Runnable)**
- **By Extending the Thread class (java.lang.Thread)**

In my application design, I have implemented the Runnable interface and I have also overridden the **Run** method, which contains he code that will be run several times at the runtime.

2. General aspects

2.1 Problem analysis

Considering the fact that the time is an important factor for implementing this program, I had to use resources able to manage this aspect. A queue system simulation implies the existence of a **simulation time** and also an **individual time** for each queue. Our goal is to distribute the clients in such a way that each of them waits for the minimal possible time at the queue. This means that, at any time, when a client arrives, we have to find the suitable queue and send them there.

2.2 Modeling

At the modelling level, I have sketched some of the classes which I needed to implement. A client is called "***Task***", each queue is a "***Server***". In order to simulate the way that clients are coming to queues, I used a "***Task Generator***" to continuously generate tasks and a "***Task Scheduler***" which imposes to the servers an order of receiving tasks. The criteria of choosing the right server each time is the number of tasks at a certain server -> the server with the minimum number of tasks waiting will be chosen.

2.3 Scenarios and use cases

Before getting to the final version, it is obvious that I have tried different scenarios and my program passed through different stages.

The main problem I have encountered was working with threads, that is why it took me a long time to figure it out how to correctly use them.

An obvious use case of my application would be managing a supermarket or a big shop, where there are several queues and also many clients.

3. Projection

3.1. UML Diagrams

3.1.1 Use – case diagram

A **use case diagram** at its simplest is a representation of a user's interaction with the system that shows the relationship between the user and the different use cases in which the user is involved.

I think that the perfect user of this application would be a person who wants to simulate an ideal queue based system, where some time parameters such as arrival time and service time for each element from the queue are known. Of course, in the real life, this aspect cannot be efficiently implemented.

In this case, the use-case diagram looks like this:

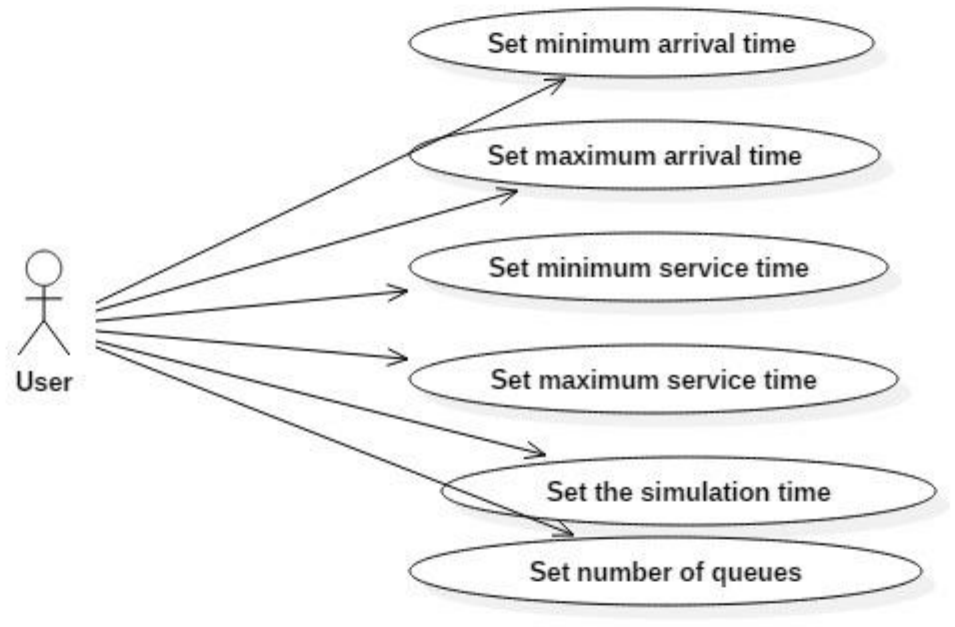


Fig. 1: Use-case diagram

3.1.2. Class diagram

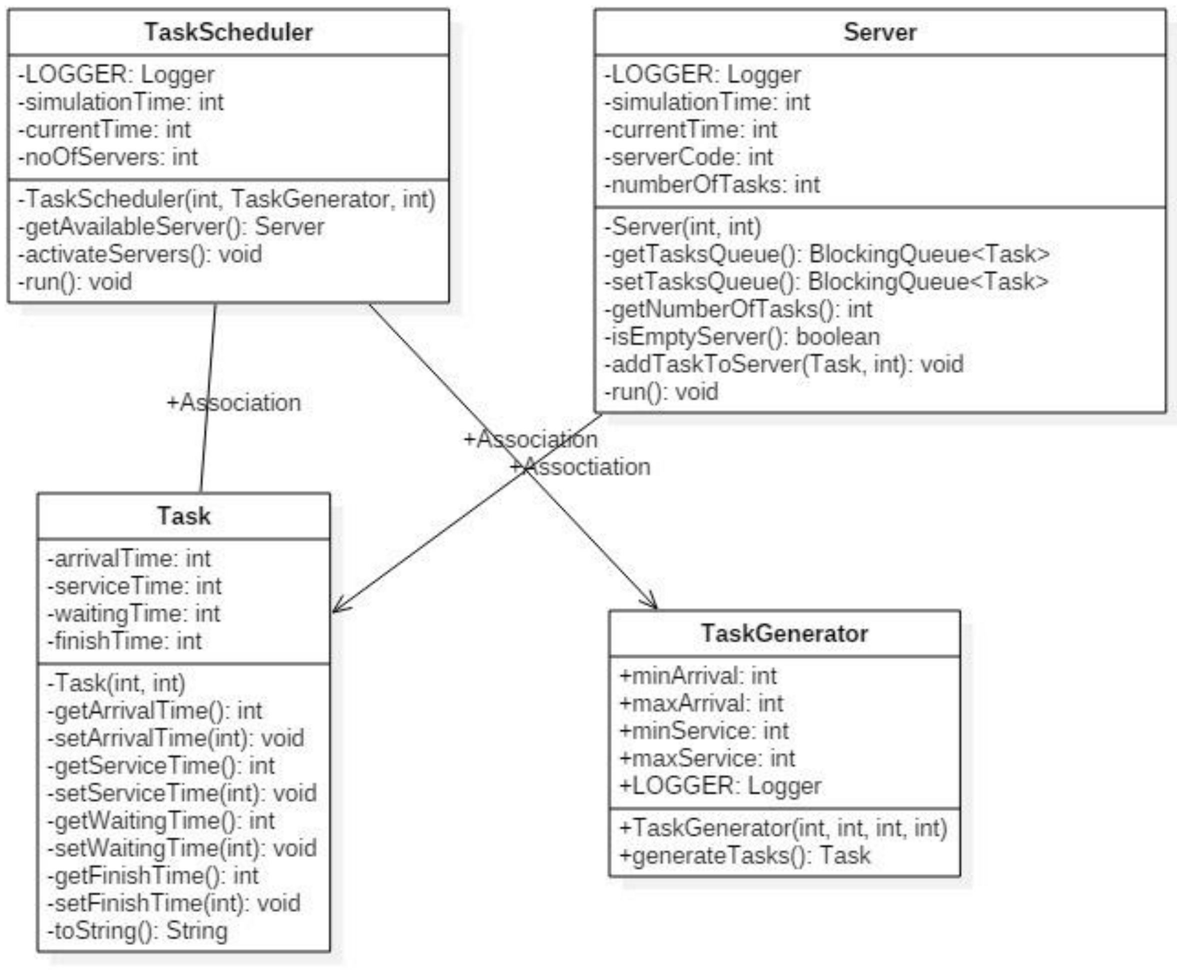


Fig. 2: Class diagram

3.1.3. Sequence diagram

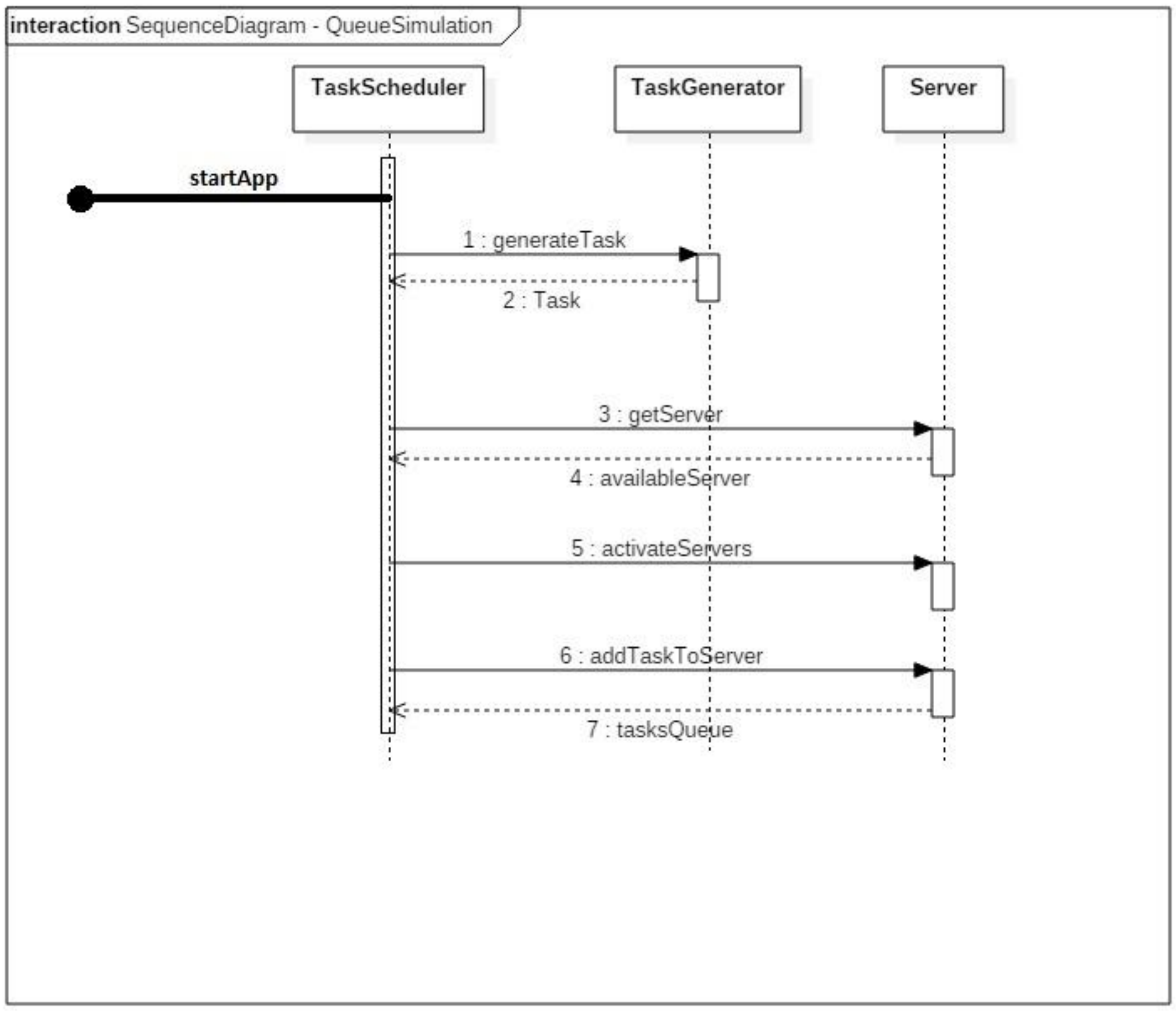


Fig3. Sequence diagram

3.1.4. Activity diagram

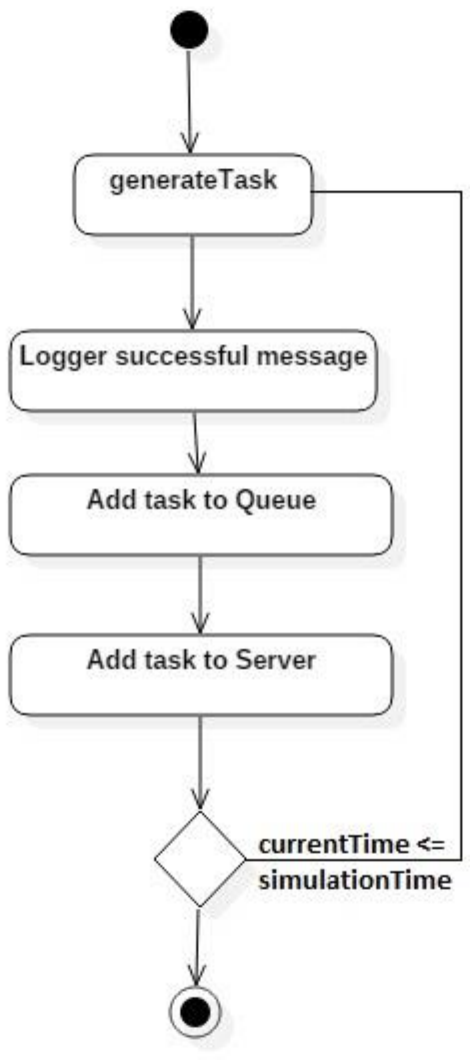


Fig4. Activity diagram

3.2 Data Structures

In this project implementation I have used several data structures, but the most significant ones are:

- **ArrayBlockingQueue**: this data structures allows working with objects in a FIFO manner. The first element is removed from the blocking queue with **take()** and an element is added to the queue using **put().**
- **ArrayList**: I have used the ArrayList collection in order to keep track of the Servers which are active at the simulation time.

3.3 Class Projection

In the project design implementation I tried to use the **MVC** pattern. After several changes, I obtained the final version of the system's structure. It contains 2 packages, namely: **GUI** and **Model**. Each of these consists of other classes, which perform specific tasks. I will present them below:

- ***Model*** package:
  - **Task** class – used for modeling the client entity
  - **Server** class – it models the queues
  - **TaskGenerator** class **–** is responsible with continuously generating tasks
  - **TaskScheduler** class – it takes the decision of where to send the tasks. It manages the tasks and the queues.
  -
- **GUI** package:
  - **SimulationFrame** class – creates the user interface

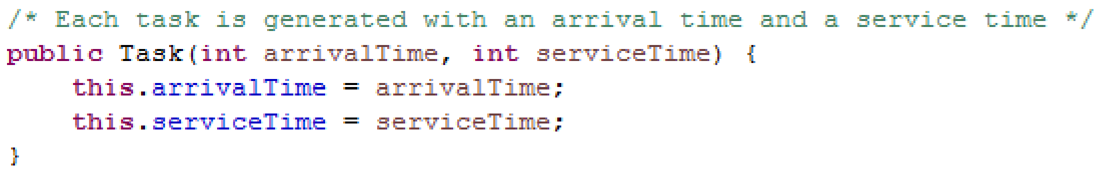
3.4 Packages

3.4.1 **Model** package

It contains 4 classes, which capture the behavior of the application in terms of the problem domain: **Task**, **Server, TaskGenerator** and **TaskScheduler**.

a) Task class

It models an entity of type **Task**. Each task represents a client. It has the following properties: an arrival time, a service time, a waiting time and a finish time. The arrival time and the service time are known from the beginning, so I have used the following constructor:



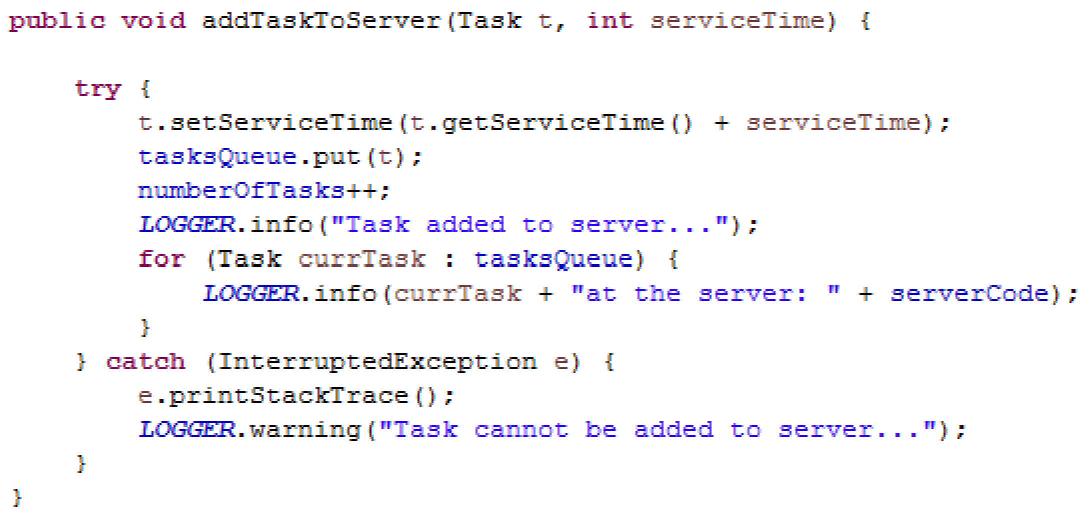
```
/* Each task is generated with an arrival time and a service time */
public Task(int arrivalTime, int serviceTime) {
    this.arrivalTime = arrivalTime;
    this.serviceTime = serviceTime;
}
```

➢ This class also also contains some **getters** and **setters** which allow the modification of the attributes mentioned before

b) Server class

This class represents a queue. It implements the runnable interface, because it contains some piece of code that needs to be executed several times, for each thread. A Server has a blocking queue of tasks, in order to process them in a FIFO manner. Each Server has its own **currentTime**, which must not exceed the **simulationTime**, given by the user.

An important method used here is *addTaskToServer*, which adds a generated task to the waiting queue of a Server.



```
public void addTaskToServer(Task t, int serviceTime) {

    try {
        t.setServiceTime(t.getServiceTime() + serviceTime);
        tasksQueue.put(t);
        numberOfTasks++;
        LOGGER.info("Task added to server...");
        for (Task currTask : tasksQueue) {
            LOGGER.info(currTask + "at the server: " + serverCode);
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
        LOGGER.warning("Task cannot be added to server...");
    }
}
```

This class also contains the method *Run*, where as long as the **currentTime** is less than the **simulationTime**, if the **serviceTime** of a task is reached, that task is removed from the Server.

c) TaskGenerator class

This class generates tasks, by using some input data from the user: the minimum arrival time, the maximum arrival time, the minimum service time and the maximum service time. An integer is generated, one for each interval, representing the **arrival** time of a task and the **service** time, respectively.

```
/* Creates a new task with the given time */
public Task generateTask() {
    int randomArrival = ThreadLocalRandom.current().nextInt(minArrival, maxArrival);
    int randomService = ThreadLocalRandom.current().nextInt(minService, maxService);
    Task newTask = new Task(randomArrival, randomService);
    LOGGER.setLevel(Level.INFO);
    LOGGER.info("Generated new task...");
    return newTask;
}
```

d) TaskScheduler class

In this class, a lot of logic had to be used, because this is the core of the queue based systems, it manages the tasks and when they are sent to the servers. It contains the following attributes, which I will describe shortly:

- **private List<Server> servers:** this list contains a number of servers chosen by the user
- **private TaskGenerator taskGen:** is an object of TaskGenerator type, used to generate tasks
- **private int simulationTime:** input from user, represents the simulation duration in seconds
- **private int currentTime:** this time is specific to the class and it is incremented by one whenever the **run** method is executed. It helps us keeping track of the current time.
- **private int noOfServers:** input from user, the number of Servers active at one simulation

It also contains the following method:

```
public Server getAvailableServer() {
   int minTasks = 10000;
   int currTasksNo;
   int availableServer = 0;

   for (int i = 0; i < servers.size(); i++) {
         currTasksNo = (servers.get(i)).getNumberOfTasks();
         if (currTasksNo < minTasks) {
               minTasks = currTasksNo;
               availableServer = i;
         }
   }
   LOGGER.info("Found available server");
   return servers.get(availableServer);
}
```

The above method is used to select the proper Server. Each time, the server with the minimum number of tasks in the queue has to be found. That Server is returned by the method.

The ***run()*** method is used for adding task to the chosen Server and also here we generate tasks.

3.4.2 GUI package

It contains a class responsible with creating the user interface. I used **Swing**, which is a GUI widget toolkit for **Java:**

- **a top level container:** the class *SimulationFrame* extends **JFrame** and I have created an object of this class called *f*

- **JComponents: JPanels, JLabels, JTextFields:**

  - **private JPanel** panel, content, servers, tasks;

    These are the panels where the simulation will take place.
  - **private JButton b1, b2, b3, b4, b5, b6, startButton;**

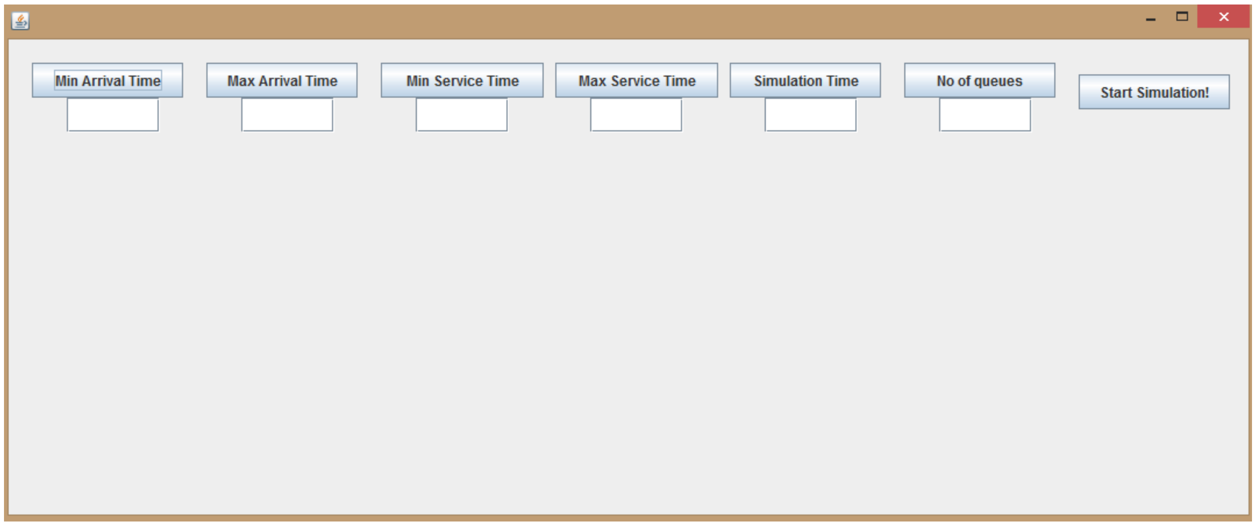
    The **startButton** has an action listener and when this button is pressed the simulation begins.

  - **private JTextField field1, field2, field3, field4, field5, field6.**

These JTextFields are used for taking the user inputs.

3.6 Interface

This application GUI looks like in the following image:



- The **user** should:
  - Enter the minimum arrival time
  - Enter the maximum arrival time
  - Enter the minimum service time
  - Enter the maximum service time
  - Enter the simulation time
  - Enter the number of queues
  - Press the “Start simulation!” button

4. Implementation and testing

Regarding the implementation process, I used the **Eclipse IDE**. During the program development steps, many changes were made and as I started to have more and more methods I realized the importance of a good structure. I refer here to organization of the code in packages and classes. I consider that the code I wrote is understandable and reusable. The algorithms I used are relatively easy, based on the well-known mathematical algorithms.

I have also tested each method, in the following way:

-> I used the console at first, for displaying the result
-> I initialized data with some appropriate values
-> I checked if the obtained result was the expected one
-> if YES, I continued the implementation process
-> if NO, I tried to figure out where is the error / problem coming from and solve it

5. Results

Concerning this project, I believe that it does not cover all the requirements from the problem specification, but it fulfills the main purpose, which is determining and minimizing client's waiting time. I worked a lot at this aspect and finally, I obtained the desired result.

6. Conclusion

To conclude, I can say that this project meant hard work, a lot of new things learned, focusing, development and creativity. Even if I encountered a lot of problems, I was able to fix them after all, by searching on the internet or asking a colleague for advice. I think that my application satisfies the requirements and the users will have at their disposal all its functionalities.

7. References:

- http://stackoverflow.com
- http://www.wideskills.com/java-tutorial/java-threads-tutorial
- https://www.javacodegeeks.com/2013/01/java-thread-pool-example-using-executors-and-threadpoolexecutor.html