Technical University of Cluj-Napoca
Programming Techniques
Assignment 2

Queues Simulator

Teacher: Prof. Ioan Salomie
Teacher Assistant: Dr. Viorica Chifu
Student: Domokos Robert - Ștefan
Group: 30421

Content

1. Objective

The main objective of this assignment is to design and implement a simulator that allows customers to be divided into queues to particular queues based on the time they arrive at the queue and the time they wait while they are processed. While the simulation is running the user should not interact with it, it should work on its own.

Queues are frequently seen both in the models and in the real world. Their main task is to provide a place for a “client” / ”customer” to wait before receiving a “service”. The operation of a queue based system is interested in minimizing the time of waiting for each of its “clients” / “customers”.

The secondary objectives of the assignment are:

- Using a GUI for introducing the inputs and then
- Save the log of events in a .txt file;
- Displaying the simulation results in the GUI, and their evolution;
- Use one thread for each server;
- Find the average waiting time, average service time and the peak hour, for a much easier understanding of the output;
- Generate random values for a list of created students;

2. Analysis

- Modeling and requirements

The requirements of this assignment consists of managing a dynamic number of threads that will be launched to process in parallel the clients, another thread that will be launched to hold the simulation time and distribute each “client” to the queue with the smallest waiting time.

The input data that will be inserted in the simulation must be expressed by:

- A number of clients (represented by the letter N);
- A number of queues (represented by the letter Q);
- The simulation interval (represented by *tsimulationMAX*);
- The minimum and maximum arrival time, so that the arrival time is bounded between the minimum and maximum (represented by *tarrival*);
- The minimum and maximum service time, so that the service time is bounded between the minimum and maximum (represented by *tservice*);

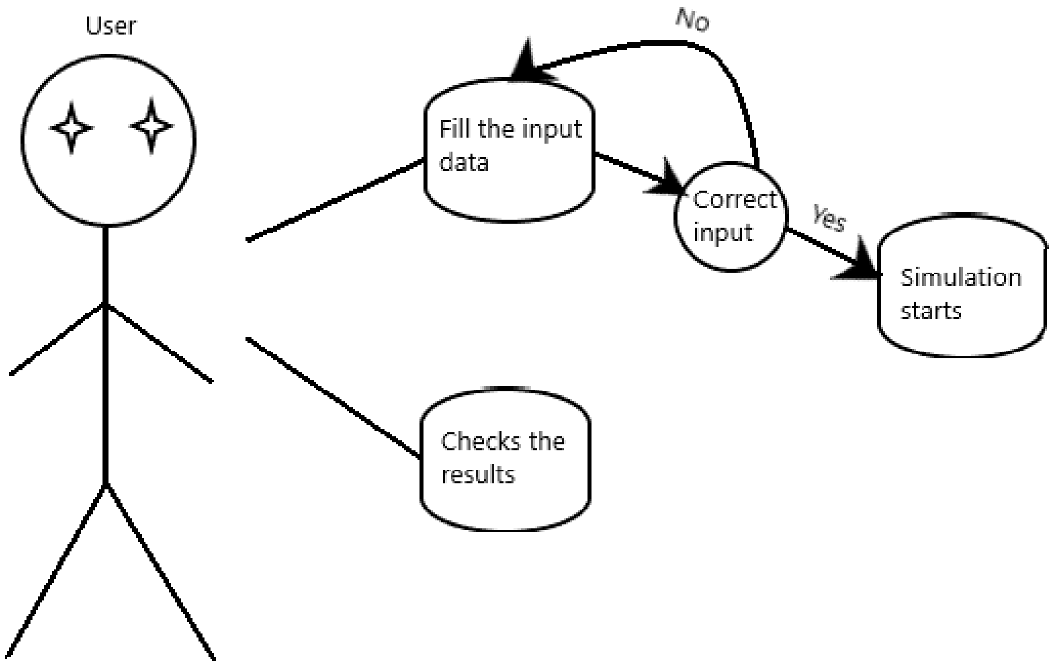
- Scenarios and use case

The following data that should be considered as input values for the simulator that the user will insert is the only correct use case scenario that the application has:

- The number of clients as an integer (N);
- The number of queues as an integer (Q);
- The number of the simulation interval as an integer (*tsimulationMAX*);
- The values of the maximum and minimum arrival time, as the format [integer, integer] (*tarrival*)
- The values of the maximum and minimum service time, as the format [integer, integer] (*tservice*)

Any other ways in which the data might be inserted by the user will consist of just wrong use case scenarios.

- Use case diagram



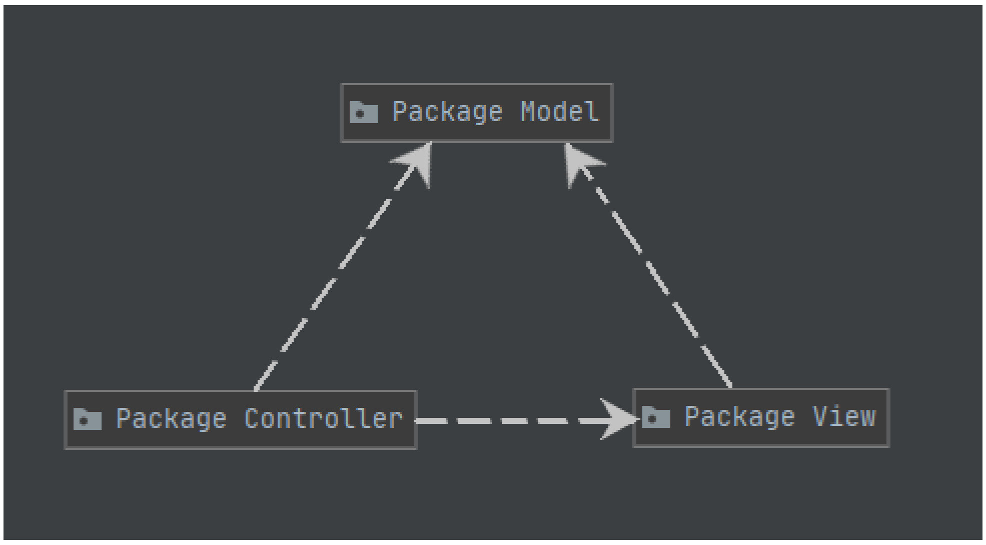
3. Design

The design part of the project/assignment consists of the design decisions, UML diagrams, relationships, data structures, class design, packages and user interface;

- Design decision

Like in any other project that requires a graphic interface, the design of this projects is constructed around an MVC implementation, which separates the controller, model and view into diverse classes.

- UML Diagram and relationships



As already mentioned, the design follows the MVC structure being divided into 3 main packages:

- Model which consists of the classes: Task, Scheduler, Server, Strategy, SelectionPolicy, ConcreteStrategyQueue, ConcreteStrategyTime;

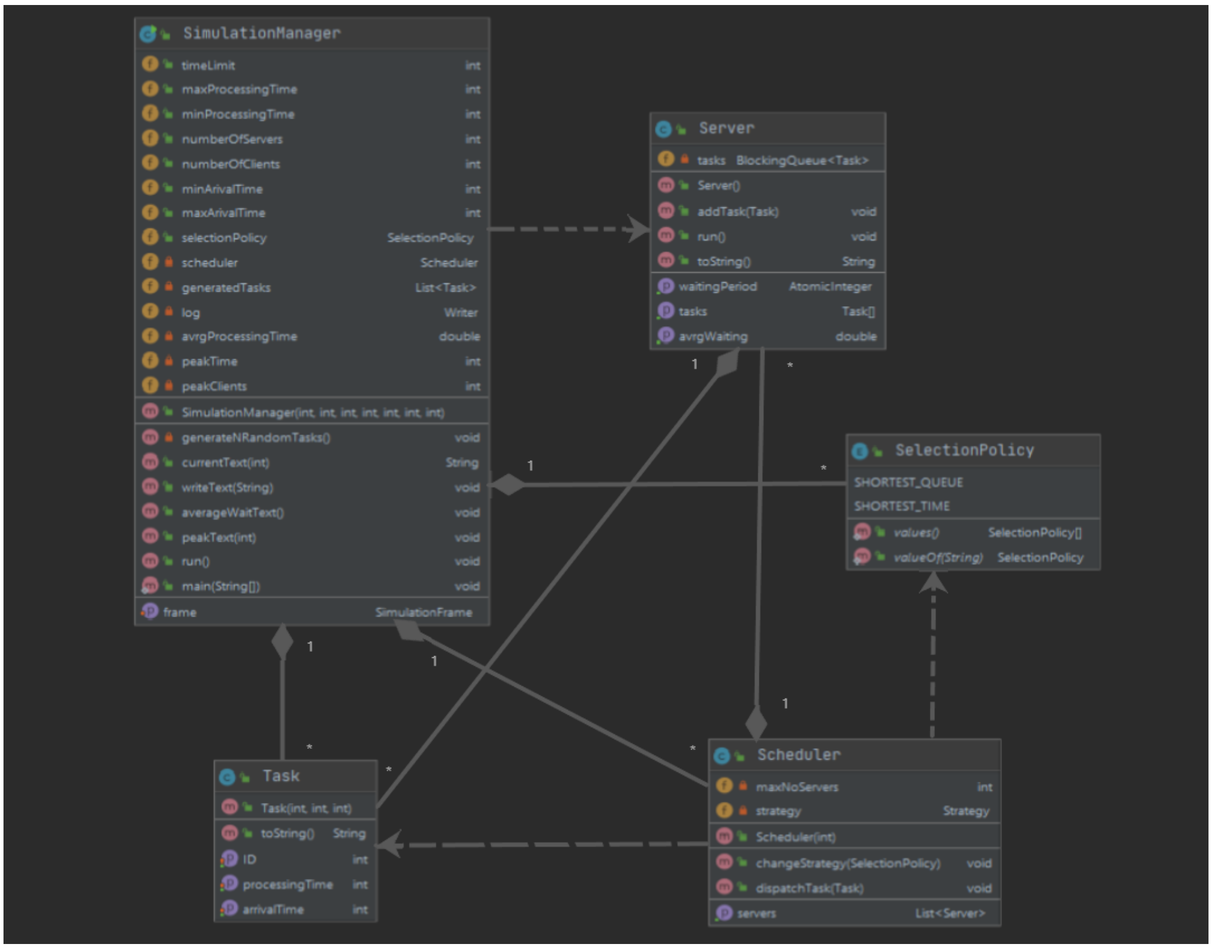
- Controller which only consists of the class SimulationManager which opens the user interfaces and starts the simulation;
- View which consists of the classes that forms the user interfaces of the project: GUI and SimulationFrame;

The relationships between these 3 packages are visible in the UML Diagram as the dependency between Controller and Model, Controller and View, View and Model.

- Data structures

The data structures used in this project are: int, double, String, ArrayList, BlockingQueue, AtomicInteger and Threads. Of course, the use of threads was a requirement in this assignment, but the most unusual data structures used are BlockingQueue and AtomicInteger that were an important aspect in implementing thread safe queues and safe addition between these threads.

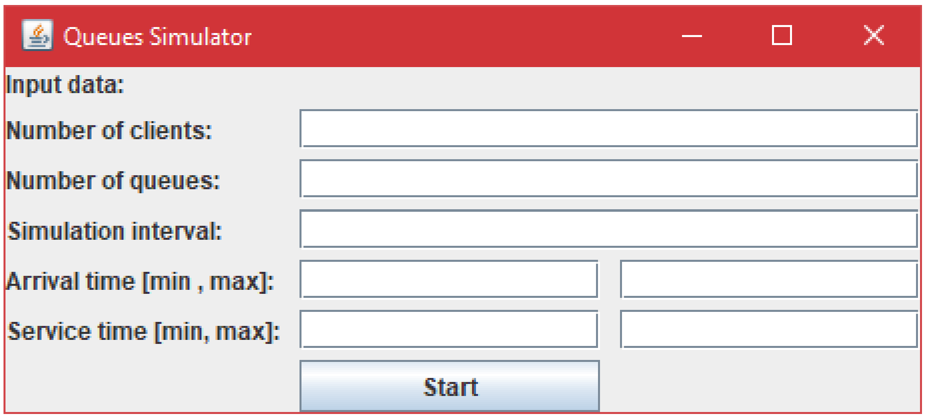
- Class design



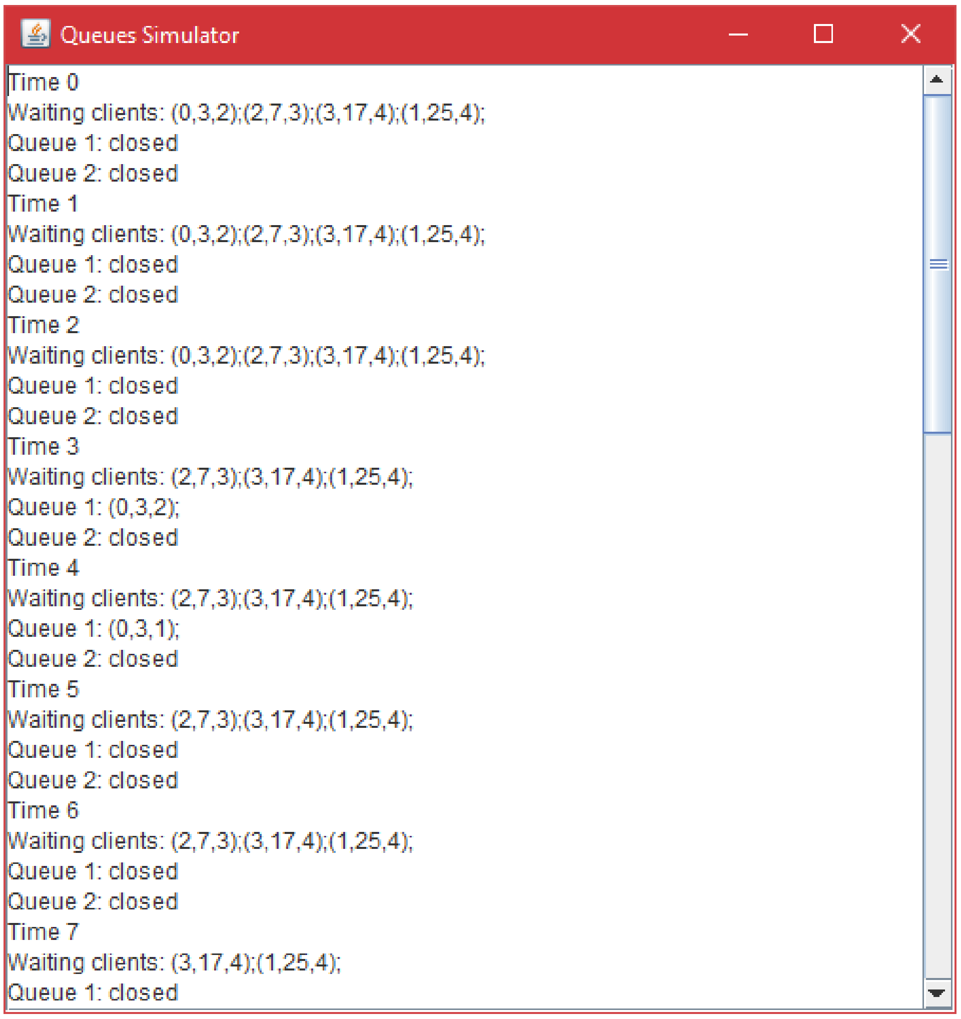
The only class that outstands the others in the class design is the SimulationManager. Being a class from the Controller package, it starts the interfaces and not only that, but it is used for keeping a safe order between threads and starts the simulation.

- User Interface

This assignment required 2 user interfaces for both, input data and visualizing the output data, or so called “log of events”.
The input data user interface is represented by the GUI class which has a field for each required input, and a “Start” button that pressed starts the simulation and the output data interface.



The output data user interface is the SimulationFrame which consists of a panel on which are printed the queues, waiting clients, the time and at the end, the average service time, peak hour and the average waiting time.



```
Time 0
Waiting clients: (0,3,2);(2,7,3);(3,17,4);(1,25,4);
Queue 1: closed
Queue 2: closed
Time 1
Waiting clients: (0,3,2);(2,7,3);(3,17,4);(1,25,4);
Queue 1: closed
Queue 2: closed
Time 2
Waiting clients: (0,3,2);(2,7,3);(3,17,4);(1,25,4);
Queue 1: closed
Queue 2: closed
Time 3
Waiting clients: (2,7,3);(3,17,4);(1,25,4);
Queue 1: (0,3,2);
Queue 2: closed
Time 4
Waiting clients: (2,7,3);(3,17,4);(1,25,4);
Queue 1: (0,3,1);
Queue 2: closed
Time 5
Waiting clients: (2,7,3);(3,17,4);(1,25,4);
Queue 1: closed
Queue 2: closed
Time 6
Waiting clients: (2,7,3);(3,17,4);(1,25,4);
Queue 1: closed
Queue 2: closed
Time 7
Waiting clients: (3,17,4);(1,25,4);
Queue 1: closed
```

4. Implementation

- Task.java

This class has as private parameters the arrivalTime, processingTime and ID that each customer has and the easily generated constructor, getters and setters. It also has a toString() method that helps us print the customers/ clients in the required way.

- Server.java

This class has as private parameters the waitingPeriod as an AtomicInteger, tasks which is a BlockingQueue list of tasks, avrgWaiting which is a double that saves the average waiting time and the getters and a constructor that initializes the parameters. It also has an addTask method that adds a new task to the list while computing the average waiting time, a toString() method and the run() method which “sleeps”(waits) for a task to be accessible and then waits until it’s over. This main function takes out the task after its waiting period runs out.

```
@Override
  public void run() {
    while(true){
      if(tasks.isEmpty())
      {
        try {
          Thread.sleep(1000);
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
        continue;
      }
      Task t = null;
      try {
        t = tasks.peek();
        Thread.sleep(1000*t.getProcessingTime());
      } catch (InterruptedException e) {
        e.printStackTrace();
      }

      waitingPeriod.getAndAdd(-t.getProcessingTime());
      try {
        tasks.take();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }

    }

  }
```

- Strategy.java

This class is an Interface that helps us make use of the ConcreteStrategyQueue and the ConcreteStrategyTime for modeling both strategies.

- ConcreteStrategyQueue.java

This class overrides the addTask() method from Strategy so that we can use the shortest queue method.

```
public void addTask(List<Server> servers, Task t) {
      Server min = servers.get(0);
      int l, lmin;
      lmin = min.getTasks().length;
      for(Server s: servers){
         l = s.getTasks().length;
         if(l < lmin)
         {
            lmin = l;
            min = s;
         }
      }

      min.addTask(t);
   }
```

- ConcreteStrategyTime.java

This class overrides the addTask() method from Strategy so that we can use the shortest time method.

```
public void addTask(List<Server> servers, Task t) {
      Server min = servers.get(0);
      int time, tmin;
      tmin = min.getWaitingPeriod().intValue();
      for(Server s: servers){
         time = s.getWaitingPeriod().intValue();
         if(time < tmin)
         {
            tmin = time;
            min = s;
         }
      }

      min.addTask(t);
   }
```

- SelectionPolicy.java

This class is an enumeration that helps us choose which Strategy will be used.

- Scheduler.java

This class has as private parameters servers a list of servers(List<Server>), the maxNoServers which is the maximum number of servers and a strategy. The role of this class is to create for each server object a thread with it, implemented in the constructor. It also has the method changeStrategy(SelectionPolicy policy) that maintains the instance of the strategies.

- SimulationManager.java

This is the largest class from this implementation and it has as parameters:

```
    public int timeLimit;
    public int maxProcessingTime;
    public int minProcessingTime;
    public int numberOfServers;
    public int numberOfClients;
    public int minArivalTime;
    public int maxArivalTime;
    public SelectionPolicy selectionPolicy = SelectionPolicy.SHORTEST_TIME;
    private Scheduler scheduler;
    private SimulationFrame frame;
    private List<Task> generatedTasks;
    private Writer log;
    private double avrgProcessingTime;
    private int peakTime;
    private int peakClients = 0;
```

This class also has a constructor which initializes the parameters, but also calls the generateNrandomTasks(); method to generate the task and computes the averageProcessingTime by also calling its method.

```
private void generateNRandomTasks(){
        generatedTasks= new ArrayList<>();
        Random r = new Random();
        for(int i=0;i<numberOfClients;i++){
            Task t = new Task(minArivalTime+r.nextInt(maxArivalTime-minArivalTime+1),minProcessingTime+
r.nextInt(maxProcessingTime-minProcessingTime+1),i);
            generatedTasks.add(t);
        }
        Collections.sort(generatedTasks, Comparator.comparingInt(Task::getArrivalTime));
    }
```

Other important methods implemented and used here are the ones that print the outputs in the wanted way but also computing the required peak hour and average waiting time:

- currentText(int currentTime);
- writeText(String s);
- averageWaitText();
- peakText(int currentTime);

Each of these methods is called in the run() method.

Last but not least this class has also the overridden method run() that runs on its own thread and while the current time is smaller than the time limit, at each step the tasks are processed and subtracts one, from the processing time sleeping a second after that.

```
@Override
    public void run() {
        int currentTime = 0;
        while( currentTime < timeLimit){
            List<Task> delClient= new ArrayList<>();
            for(int i=0;i<generatedTasks.size();i++){
                Task t = generatedTasks.get(i);
                if (currentTime == t.getArrivalTime()){
                    scheduler.dispatchTask(t);
                    delClient.add(t);
                }
            }
            generatedTasks.removeAll(delClient);
            writeText(currentText(currentTime));
            peakText(currentTime);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            currentTime++;
        }
        averageWaitText();
        try {
            log.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

This class is also the main class which creates the first frame of the user interface.

- GUI.java

This class represents the first user interface where the user must put the input data. It also has a button that will start the second interface and also the simulation. Here is also the listener for the Start button of the interface. The private parameters that represent the text fields button and the panel are:

```
private JTextField minService;
private JTextField nrClients;
private JTextField simInterval;
private JTextField minArrival;
private JButton Start;
private JTextField maxArrival;
private JTextField maxService;
private JTextField nrQueues;
private JPanel FirstPanel;
```

- SimulationFrame.java

This class represents the second user interface which allows the user to check the results of the simulation in real time, second by second.

5. Results

As for the results part for this assignment we were required to test our program implementation with 3 given tests. The results of these tests are saved also in the Log.txt file after running each of them and also on the text files created T1.txt, T2.txt, T3.txt.

- Test 1:

Number of clients: 4
Number of queues: 2
Simulation interval: 60 seconds
Minimum and maximum arrival time: [2, 30]
Minimum and maximum service time: [2, 4]
Results in T1.txt;

- Test 2:

Number of clients: 50
Number of queues: 5
Simulation interval: 60 seconds
Minimum and maximum arrival time: [2, 40]
Minimum and maximum service time: [1, 7]
Results in T2.txt;

- Test 3:

Number of clients: 1000
Number of queues: 20
Simulation interval: 200 seconds
Minimum and maximum arrival time: [10, 100]
Minimum and maximum service time: [3, 9]
Results in T3.txt;

6. Conclusions

In conclusion, the most important part of this assignment, that, I can truly say that it improved my skills not only as a student, but as a person who wants to learn more and more programming techniques, it was working with threads and learning about how they work in Java and other programming languages. Working with threads was a little bit challenging, but it was a great exercise in seeing how important thread safety is.

The assignment also helped me to better understand how queues actually work by using the 2 strategies already mentioned in this documentation:

- The shortest time strategy
- The shortest queue strategy

One improvement that could make this application usable by the people, in order to not waste time in a real life queue could be that the application will use a database of clients with not only the ID as information, but informations as: first and last name, purpose, priority and many more, instead of using random generated clients.

Another improvement that I think would make a change would be developing more strategies, for example a strategy that prioritizes the clients that are old or have health problems or even pregnant women.

7. Bibliography

http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html
http://www.tutorialspoint.com/java/util/timer_schedule_period.htm
http://www.javacodegeeks.com/2013/01/java-thread-pool-example-using-executors-and-threadpoolexecutor.htm
l
https://www.w3schools.com/java/java_threads.asp
https://stackoverflow.com/questions/15668032/how-to-call-the-overridden-method-of-a-superclass
https://www.javatpoint.com/java-thread-run-method
https://stackoverflow.com/questions/18091172/how-do-i-create-2-frames-in-java-and-link-them-together