DOCUMENTATION

ASSIGNMENT *2*

STUDENT NAME: RUNCAN PAUL-MARIUS
GROUP: e_30422

CONTENTS

1. Assignment Objective

Main Objective:

- Design and implement an application aiming to analyze queuing-based systems by (1) simulating a series of N clients arriving for service, entering Q queues, being served and finally leaving the queues, and (2) computing the average waiting tine, average service time and peak hour.

Secondary Objectives:

- Analyze the problem and identify the requirements
- Design the simulation application
- Implement the simulation application
- Test the simulation application

2. Problem Analysis, Modeling, Scenarios, Use Cases

Functional requirements:

The simulation manager should:

- allow the user to insert the necessary data for the simulation
- allow the user to select the strategy type which he wants to use (time strategy/shortest queue strategy)
- print an error message if the data is incorrect.
- Display on the GUI the simulation in real time
- Print the results in a text file.

Non-Functional requirements:

The simulation manager should:

- be intuitive, it must be clear for the user where the data should be inserted and it should be clear what the output means.
- be reliable, failures should appear only if the bounds of Integer number aren't respected.
- be maintainable, issues should be easily fixed because the code is well-structured and is based on simple algorithms.
- be secure, the application should be secure, having all fields private and inaccessible from the exterior.

Use Cases:

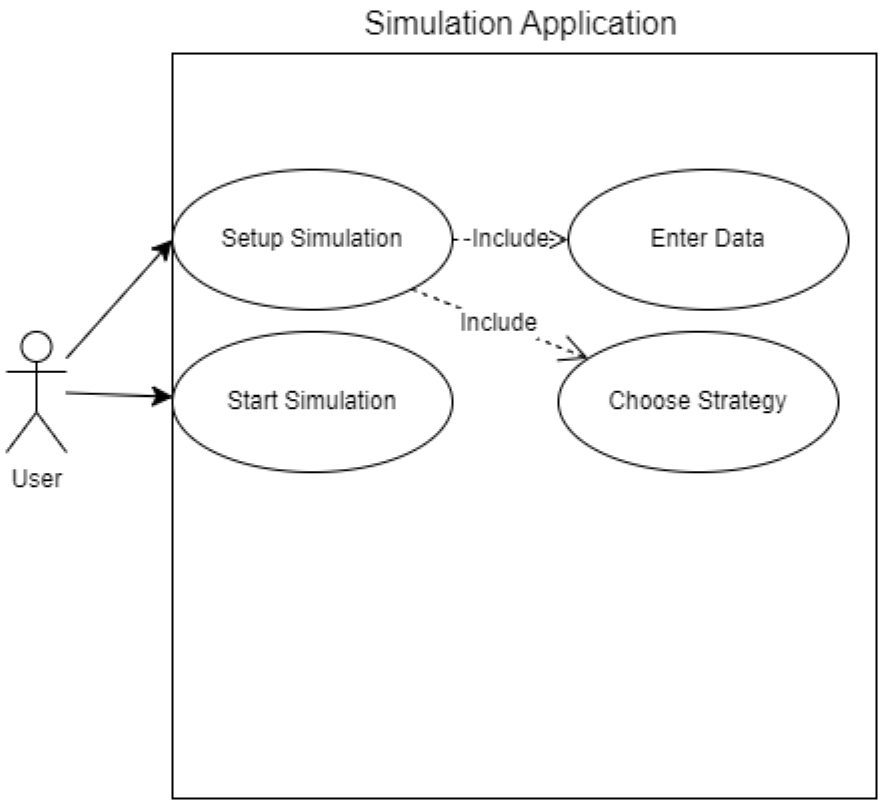


Figure 1. Use Case Diagram

Primary actor: User

Main Succes Scenario:

1. The user inserts correct data for the simulation in the fields from the Graphical User Interface.
2. The user selects one of the strategies (time strategy/shortest queue strategy).
3. The user presses the "Start" button.
4. The simulation start with n empty queues(n=the number of queues entered in the graphical user interface).
5. The GUI updates in real-time simulating the queues. If a client has the arrival time equal to the current time of the simulation he enters the queue. After the processing time of that client is passed, he is deleted from the queue.
6. After the simulation time has passed, the GUI will show the Average Waiting Time, Average Processing Time and Peek Hour.
7. All the data will be also stored in a .txt file.

Alternative Sequence:

1. The user inserts wrong data(maximum arrival time < minimum arrival time, maximum service time < minimum service time).
2. The application displays an error message.
3. The scenario return to step 1.

3. Design

Level 1: Overall System Design

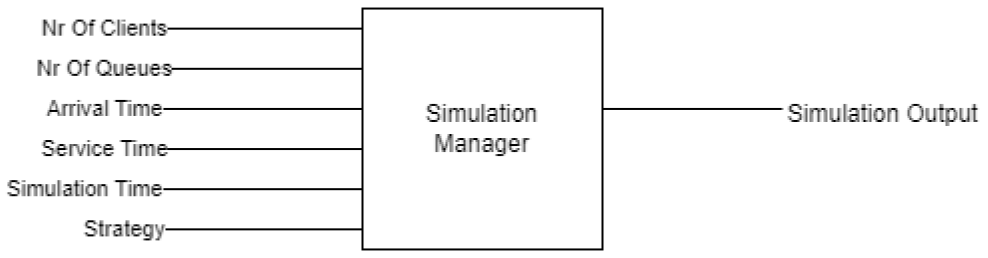


Figure 1. System Black Box

The system consists of 6 inputs (Number Of Clients, Number Of Queues, Arrival Time, Service Time, Simulation Time, Strategy) and an Output.

- Number Of Clients: the number of tasks that will be generated and the queues(servers) must handle;
- Number Of Queues: the number of servers that will be created and usable.
- Arrival: it has 2 input ports: first one is the minimum arrival time and the second one is the maximum arrival time. This gives the interval in which the arrival time of each client is generated.
- Service: it has 2 input ports: first one is the minimum service time and the second one is the maximum service time. This gives the interval in which the service time of each client is generated.
- Simulation Time: the duration of the simulation.
- Simulation Output: (1) during the simulation time it will display in real-time the queues and the clients that haven't arrived yet at the queues, (2) after the simulation it will display the average waiting time, average processing time and peek hour.

Level 2: Design in sub-systems/packages

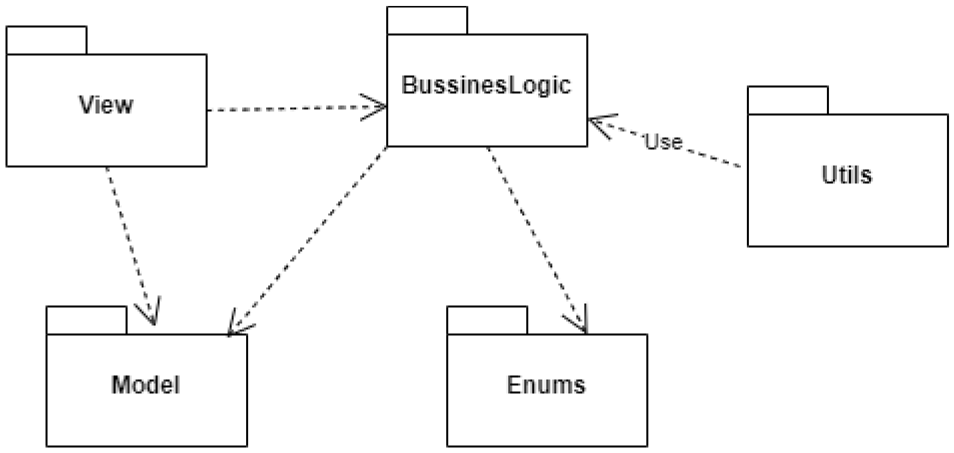


Figure 2. Package Diagram

View: contains the classes that implement the Graphical User Interface

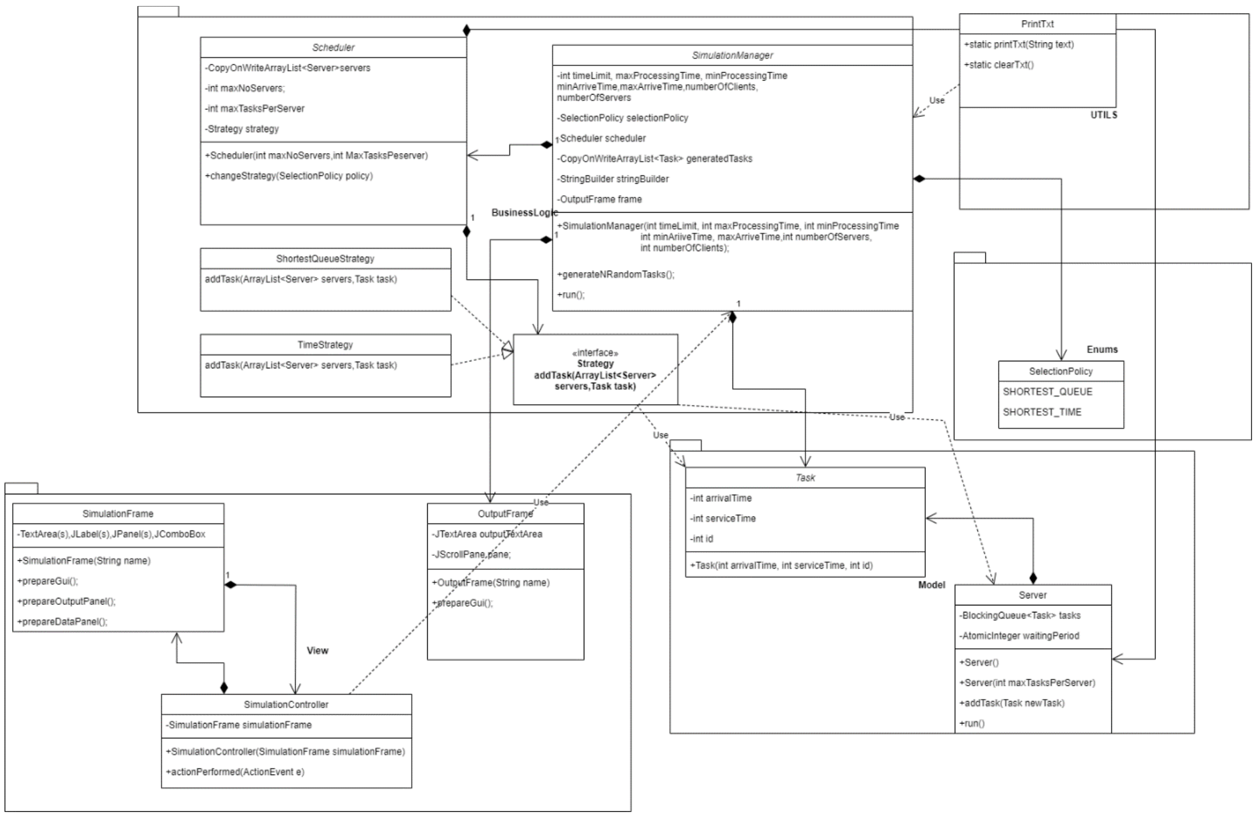
BusinessLogic: contains the classes that handle the simulation.

- Simulation Manager: creates a new Scheduler,generates the tasks, and handles the real-time output and queues.
- Scheduler: creates the servers necessary for the simulation.
- Strategy: is an interface that contains the addTask method which handles adding a task to the correct server.
- ShortestQueueStrategy: implements the Strategy interface and Overrides the addTask method such that the task will be added to the shortest queue;
- TimeStrategy: implements the Strategy interface and Overrides the addTask method such that the task will be added to the shortest waiting time queue;

Model:contains the classes that model the Task(Client) and Server(Queue) .

Enums: contains the enumeration of the strategies.

Level 3: Division into classes/Class Diagram.



Scheduler
-CopyOnWriteArrayList<Server>servers
-int maxNoServers;
-int maxTasksPerServer
-Strategy strategy
+Scheduler(int maxNoServers,int MaxTasksPeserver)
+changeStrategy(SelectionPolicy policy)
SimulationManager
-int timeLimit, maxProcessingTime, minProcessingTime minArriveTime,maxArriveTime,numberOfClients, numberOfServers
-SelectionPolicy selectionPolicy
-Scheduler scheduler
-CopyOnWriteArrayList<Task> generatedTasks
-StringBuilder stringBuilder
-OutputFrame frame
+SimulationManager(int timeLimit, int maxProcessingTime, int minProcessingTime int minAriiveTime, maxArriveTime,int numberOfServers, int numberOfClients);
+generateNRandomTasks();
+run();
BusinessLogic
ShortestQueueStrategy
addTask(ArrayList<Server> servers,Task task)
TimeStrategy
addTask(ArrayList<Server> servers,Task task)
«interface»
Strategy
addTask(ArrayList<Server> servers,Task task)
PrintTxt
+static printTxt(String text)
+static clearTxt()
UTILS
Use
Enums
SelectionPolicy
SHORTEST_QUEUE
SHORTEST_TIME
Task
-int arrivalTime
-int serviceTime
-int id
+Task(int arrivalTime, int serviceTime, int id)
Model
Server
-BlockingQueue<Task> tasks
-AtomicInteger waitingPeriod
+Server()
+Server(int maxTasksPerServer)
+addTask(Task newTask)
+run()
SimulationFrame
-TextArea(s),JLabel(s),JPanel(s),JComboBox
+SimulationFrame(String name)
+prepareGui();
+prepareOutputPanel();
+prepareDataPanel();
OutputFrame
-JTextArea outputTextArea
-JScrollPane pane;
+OutputFrame(String name)
+prepareGui();
View
SimulationController
-SimulationFrame simulationFrame
+SimulationController(SimulationFrame simulationFrame)
+actionPerformed(ActionEvent e)
1

Figure 3. Class Diagram

Used Data Structures:

The data structure used is a BlockingQueue because it is Thread Safe. Each Queue is considered to be a server and each client joins one queue when his arrival time is equal to the current time of the simulation.

Also, AtomicInteger has been used because it is also Thread Safe.

GUI Design:

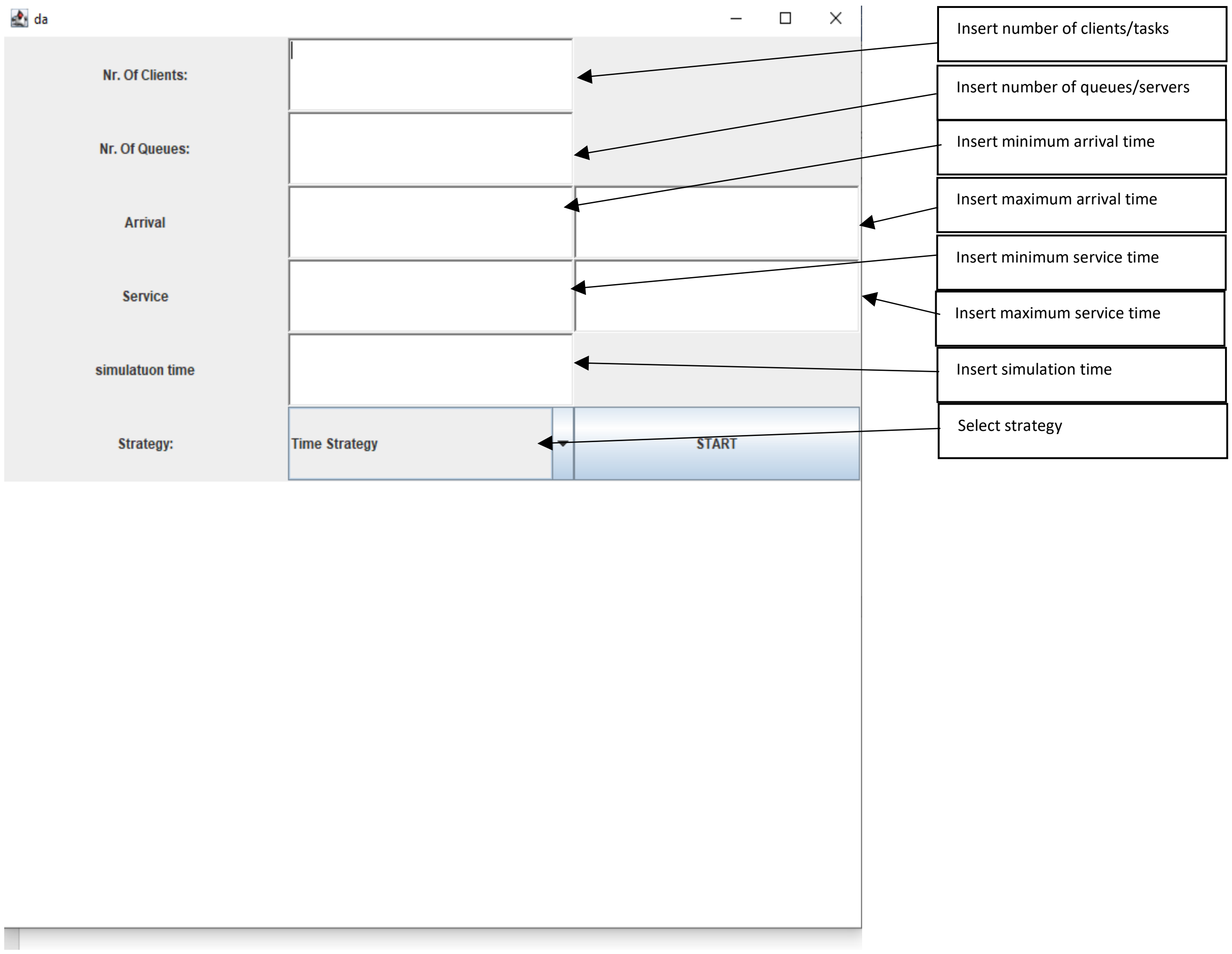
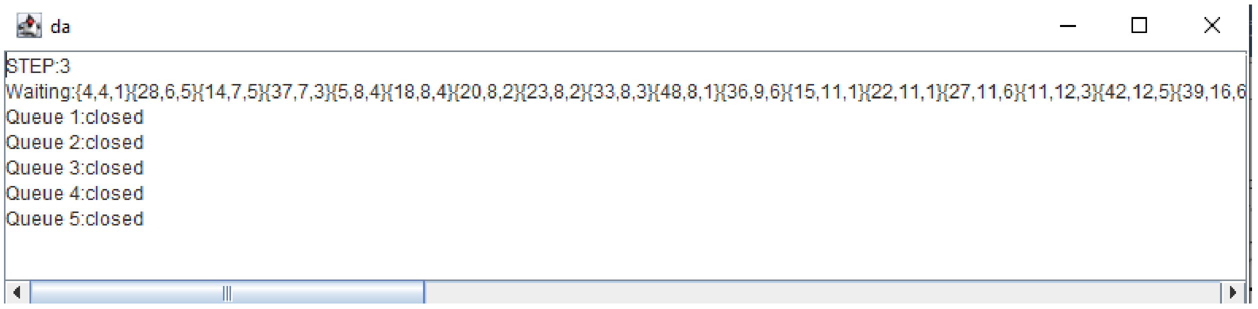

da
Nr. Of Clients:
Nr. Of Queues:
Arrival
Service
simulatuon time
Strategy:
Time Strategy
START
Insert number of clients/tasks
Insert number of queues/servers
Insert minimum arrival time
Insert maximum arrival time
Insert minimum service time
Insert maximum service time
Insert simulation time
Select strategy

Figure 5. GUI

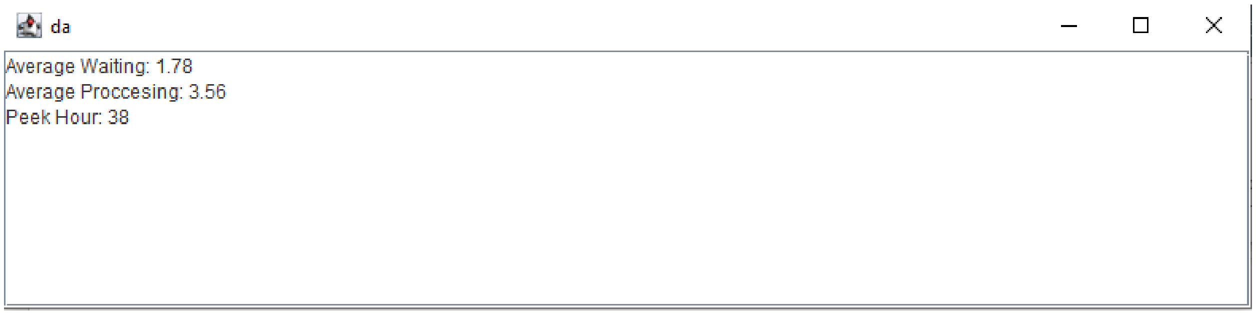
The GUI will look like this during the simulation time:



```
STEP:3
Waiting:{4,4,1}{28,6,5}{14,7,5}{37,7,3}{5,8,4}{18,8,4}{20,8,2}{23,8,2}{33,8,3}{48,8,1}{36,9,6}{15,11,1}{22,11,1}{27,11,6}{11,12,3}{42,12,5}{39,16,6
Queue 1:closed
Queue 2:closed
Queue 3:closed
Queue 4:closed
Queue 5:closed
```

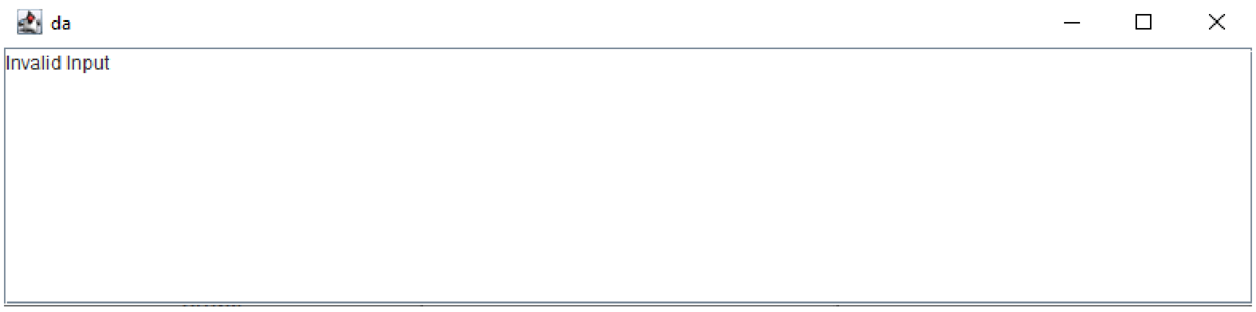
The GUI refreshes in real time allowing the user to see the evolution of the simulation at every moment.

At the end the user will see the Average Waiting time, Average Processing Time and Peek Hour:



```
Average Waiting: 1.78
Average Proccesing: 3.56
Peek Hour: 38
```

In case of an error this message will be shown:



```
Invalid Input
```

4.Implementation

Classes:

View Package classes:

SimulationFrame:

Fields: contains all the components of the Frame created for the GUI(Labels, ComboBox, Button, TextFields) and an instance of the controller for controlling the behavior of the application.

Methods: contains the methods used for setting up the GUI:

prepareGui(): creates a grid layout with 2 rows and 1 columns in which the resultPanel and the numbersPanel are placed.

prepareDataPanel(): creates a grid layout with 6 rows and 3 columns. First row is used for entering the number of clients. The second row is used for entering the number of queues. The third row is used for entering the minimum arrival time and the maximum arrival time. The fourth row is used for entering the minimum service time and the maximum service time. The fifth row is used for entering the simulation time. The sixth row is used for selecting the strategy and for the "Start" button.

SimulationController:

Fields: contains an attribute of type SimulationFrame. The view that the controller must handle.

Methods:

actionPerformed(ActionEvent e): - if the "Start" button is pressed in the GUI, this method validates the input and if it is correct it creates a new SimulationManager for the inputs given. If the input is not ok, it throws an error and writes the error message in the OutputFrame.

OutputFrame:

Fields: constains one TextArea field where the output is printed.

Methods:

prepareGui(): creates a view that contains a TextArea where the output will be printed

BusinessLogic Package classes:

Scheduler:

Fields:

```
private CopyOnWriteArrayList<Server> servers;
private int maxNoServers;
private int maxTasksPerServer;
private Strategy strategy;
```

Methods:

```
public Scheduler(int maxNoServers, int maxTasksPerServer){
    this.servers=new CopyOnWriteArrayList<>();
    this.maxNoServers=maxNoServers;
    this.maxTasksPerServer=maxTasksPerServer;
    for(int i=0;i<maxNoServers;i++){
        Server server = new Server(maxTasksPerServer);
        Thread t = new Thread(server);
        t.start();
        this.servers.add(server);
    }
}
```

The constructor creates the necessary number of servers and starts the threads.

```
public void changeStrategy(SelectionPolicy policy){
    if(policy == SelectionPolicy.SHORTEST_TIME)
        strategy = new TimeStrategy();
    if(policy == SelectionPolicy.SHORTEST_QUEUE)
        strategy = new ShortestQueueStrategy();
}
```

The changeStrategy is used when the user wants to switch the strategy used when allocating clients to the queues.

Strategy:

It is an interface that has one method that adds the client to the correct queue.

ShortestQueueStrategy:

Methods:

```
public int addTask( CopyOnWriteArrayList<Server> servers, Task task ) {
    Server shortQueueServer=new Server();
    int shortestSize=10000;
    for(Server server:servers){
        if(shortestSize>server.getTasks().size()){
            shortQueueServer=server;
            shortestSize=server.getTasks().size();
        }
    }
    int waitingPer=shortQueueServer.getWaitingPeriod().get();
    shortQueueServer.addTask(task);
    return waitingPer;
}
```

The addTask method overrides the method from Strategy interface and it iterates through the queues and adds the client to the one that is the smallest in size.

TimeStrategy:

Methods:

```
public int addTask( CopyOnWriteArrayList<Server> servers, Task task ) {
    Server minTimeServer=new Server();
    int minWaitingTime=10000;
    for(Server server:servers){
        if(minWaitingTime>server.getWaitingPeriod().get()){
            minTimeServer=server;
            minWaitingTime=server.getWaitingPeriod().get();
        }
    }
    int waitingPer=minTimeServer.getWaitingPeriod().get();
    minTimeServer.addTask(task);
    return waitingPer;
}
```

The addTask method overrides the method from Strategy interface and it iterates through the queues and adds the client to the one that has the smallest waiting period.

SimulationManager:

Fields:

```
private int timeLimit = 60;
private int maxProcessingTime = 4;
private int minProcessingTime = 2;
private int minArriveTime = 0;
private int maxArriveTime = 10;
private int numberOfServers = 2;
private int numberOfClients = 4;
private SelectionPolicy selectionPolicy = SelectionPolicy.SHORTEST_TIME;

private Scheduler scheduler;
private CopyOnWriteArrayList<Task> generatedTasks;
```

```java
private StringBuilder stringBuilder;
private OutputFrame frame;
```

Methods:

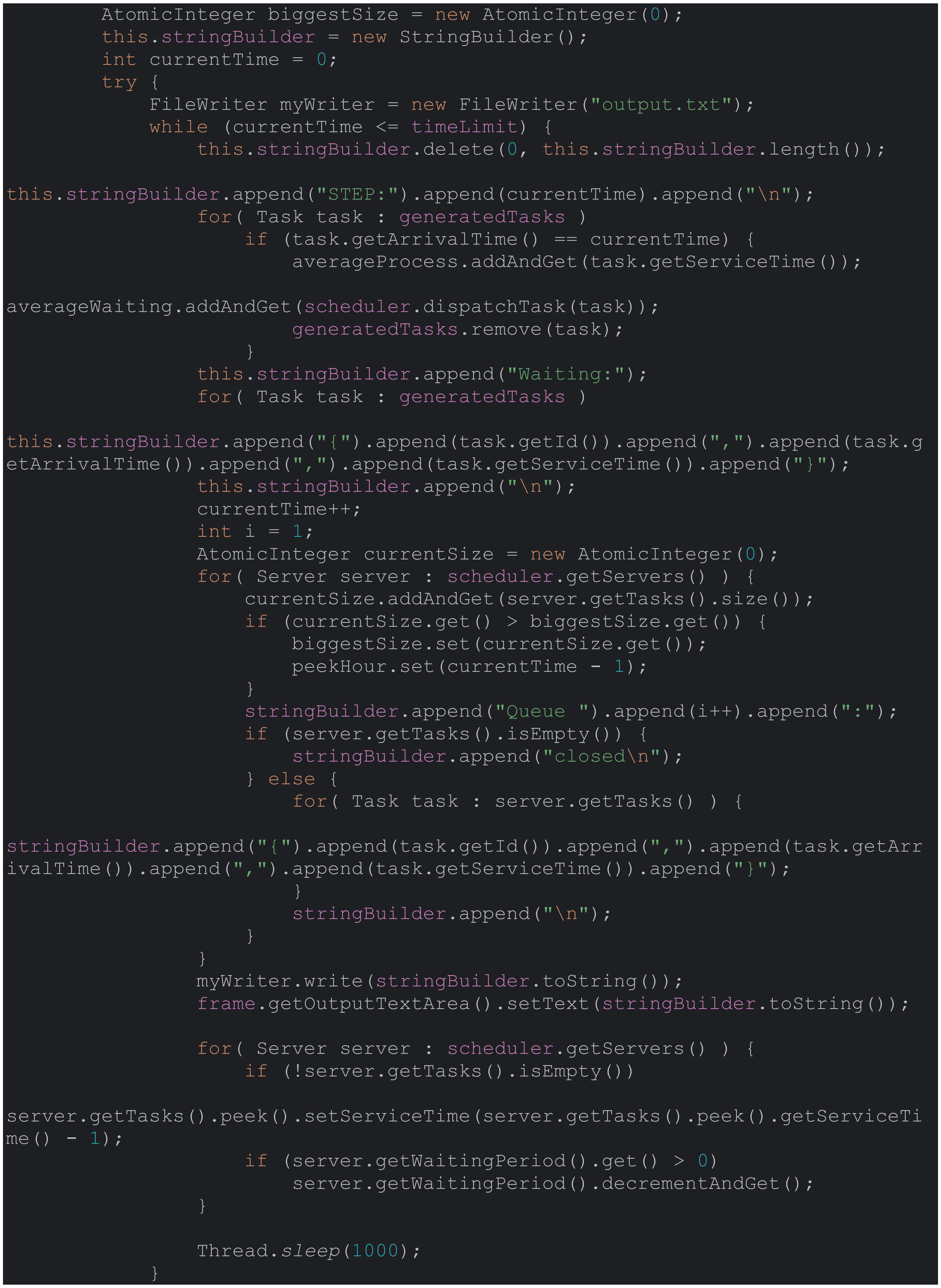
```java
public SimulationManager( int timeLimit, int maxProcessingTime, int
minProcessingTime, int minArriveTime, int maxArriveTime, int numberOfServers,
int numberOfClients ) {
    this.timeLimit = timeLimit;
    this.maxProcessingTime = maxProcessingTime;
    this.minProcessingTime = minProcessingTime;
    this.minArriveTime = minArriveTime;
    this.maxArriveTime = maxArriveTime;
    this.numberOfServers = numberOfServers;
    this.numberOfClients = numberOfClients;
    scheduler = new Scheduler(numberOfServers, 100);
    generatedTasks = generateNRandomTasks();
    if(!generatedTasks.isEmpty())
    {frame = new OutputFrame("da");
    frame.setVisible(true);}
    //frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
}
```

The constructor sets the fields, creates a scheduler for the given fields and generates the necessary number of random Tasks. It also initializes the output frame.

```java
private CopyOnWriteArrayList<Task> generateNRandomTasks() {
    Random random = new Random();
    CopyOnWriteArrayList<Task> tasks = new CopyOnWriteArrayList<>();
        for( int i = 1; i <= numberOfClients; i++ ) {
            Task task = new Task(random.nextInt(minArriveTime,
maxArriveTime), random.nextInt(minProcessingTime, maxProcessingTime), i);
            tasks.add(task);
        }
        Collections.sort(tasks, new Comparator<Task>() {
            @Override
            public int compare( Task o1, Task o2 ) {
                if (o1.getArrivalTime() == o2.getArrivalTime())
                    return 0;
                return o1.getArrivalTime() < o2.getArrivalTime() ? -1 : 1;
            }
        });
```

This method is used for generating the random tasks. It generates numberOfClients tasks with the given bounds for the arrival time and service time.

```java
public void run() {
    if(!this.generatedTasks.isEmpty()) {
        AtomicInteger averageWaiting = new AtomicInteger(0);
        AtomicInteger averageProcess = new AtomicInteger(0);
        AtomicInteger peekHour = new AtomicInteger(0);
```



```
        AtomicInteger biggestSize = new AtomicInteger(0);
        this.stringBuilder = new StringBuilder();
        int currentTime = 0;
        try {
            FileWriter myWriter = new FileWriter("output.txt");
            while (currentTime <= timeLimit) {
                this.stringBuilder.delete(0, this.stringBuilder.length());

this.stringBuilder.append("STEP:").append(currentTime).append("\n");
                for( Task task : generatedTasks )
                    if (task.getArrivalTime() == currentTime) {
                        averageProcess.addAndGet(task.getServiceTime());

averageWaiting.addAndGet(scheduler.dispatchTask(task));
                        generatedTasks.remove(task);
                    }
                this.stringBuilder.append("Waiting:");
                for( Task task : generatedTasks )

this.stringBuilder.append("{").append(task.getId()).append(",").append(task.g
etArrivalTime()).append(",").append(task.getServiceTime()).append("}");
                this.stringBuilder.append("\n");
                currentTime++;
                int i = 1;
                AtomicInteger currentSize = new AtomicInteger(0);
                for( Server server : scheduler.getServers() ) {
                    currentSize.addAndGet(server.getTasks().size());
                    if (currentSize.get() > biggestSize.get()) {
                        biggestSize.set(currentSize.get());
                        peekHour.set(currentTime - 1);
                    }
                    stringBuilder.append("Queue ").append(i++).append(":");
                    if (server.getTasks().isEmpty()) {
                        stringBuilder.append("closed\n");
                    } else {
                        for( Task task : server.getTasks() ) {

stringBuilder.append("{").append(task.getId()).append(",").append(task.getArr
ivalTime()).append(",").append(task.getServiceTime()).append("}");
                        }
                        stringBuilder.append("\n");
                    }
                }
                myWriter.write(stringBuilder.toString());
                frame.getOutputTextArea().setText(stringBuilder.toString());

                for( Server server : scheduler.getServers() ) {
                    if (!server.getTasks().isEmpty())

server.getTasks().peek().setServiceTime(server.getTasks().peek().getServiceTi
me() - 1);
                    if (server.getWaitingPeriod().get() > 0)
                        server.getWaitingPeriod().decrementAndGet();
                }

                Thread.sleep(1000);
            }
            Double x = 1.0 * averageWaiting.get() / numberOfClients;
            Double y = 1.0 * averageProcess.get() / numberOfClients;
            String outputData = "Average Waiting: " + x.toString() + "\n" +
"Average Proccesing: " + y.toString() + "\n" + "Peek Hour: " +
peekHour.toString() + "\n";
            myWriter.write(outputData);
            frame.getOutputTextArea().setText(outputData);
            myWriter.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

The run method handles the queues. It decrements the service time for the front tasks each second. Also, it creates the output using the string builder and writes in the txt files when needed.

Model Package:

Task:

Fields:

```
    private int arrivalTime;
    private int serviceTime;
    private int id;
```

Methods:

```
public Task(int arrivalTime, int serviceTime, int id) {
    this.arrivalTime=arrivalTime;
    this.serviceTime=serviceTime;
    this.id=id;
}
```

Server:

Fields:

```
private BlockingQueue<Task> tasks;
private AtomicInteger waitingPeriod;
```

Methods:

```
public Server(){
    this.waitingPeriod=new AtomicInteger(0);
    this.tasks = new LinkedBlockingQueue<>();
}
public Server(int maxTasksPerServer){
    this.waitingPeriod=new AtomicInteger(0);
    this.tasks = new LinkedBlockingQueue<>(maxTasksPerServer);
}
```

```
public void addTask(Task newTask){
    this.tasks.add(newTask);
    this.waitingPeriod.addAndGet(newTask.getServiceTime());
    //System.out.println(waitingPeriod);
}
```

The method adds a client to the queue and increments the queue waiting period.

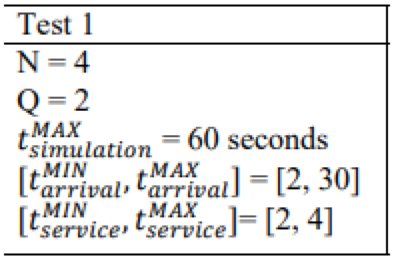
```
public void run(){
    while(true){
        try {
            if(!this.tasks.isEmpty()){
            Task nextTask=this.tasks.peek();
            Thread.sleep(nextTask.getServiceTime()*1000);
            nextTask = this.tasks.take();
            this.waitingPeriod.addAndGet(-nextTask.getServiceTime());}
            } catch (Exception e){
            e.printStackTrace();
        }
    }
}
```

The run method handles the moment when the client must leave the queue. It also pauses the thread for a duration equal to the service time of the client.

5.Results

Testing will be done through the GUI and through the output txt files which will save the entire simulation history for the last simulation.

For the tests given in the Assignment Requirements:

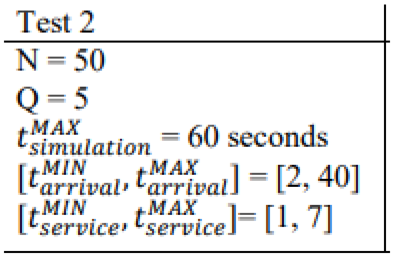


Test 1
N = 4
Q = 2
$t_{simulation}^{MAX}$ = 60 seconds
$[t_{arrival}^{MIN}, t_{arrival}^{MAX}]$ = [2, 30]
$[t_{service}^{MIN}, t_{service}^{MAX}]$= [2, 4]

The results for this test will be saved in the output1.txt

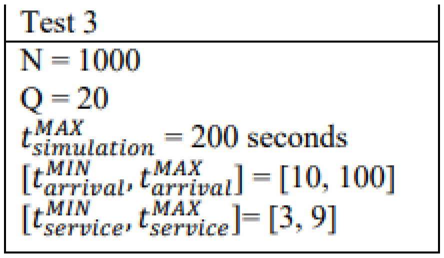
```
STEP:0
Waiting:{4,5,3}{1,12,2}{3,19,3}{2,23,3}
Queue 1:closed
Queue 2:closed
STEP:1
Waiting:{4,5,3}{1,12,2}{3,19,3}{2,23,3}
Queue 1:closed
Queue 2:closed
STEP:2
Waiting:{4,5,3}{1,12,2}{3,19,3}{2,23,3}
Queue 1:closed
Queue 2:closed
STEP:3
Waiting:{4,5,3}{1,12,2}{3,19,3}{2,23,3}
Queue 1:closed
Queue 2:closed
STEP:4
Waiting:{4,5,3}{1,12,2}{3,19,3}{2,23,3}
Queue 1:closed
Queue 2:closed
```

These are the first 4 steps of 1 simulation.



Test 2
N = 50
Q = 5
$t_{simulation}^{MAX}$ = 60 seconds
$[t_{arrival}^{MIN}, t_{arrival}^{MAX}]$ = [2, 40]
$[t_{service}^{MIN}, t_{service}^{MAX}]$= [1, 7]

The results will be saved in output2.txt.



Test 3
N = 1000
Q = 20
$t_{simulation}^{MAX}$ = 200 seconds
$[t_{arrival}^{MIN}, t_{arrival}^{MAX}]$ = [10, 100]
$[t_{service}^{MIN}, t_{service}^{MAX}]$= [3, 9]

The results will be saved in output3.txt.

6.Conclusions

In conclusion, the Simulation Manager has been implemented using a Thread based approach. Every queue uses its own thread. Putting the thread to sleep for a certain amount of time simulates the effect of time passing.

The Graphical User Interface was implemented using Java Swing.

Working on this project has helped me to develop my knowledge about threads, synchronized data structures and how to handle them. It also helped me in developing the real-time actualization of the interface.

7.Bibliography

[1] https://dsrl.eu/courses/pt/ -> PT_2024_A2_S1, PT_2024_A2_S2, Java Swing, Lectures