DOCUMENTATION

PROJECT 2

NAME OF STUDENT: Jakab-Gyik Sarolta
GROUP: 30423

TABLE OF CONTENTS

1. Objectives

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, waiting, being served, and finally leaving the queues, and (2) computing the average waiting time, average service time and peak hour.

Sub-objectives:

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

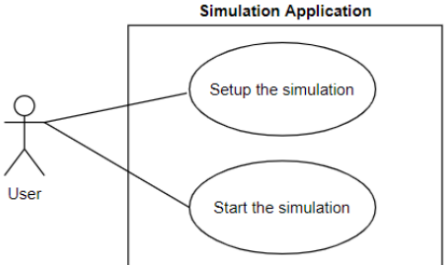
2. Problem analysis, modelling, scenarios, cases of utility

Functional requirements:

The simulation application should allow users to setup the simulation
The simulation application should allow users to start the simulation
The simulation application should display the real-time queues evolution
The simulation application should write the real-time queues evolution and statistics in a log.txt file.
The simulation application should allow users to stop the queue evolution
The simulation application should allow users to restart the simulation

Use Case: *setup simulation*
Primary Actor: user
Main Success Scenario:

1. The user inserts the values for the: number of clients, number of queues, simulation interval, minimum and maximum arrival time, and minimum and maximum service time
2. The user clicks on the START button
3. The application validates the data and displays the start of the simulation



Alternative Sequence:

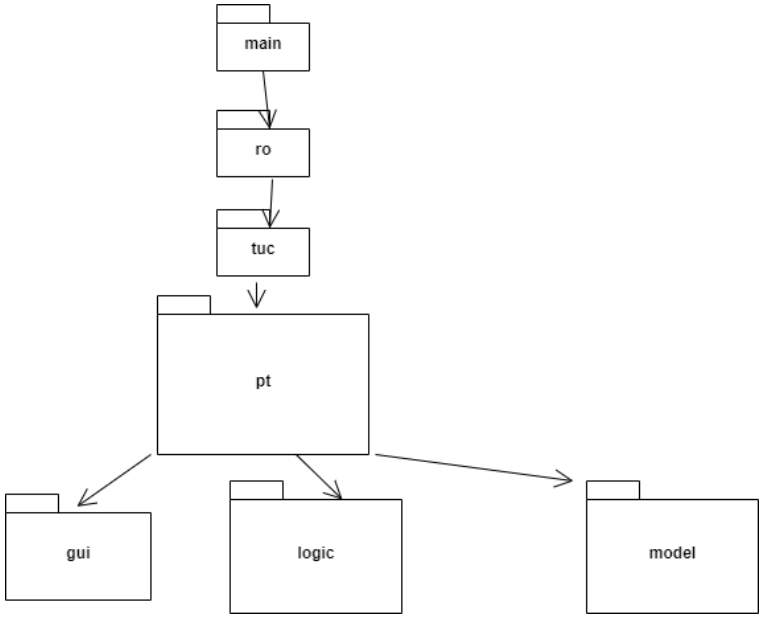
- Invalid values for the setup parameters
- The user inserts invalid values for the application's setup parameters
- The application displays an error message and requests the user to insert valid values
- The scenario returns to step 1

Use Case: *start the simulation*
Primary Actor: user
Main Success Scenario:

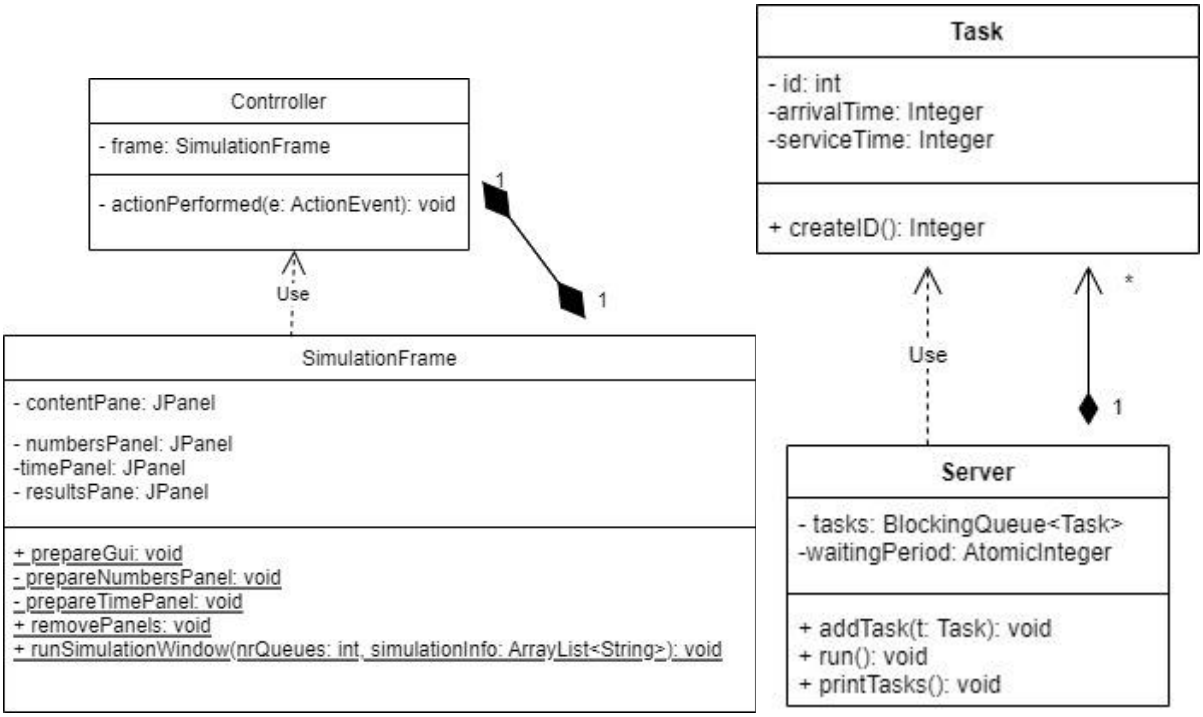
1. The user clicks on the START button
2. The application validates the data and displays the start of the simulation
3. The application updates the interface in real time with the new customers distributed to the queues
4. The simulation time is up and the application stops the refreshes
5. The user clicks the STOP button or the exit window top right corner.

Alternative Sequence:

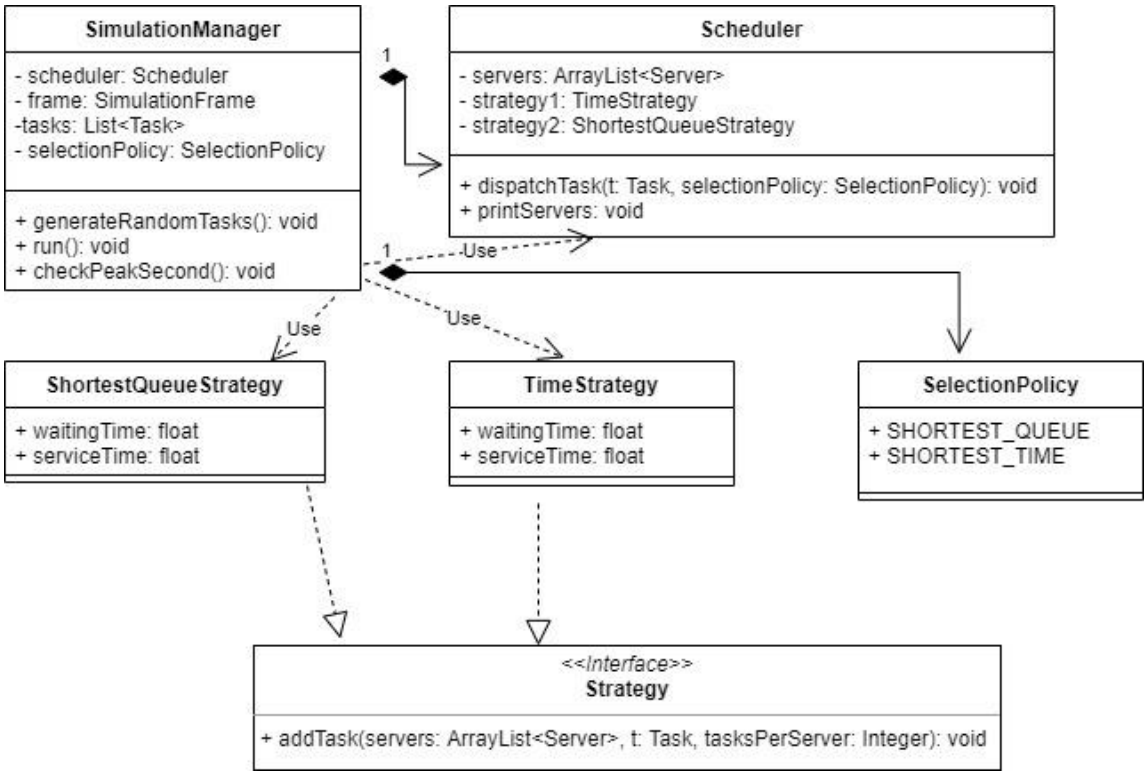
- The simulation is interrupted with the STOP button => return to the setup simulation use case, step 1
- The user does not click on the STOP button => the simulation results keep being displayed



~ Package diagram~



~Class diagrams~



3. Description

The package diagram indicates the Model-View-Controller type architecture used in the realization of the project. These 3 packages are all found inside the ro.tuc.pt package. The “gui” package (aka graphical user interface) contains both the Controller and the View parts of the indicated architecture. The Controller part is found inside the Controller class, while the View inside the View class, as the names suggest.

The Model part with the created data structures, Server and Task classes, is found inside the model package. Also, the logic package belongs to the Model part of the architecture. This is where the SimulationManager class executes the simulation with the parameters indicated by the user of the application, the parameters that the Controller transmits to the Model.
The application uses the concepts of parallel programming. It is heavily dependent on the threads generated for every queue in action. These threads are instances of the Server class that extends the Thread class. Furthermore, to synchronize these threads that run in parallel, I used the AtomicInteger class and BlockingQueue types of queues. These special data structures ensure that the changes a thread makes on any of the fields of a server, all the threads receive the updated values of the specific fields.
Another way to work with threads is implementing the Runnable interface, as for the SimulationManager class. This interface has a run() method that must be overridden by all classes that implement this interface. Therefore, the SimulationManager’s main functionality is found inside this overridden method. It handles all the other threads, the Server threads, and also writes the result of the simulation in a .txt file, after the allocated simulation time is over.
The application implements the ActionListener interface as well. The Controller class detects the pressing of the START or STOP buttons and it calls a method from the SimulationFrame class to be executed, based on the chosen command/button. When START is pressed, the frame will change to the simulation window and the execution of the real-time simulation starts. To the contrary, when STOP is pressed, the simulation is stopped, if it was still in execution, or if it has already been terminated, then it simply changes the interface window to show the setup phase, where the user can determine the parameters of the simulation.

4. Implementation

About every class in details:

Task class

```
public class Task {
    private int id;
    private Integer arrivalTime;
    private Integer serviceTime;

    //to generate the unique IDs
    private static long idCounter = 0;

    public static synchronized Integer createID()
    {
        return (int)idCounter++;
    }

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

The Task class is the abstraction of the client from real life, who waits in a queue in a supermarket. The Task onjects have 3 important fields, the unique id generated by the createID() method, the arirvalTime and the serviceTime, that help the algorithm sort out the clients to the most appropriete queue. This class has getter methods for all its fields' values and a setter method, where the service time could be changed when the certain task is being positioned at the front of the queue. The other fields, such as the id and arrival time, will never change values during the simulation.

Server class

```
public class Server extends Thread{ //better than implements Runnable
    private BlockingQueue<Task> tasks;
    private AtomicInteger waitingPeriod = new AtomicInteger();

    public Server() {
        this.tasks = new LinkedBlockingDeque<>();
        this.waitingPeriod.set(0); //with 0
    }
```

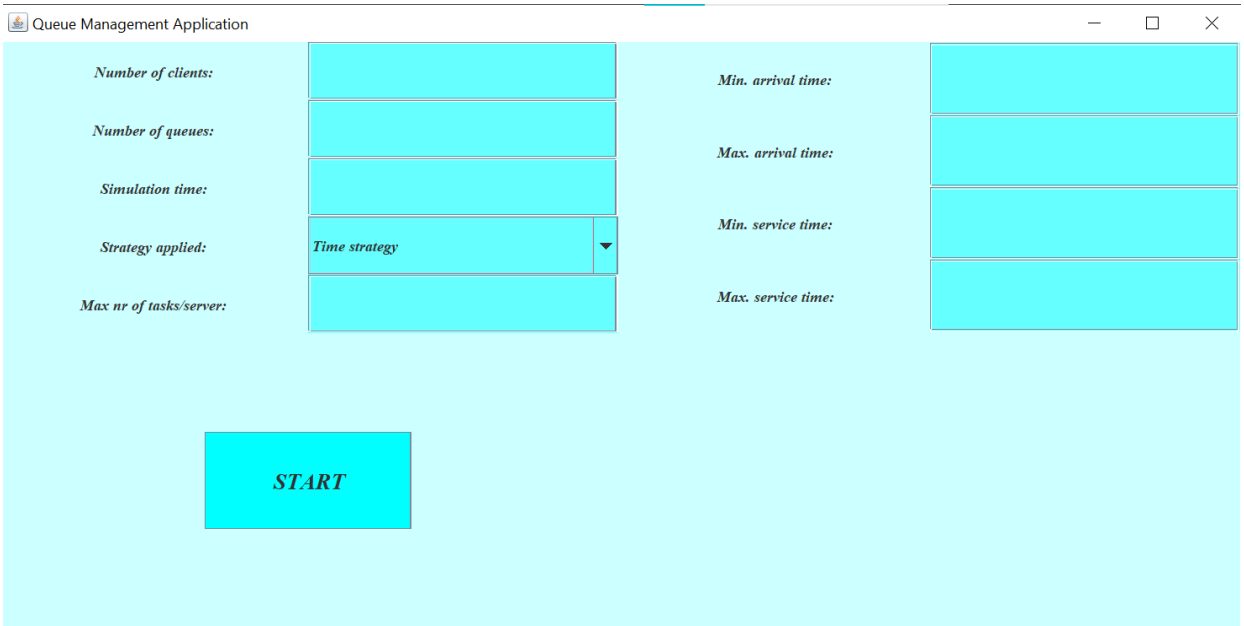
The Server class extends Thread class, meaning that the instances of this class will be threads and executed in parallel. The two fields of this class represent the tasks that are assigned to that queue, so the clients who are waiting in that certain line, and the waiting period which characterizes that queue. Initially, this waiting period is set to 0. As more clients are added to the queue, this number increases with their service time. This field becomes extremely important when TimeStrategy is used, so when the clients are distributed in the queues based on the smallest waiting time.Similarly, the size of the tasks queue is important for the ShortestQueue implementation of the simulation. In that case, the clients are distributed always to the queue with the least number of clients in it.
This class overridden the run() method, which specifies the functionality of the thread when run.
An important method is also the getStringTasks() method.

```java
public String getStringTasks(){
    String taskString = "";
    Iterator<Task> i = tasks.iterator();
    Task currTask = i.next();
    taskString += "(" + currTask.getId() + ", " + currTask.getArrivalTime() + ", " + currTask.getServiceTime() + ") ";
    while (i.hasNext()) {
        currTask = i.next();
        taskString += "(" + currTask.getId() + ", " + currTask.getArrivalTime() + ", " + currTask.getServiceTime() + ") ";
    }
    return taskString;
}
```

This method iterates through the tasks queue and transforms all its contents into a string. With the help of this function, the graphical user interface can be updated: the JLabel to the corresponding queue gets the returned string as the new value to display. And this is how the real time simulation is possible.

SimulationFrame class

This class extends JFrame, being part of the JavaSwing implementation. It has quite a few fields that are graphical components, being part of the contentPane. These components are JLabels, JTextFields or ComboBoxes.
The preparegui() method sets the initial values for the components to be displayed and sets up the window where the user can input all the parameters of the simulation.
The prepareNumbersPanel() method sets the left part of the interface: number of clients, number of queues, simulation time, also the maximum number of tasks per server, which is only relevant in case we choose the ShortestQueue strategy pattern.
The prepareTimePanel() method sets the right part of the interface, namigly the min and max arrival time and min and max service time for all clients.



After pressing the START button, the interface changes to the simualtion window. That is described in the runSimulationWindow() method.


```
ArrayList<JLabel> queueLabels = new ArrayList<>();
tupleLabels = new ArrayList<>();
for (int i = 0; i < nrQueues; i++){
    queueLabels.add(new JLabel( text: "Queue" + (i+1) + ":", JLabel.CENTER));
    queueLabels.get(i).setFont(new Font( name: "Serif",  style: Font.ITALIC | Font.BOLD,  size: 14));
    queueLabels.get(i).setBackground(backgroundColor);
    tupleLabels.add(new JLabel(simulationInfo.get(i + 2), JLabel.CENTER));
    tupleLabels.get(i).setFont(new Font( name: "Serif",  style: Font.ITALIC | Font.BOLD,  size: 20));
    tupleLabels.get(i).setBackground(backgroundColor);
    resultsPane.add(queueLabels.get(i));
    resultsPane.add(tupleLabels.get(i));
}
```

In the above code segment the updated values of the queues are shown, as the JLabels get the new text set in every second. For the whole user interface I used the Grid layout throughout the methods.

Controller class

This class validates all the inputs coming from the user of the application, starts the running threads, decides the strategy pattern and calls the functions accordingly, and also contains the SwingWorker thread, so that the real time simulation is synchronized with the changing values in the queues of the threads.

```
int nrClients = Integer.parseInt(frame.getNumberOfClientsTextField().getText());
if (nrClients <= 0) {
    JOptionPane.showMessageDialog( parentComponent: null,  message: "Incorrect number of clients parameter");
}
int nrQueues = Integer.parseInt(frame.getNumberOfQueuesTextField().getText());
if (nrQueues <= 0) {
    JOptionPane.showMessageDialog( parentComponent: null,  message: "Incorrect number of queues parameter");
}
int simulationTime = Integer.parseInt(frame.getSimulationIntervalTextField().getText());
if (simulationTime <= 0) {
    JOptionPane.showMessageDialog( parentComponent: null,  message: "Incorrect simulation time parameter");
}
int tasksPerServer = Integer.parseInt(frame.getTasksPerServerTextField().getText());
if (tasksPerServer <= 0) {
    JOptionPane.showMessageDialog( parentComponent: null,  message: "Incorrect number of tasks per server");
}
```

As in the example above, the Controller class validates every inputs in its own, and where the case arises, it compares it to the other given inputs: for example, the minimum and maximum arrival time cannot exceed the limits of the simulation interval, therefore it is compared to the simulation time provided. If an input does not correspond to the required formula, then a window pops up, notifying the user about the problem and then lets the user reinitialize the fields.

```
SwingWorker<Void, ArrayList<String>> swingWorker = new SwingWorker<Void, ArrayList<String>>() {
    @Override
    protected Void doInBackground() throws Exception {
        SimulationManager simulation1;
        if(strategies.equals("Shortest queue strategy")) {
            simulation1 = new SimulationManager(SelectionPolicy.SHORTEST_QUEUE, simulationTime, maxServiceTime, minServiceTime, maxArrivalTime, minArrivalTime, nrQueues, nrClients, tasksPerServer);
        }else{
            simulation1 = new SimulationManager(SelectionPolicy.SHORTEST_TIME, simulationTime, maxServiceTime, minServiceTime, maxArrivalTime, minArrivalTime, nrQueues, nrClients, tasksPerServer);
        }

        Thread myThread = new Thread(simulation1);
        myThread.start();

        while(simulation1.isRunning){
            publish(simulation1.queues);
            Thread.sleep( millis: 500);
        }
        return null;
    }

    @Override
    protected void process(List<ArrayList<String>> chunks) {
        ArrayList<String> result = chunks.get(chunks.size()-1);
        frame.runSimulationWindow(nrQueues, result);
        super.process(chunks);
    }
};

swingWorker.execute();
{
if(command.equals("STOP")){
    frame.removePanels();
}
```

The SwingWorker tool helps us realize the updates of the user interface in real time. The 500 millisecond sleep provides just enough time for the user to see the correct output, but when the new values are calculated, then it gets updated again.

SimulationManager class

This class contains the core logic of the application. It generates the tasks, transforms everything into strings to appear on the graphical interface, and last but not least, the run method, calling other functions, distributes the tasks, decrements the first task's service time each second, computes the statistical values and writes them into a log.txt file.

```
public void generateRandomTasks(){
    System.out.println("Generated random tasks:");
    for (int i = 0; i < numberOfClients; i++){
        int arrivalTime = (int) Math.floor(Math.random()*(maxArrivalTime-minArrivalTime+1)+minArrivalTime);
        int processingTime = (int) Math.floor(Math.random()*(maxProcessingTime-minProcessingTime+1)+minProcessingTime);
        Task newTask = new Task(arrivalTime, processingTime);
        System.out.println("(" + newTask.getId() + ", " + newTask.getArrivalTime() + ", " + newTask.getServiceTime() + ")");
        tasks.add(newTask);
    }
}
```

The generate function uses the random number generator provided in the Math library, and generates an arrival time and a service time for every task in the given range (min – max arrival time and min – max service time). Afterwards, it also adds all these tasks to the waiting list.

```
public void checkPeakSecond(){
    ArrayList<Integer> seconds = new ArrayList<>();
    for (int i = 0; i < timeLimit; i++){
        seconds.add(0);
    }
    for (int i = 0; i < tasks.size(); i++){
        seconds.set(tasks.get(i).getArrivalTime(), seconds.get(tasks.get(i).getArrivalTime()) + 1);
    }
    for (int i = 0; i < timeLimit; i++){
        if (seconds.get(i) > peakSecond){
            peakSecond = i;
        }
    }
}
```

The peak second function is called right after the generate function. Its role is to determine the busiest second during the simulation. Because all the arrival times are generated at random, the peak second could be anywhere in between the minimum and the maximum arrival time moments. Therefore, with an array that has the exact same length as the simulation time we check how many clients arrive in each second. And after adding + 1 for every client, at the end we iterate through the array once more to see which index (second in our case) has the highest number stored in the array at the corresponding index.

```
for (int i = 0; i < tasks.size(); i++) {
    if (tasks.get(i).getArrivalTime() <= currentTime){
        //System.out.println("Start task: " + tasks.get(i).getId() + ", with arrival time: " + tasks.get(i).getArrivalTime());
        if(scheduler.dispatchTask(tasks.get(i), selectionPolicy)){
            tasks.remove(i);
            i--;
        }
    }
}
```

In this part of the run() method the program iterates through the tasks array and checks the arrival times. If any arrival time corresponds to the current time, meaning that the task has just arrived and it can be placed into a queue, then the dispatchTask method does the placing. After that the task is removed from the waiting list.

Scheduler class

This class has a single most important method, the dispatchTask method, which decides, based on the strategy pattern selected, which add method to call. In case the add function returns false, which means that the task could not be added to the queue because the number of tasks per server is already achieved, then this function returns a false, signaling to the simulation manager that the task was not added, therefore, it should not be removed from the waiting list.

```java
public boolean dispatchTask(Task t, SelectionPolicy selectionPolicy){ //call the strategy add task method
    if(selectionPolicy == SelectionPolicy.SHORTEST_TIME){
        this.strategy1.addTask(this.servers, t, maxTasksPerServer);
    }
    else{
        if(!this.strategy2.addTask(this.servers, t, maxTasksPerServer)){
            System.out.println("All servers are full at the moment");
            return false;
        }
    }
    return true;
}
```

Another method that I have implemented in the Scheduler class was the changeStrategy method. However, for the final implementation of the application I decided not to use it. There is no case when the user would want a strategy change in the middle of the simulation, so I have just left the function as a comment.

Strategy interface

```java
public interface Strategy {
    public boolean addTask(ArrayList<Server> servers, Task t, Integer tasksPerServer);
}
```

TimeStrategy class

This class implements the Strategy interface. It always compares the waiting periods for every queue, and places the task at hand in the queue with the smallest waiting time. It also calculates the total waiting time for every task that enters any queue. The public float number will be used to calculate the average waiting time at the end. Same for the service time, adding all of them and then dividing them with the number of tasks gives us an average value.

```java
public class TimeStrategy implements Strategy{
    public static float waitingTime = 0;
    public static float serviceTime = 0;

    @Override
    public boolean addTask(ArrayList<Server> servers, Task t, Integer tasksPerServer) {
        AtomicInteger waitingPeriod = new AtomicInteger( initialValue: 500);
        int ind = 0;
        for (int i = 0; i < servers.size(); i++){
            if (servers.get(i).getWaitingPeriod().intValue() < waitingPeriod.intValue()){
                waitingPeriod = servers.get(i).getWaitingPeriod();
                ind = i;
            }
        }
        waitingTime += waitingPeriod.intValue();
        System.out.println("Waiting: " + waitingTime);
        serviceTime += t.getServiceTime();
        servers.get(ind).addTask(t);
        return true;
    }
}
```

ShortestQueueStrategy class

This class also implements the Strategy interface. This strategy could be considered the easier one out of the two. It checks if the queue has reached the maximum number of tasks in it, and if not, then the new task is placed in the queue with the least tasks in it. If all queues are full, then the return value is false, otherwise true. Inside the function we also calculate the sum of the waiting periods and the service times.

```
public class ShortestQueueStrategy implements Strategy{
    public static float waitingTime = 0;
    public static float serviceTime = 0;

    @Override
    public boolean addTask(ArrayList<Server> servers, Task t, Integer tasksPerServer) {
        int minSize = tasksPerServer + 1, ind = -1;
        for (int i = 0; i < servers.size(); i++){
            if (servers.get(i).getTasks().size() < tasksPerServer && servers.get(i).getTasks().size() < minSize){
                minSize = servers.get(i).getTasks().size();
                ind = i;
            }
        }
        if (ind == -1){ //cannot add more tasks to servers - all full
            return false;
        }
        Iterator<Task> i = servers.get(ind).getTasks().iterator();
        float currWaitingTime = 0;
        while (i.hasNext()) {
            currWaitingTime += i.next().getServiceTime();
        }
        waitingTime += currWaitingTime;
        serviceTime += t.getServiceTime();
        servers.get(ind).addTask(t);
        return true;
    }
}
```

5. Results

To test the application, we were given 3 sets of inputs, and their simulation results were the following:

For Simulation 1:
Number of clients: 4
Number of queues: 2
Simulation time: 60 secs
Arrival time interval: [2, 30]
Service time interval: [2, 4]

A part of the result:

```
Time 6
Waiting clients (0, 19, 2)  (1, 11, 2)  (2, 24, 4)  (4, 16, 4)  (5, 15, 4)  (6, 8, 3)  (7, 20, 3)  (8, 27, 4)  (9, 11, 2)
Queue 1: (3, 6, 3)
Queue 2: closed
Time 7
Waiting clients (0, 19, 2)  (1, 11, 2)  (2, 24, 4)  (4, 16, 4)  (5, 15, 4)  (6, 8, 3)  (7, 20, 3)  (8, 27, 4)  (9, 11, 2)
Queue 1: (3, 6, 2)
Queue 2: closed
Time 8
Waiting clients (0, 19, 2)  (1, 11, 2)  (2, 24, 4)  (4, 16, 4)  (5, 15, 4)  (7, 20, 3)  (8, 27, 4)  (9, 11, 2)
Queue 1: (3, 6, 1)
Queue 2: (6, 8, 3)
Time 9
Waiting clients (0, 19, 2)  (1, 11, 2)  (2, 24, 4)  (4, 16, 4)  (5, 15, 4)  (7, 20, 3)  (8, 27, 4)  (9, 11, 2)
Queue 1: closed
Queue 2: (6, 8, 2)
Time 10
Waiting clients (0, 19, 2)  (1, 11, 2)  (2, 24, 4)  (4, 16, 4)  (5, 15, 4)  (7, 20, 3)  (8, 27, 4)  (9, 11, 2)
Queue 1: closed
Queue 2: (6, 8, 1)
Time 11
Waiting clients (0, 19, 2)  (2, 24, 4)  (4, 16, 4)  (5, 15, 4)  (7, 20, 3)  (8, 27, 4)
Queue 1: (1, 11, 2)
Queue 2: (9, 11, 2)
Time 12
Waiting clients (0, 19, 2)  (2, 24, 4)  (4, 16, 4)  (5, 15, 4)  (7, 20, 3)  (8, 27, 4)
Queue 1: (1, 11, 1)
Queue 2: (9, 11, 1)
Time 13
Waiting clients (0, 19, 2)  (2, 24, 4)  (4, 16, 4)  (5, 15, 4)  (7, 20, 3)  (8, 27, 4)
Queue 1: closed
Queue 2: closed
```

For Simulation 2:
Number of clients: 50
Number of queues: 5
Simulation time: 60 secs
Arrival time interval: [2, 40]
Service time interval: [1, 7]

```
The avg waiting time is: 4.12
The avg service time is: 4.08
The peak second is: 12
```

A part of the result:

```
Time 7
Waiting clients (0, 34, 7)  (1, 22, 3)  (2, 14, 1)  (3, 26, 2)  (4, 21, 2)  (5, 12, 5)  (6, 11, 6)  (7, 18, 4)  (8, 35, 3)  (9, 14, 5)  (10, 40,
Queue 1: (11, 3, 2)
Queue 2: (21, 7, 2)
Queue 3: closed
Queue 4: closed
Queue 5: closed
Time 8
Waiting clients (0, 34, 7)  (1, 22, 3)  (2, 14, 1)  (3, 26, 2)  (4, 21, 2)  (5, 12, 5)  (6, 11, 6)  (7, 18, 4)  (8, 35, 3)  (9, 14, 5)  (10, 40,
Queue 1: (11, 3, 1)
Queue 2: (21, 7, 1)
Queue 3: (22, 8, 1)
Queue 4: (28, 8, 3)
Queue 5: (47, 8, 5)
Time 9
Waiting clients (0, 34, 7)  (1, 22, 3)  (2, 14, 1)  (3, 26, 2)  (4, 21, 2)  (5, 12, 5)  (6, 11, 6)  (7, 18, 4)  (8, 35, 3)  (9, 14, 5)  (10, 40,
Queue 1: closed
Queue 2: closed
Queue 3: closed
Queue 4: (28, 8, 2)
Queue 5: (47, 8, 4)
Time 10
Waiting clients (0, 34, 7)  (1, 22, 3)  (2, 14, 1)  (3, 26, 2)  (4, 21, 2)  (5, 12, 5)  (6, 11, 6)  (7, 18, 4)  (8, 35, 3)  (9, 14, 5)  (10, 40,
Queue 1: (44, 10, 4)
Queue 2: closed
Queue 3: closed
Queue 4: (28, 8, 1)
Queue 5: (47, 8, 3)
Time 11
Waiting clients (0, 34, 7)  (1, 22, 3)  (2, 14, 1)  (3, 26, 2)  (4, 21, 2)  (5, 12, 5)  (7, 18, 4)  (8, 35, 3)  (9, 14, 5)  (10, 40, 3)  (12, 18
Queue 1: (44, 10, 3)
Queue 2: (6, 11, 6)
Queue 3: (14, 11, 6)
Queue 4: (15, 11, 2)
Queue 5: (47, 8, 2)
```

For Simulation 3:
Number of clients: 1000
Number of queues: 20
Simulation time: 200 secs
Arrival time interval: [10, 100]
Service time interval: [3, 9]
A part of the result:

```
The avg waiting time is: 102.602
The avg service time is: 6.043
The peak second is: 23
```

```
Time 51
Waiting clients (1, 66, 3) (2, 98, 3) (3, 90, 8) (4, 97, 7) (6, 70, 9) (7, 64, 9) (8, 63, 5) (11, 54, 8) (12, 55, 9) (13, 72, 7) (14, 96, 6) (17, 52
, 97, 4) (148, 63, 3) (149, 89, 5) (150, 96, 3) (151, 98, 5) (152, 78, 5) (153, 91, 5) (155, 79, 3) (156, 74, 5) (157, 65, 5) (162, 52, 3) (164, 53,
, 75, 9) (270, 98, 4) (272, 81, 4) (273, 92, 9) (276, 62, 3) (277, 53, 4) (279, 79, 8) (280, 56, 8) (284, 86, 6) (285, 60, 6) (288, 74, 9) (289, 89,
90, 8) (416, 88, 6) (418, 68, 9) (419, 71, 3) (421, 91, 3) (422, 76, 8) (423, 100, 3) (425, 84, 3) (427, 64, 4) (428, 100, 3) (429, 89, 4) (430, 87,
39, 67, 5) (542, 89, 5) (543, 71, 8) (545, 87, 4) (546, 71, 9) (547, 63, 8) (548, 60, 6) (549, 67, 5) (550, 59, 6) (555, 87, 9) (563, 58, 4) (564, 7
9, 87, 4) (680, 84, 6) (681, 85, 4) (682, 80, 5) (683, 61, 9) (684, 53, 3) (687, 93, 9) (688, 66, 5) (690, 89, 8) (691, 97, 9) (693, 81, 3) (694, 88
98, 81, 7) (799, 85, 8) (800, 82, 7) (801, 70, 4) (803, 82, 8) (804, 99, 9) (807, 58, 7) (808, 66, 6) (815, 55, 9) (819, 76, 3) (820, 65, 7) (821, 9
6, 68, 8) (927, 98, 5) (928, 67, 7) (929, 86, 3) (930, 96, 5) (931, 56, 3) (933, 56, 8) (934, 52, 7) (936, 91, 3) (937, 53, 5) (938, 97, 3) (939, 58
Queue 1: (293, 23, 4) (109, 25, 5) (551, 26, 5) (733, 27, 5) (660, 29, 6) (208, 31, 9) (635, 33, 5) (740, 35, 4) (702, 36, 4) (957, 37, 5) (499, 40, 9) (139, 4
Queue 2: (748, 22, 4) (581, 25, 9) (420, 27, 5) (264, 29, 9) (942, 31, 6) (259, 33, 4) (403, 35, 8) (477, 37, 6) (970, 39, 4) (283, 41, 8) (862, 43, 8) (278, 4
Queue 3: (204, 23, 1) (614, 23, 8) (741, 26, 9) (472, 29, 8) (622, 31, 9) (992, 33, 3) (686, 35, 7) (692, 37, 8) (891, 40, 4) (725, 41, 9) (968, 44, 7) (91, 47
Queue 4: (248, 22, 2) (271, 24, 7) (766, 26, 6) (387, 28, 5) (715, 29, 6) (756, 31, 7) (297, 33, 4) (406, 35, 6) (940, 36, 5) (339, 38, 4) (629, 40, 4) (467, 4
Queue 5: (389, 21, 1) (811, 23, 7) (398, 26, 8) (414, 28, 5) (175, 30, 4) (305, 31, 9) (734, 33, 6) (174, 36, 9) (92, 39, 6) (325, 41, 7) (321, 43, 5) (170, 45
Queue 6: (507, 23, 8) (531, 26, 6) (773, 27, 9) (880, 30, 8) (781, 32, 4) (295, 34, 8) (107, 37, 6) (165, 39, 5) (36, 41, 8) (365, 43, 5) (184, 45, 8) (438, 47
Queue 7: (587, 23, 3) (813, 24, 6) (842, 26, 9) (559, 29, 4) (369, 30, 5) (946, 31, 9) (117, 35, 3) (812, 35, 5) (314, 37, 4) (455, 38, 4) (633, 40, 6) (244, 4
Queue 8: (354, 23, 2) (553, 24, 3) (16, 26, 6) (132, 27, 6) (577, 28, 6) (909, 30, 7) (640, 32, 9) (964, 35, 3) (601, 36, 9) (995, 39, 5) (506, 41, 3) (385, 42
Queue 9: (461, 23, 4) (621, 25, 7) (209, 27, 9) (782, 29, 8) (998, 31, 5) (340, 33, 8) (409, 36, 4) (818, 37, 4) (263, 39, 8) (791, 41, 5) (517, 43, 3) (540, 4
Queue 10: (661, 22, 4) (714, 25, 6) (69, 27, 7) (674, 28, 8) (424, 31, 7) (159, 33, 6) (707, 35, 9) (27, 38, 4) (5, 40, 9) (767, 42, 8) (426, 45, 6) (230, 47,
Queue 11: (728, 22, 4) (980, 25, 8) (379, 27, 8) (806, 29, 5) (608, 31, 9) (747, 33, 6) (268, 36, 3) (218, 37, 4) (265, 38, 3) (961, 39, 3) (945, 40, 6) (400,
Queue 12: (877, 22, 2) (567, 24, 3) (124, 26, 9) (850, 27, 6) (875, 29, 8) (15, 32, 7) (302, 34, 5) (356, 36, 9) (396, 39, 7) (560, 41, 8) (211, 44, 3) (203, 4
Queue 13: (370, 23, 1) (904, 23, 8) (34, 27, 7) (486, 28, 4) (19, 30, 8) (75, 32, 3) (898, 32, 6) (600, 35, 3) (386, 36, 6) (885, 37, 9) (395, 41, 7) (836, 43,
Queue 14: (874, 21, 1) (975, 23, 4) (154, 26, 8) (469, 27, 4) (923, 28, 5) (382, 30, 4) (878, 31, 6) (192, 33, 5) (613, 35, 4) (492, 36, 8) (489, 39, 5) (41, 4
Queue 15: (897, 22, 3) (816, 24, 6) (40, 27, 8) (94, 29, 5) (530, 30, 6) (468, 32, 7) (311, 34, 9) (315, 37, 3) (317, 38, 8) (412, 41, 8) (57, 44, 6) (976, 45,
Queue 16: (290, 23, 2) (712, 24, 4) (274, 26, 7) (541, 27, 9) (570, 30, 8) (729, 32, 6) (125, 35, 5) (573, 36, 7) (663, 38, 9) (805, 41, 3) (861, 42, 8) (537,
Queue 17: (384, 23, 3) (837, 24, 6) (64, 27, 5) (127, 28, 6) (20, 30, 8) (583, 32, 8) (287, 35, 7) (225, 37, 3) (935, 37, 3) (835, 39, 9) (324, 42, 8) (37, 45,
Queue 18: (612, 23, 7) (358, 26, 7) (140, 28, 8) (652, 30, 7) (616, 32, 6) (453, 34, 4) (973, 35, 7) (944, 37, 5) (67, 40, 3) (158, 41, 4) (346, 42, 6) (441, 4
Queue 19: (76, 23, 5) (262, 26, 9) (298, 28, 9) (121, 31, 5) (586, 32, 5) (488, 33, 6) (89, 36, 9) (775, 38, 8) (638, 41, 4) (947, 42, 3) (160, 44, 3) (134, 45
Queue 20: (739, 22, 1) (35, 24, 9) (78, 27, 4) (378, 28, 7) (281, 30, 5) (925, 31, 9) (787, 34, 4) (161, 36, 7) (948, 37, 3) (919, 39, 8) (171, 42, 3) (996, 42
```

6. Conclusion

All the code fragments presented above, and the test cases ensure that the project accomplishes the predefined objectives. The application can create and synchronize threads, run them, and manipulate data without inconsistency. The strategy patterns applied highlight the different ways this problem could be approached and solved. The shortest time strategy is the most applied solution in case of similar problems; however, shortest queue could lead to other results given the same set of inputs. The user interface lets us follow the steps that the simulation takes after creating the threads. It is visually pleasing and entertaining to follow.

As always, there are ways for further improvement, like applying different strategy patterns or working with bigger, more complex data sets. However, in this phase of the development, this application has helped me in understanding threads, how they work, how they can be implemented in two different ways in Java, and how they can be synchronized with the user interface.

7. Bibliography

The resources for this project were the pdf-s provided by the professor, the presentation of the project and the pdf guide.
I have also tackled several problems with the help of different websites explaining different Java concepts.