DOCUMENTATATION

ASSIGNMENT 2: *QUEUE MANAGEMENT APPLICATION*

STUDENT NAME: PELLE ANDREI

TABLE OF CONTENTS

1. Objectives

Design and implement a queues management application which assigns clients to queues such that the waiting time is minimized. Queues are commonly used to model real world domains. 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 are interested in minimizing the time amount their "clients" are waiting in queues before they are served.

Breaking down the problem, I considered the following tasks:

- **Analyze and design an elegant solution** : One has to decide before starting work who is going to be using this application, what the use cases are and how all the components work together at a high level. (Chapter 2 and 3)
- **Building a user-friendly graphical user interface** that allows the user to select the simulation time, number of clients, number of queues, min and max arrival time and min and max processing time(Chapter 4)
- **Data modelling**: A queue manager is a rather complex data structure/unit; it must be modelled in a way that allows the operations to be performed easily and logically. (Chapter 3)
- **Input validation**: we must make sure the input is correct and display a message to the user if the input is somehow invalid. (Chapter 4)
- **Queue management and synchronization**: Where more threads access information, this place must be synchronized in order to avoid undesired results. (Chapter 4)
- **Test the application**: To ensure that the results are the ones we expect, the application needs to be tested. The results of the tests of the assignment description have been included. (Chapter 5)

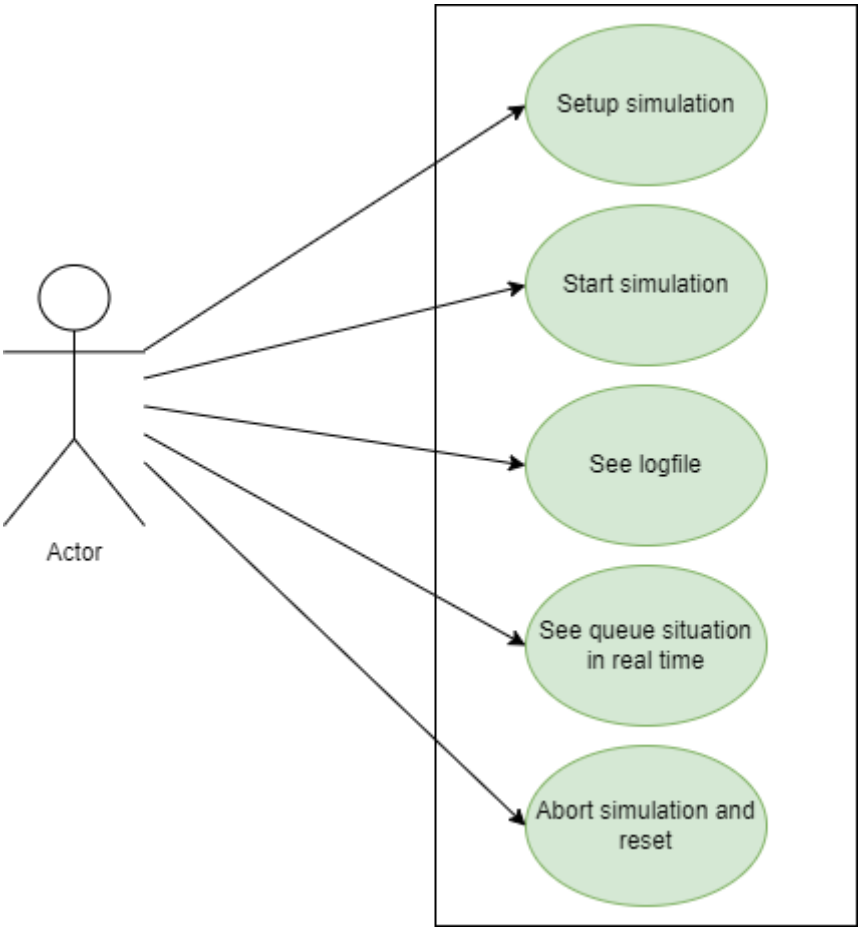
2. Problem analysis, modelling, use cases

The following **functional requirements** can be defined:

- The queue simulation manager should generate a random set of tasks each time it is created.
- The queue simulation manager should allow users to setup the simulation.
- The queue simulation manager should allow users to start the simulation.
- The queue simulation manager should allow users to abort the current simulation.
- The queue simulation manager should allow users to run multiple simulations(reset button).

The following **non-functional requirements** can be defined:

- The queue simulation manager should be intuitive and easy to use by the user
- The queue simulation manager should be easy to maintain
- The s queue simulation manager should be stable



- The queue simulation manager should be modifiable and allow for future development.

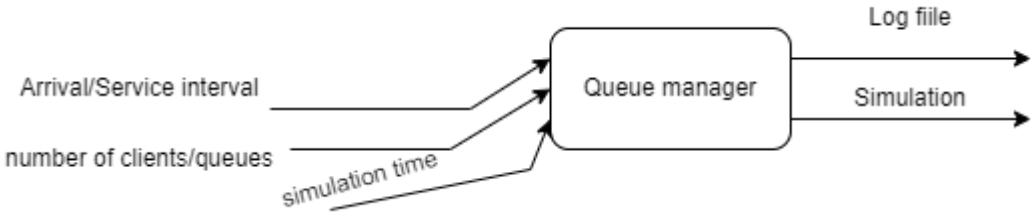
Use cases

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 validate input data button
3. The application validates the data and starts 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, indicating the source
- The scenario returns to step 1

Use case 2: abort simulation
Primary actor: user
Scenario:
1. The user successfully starts the simulation
2. The user aborts the simulation prematurely
3.The simulation halts and the user can see the output thus far

Use case 3: reset simulation
Primary actor: user
Scenario: The user resets the simulation and can start a new one with new parameters. The same log file will be overwritten, and the previous data is lost. The user can reset while the simulation is running although this is ill-advised since it causes a forceful termination of the simulation.

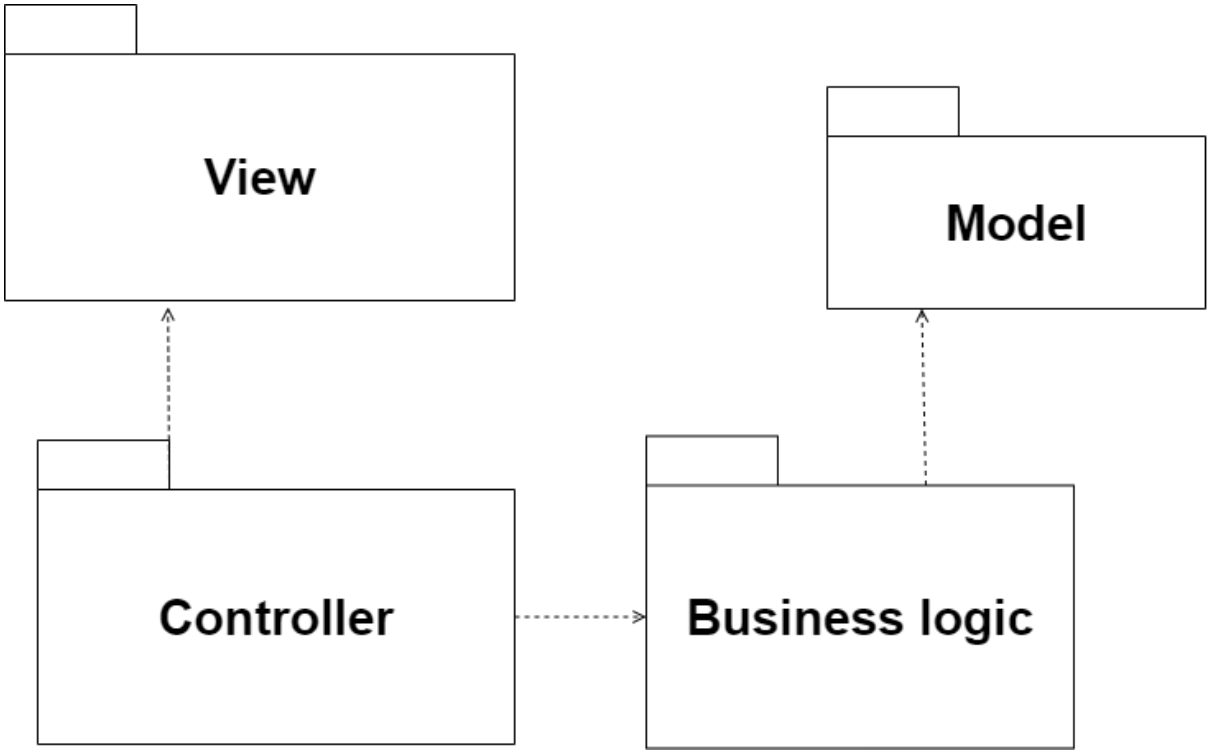
3. Design



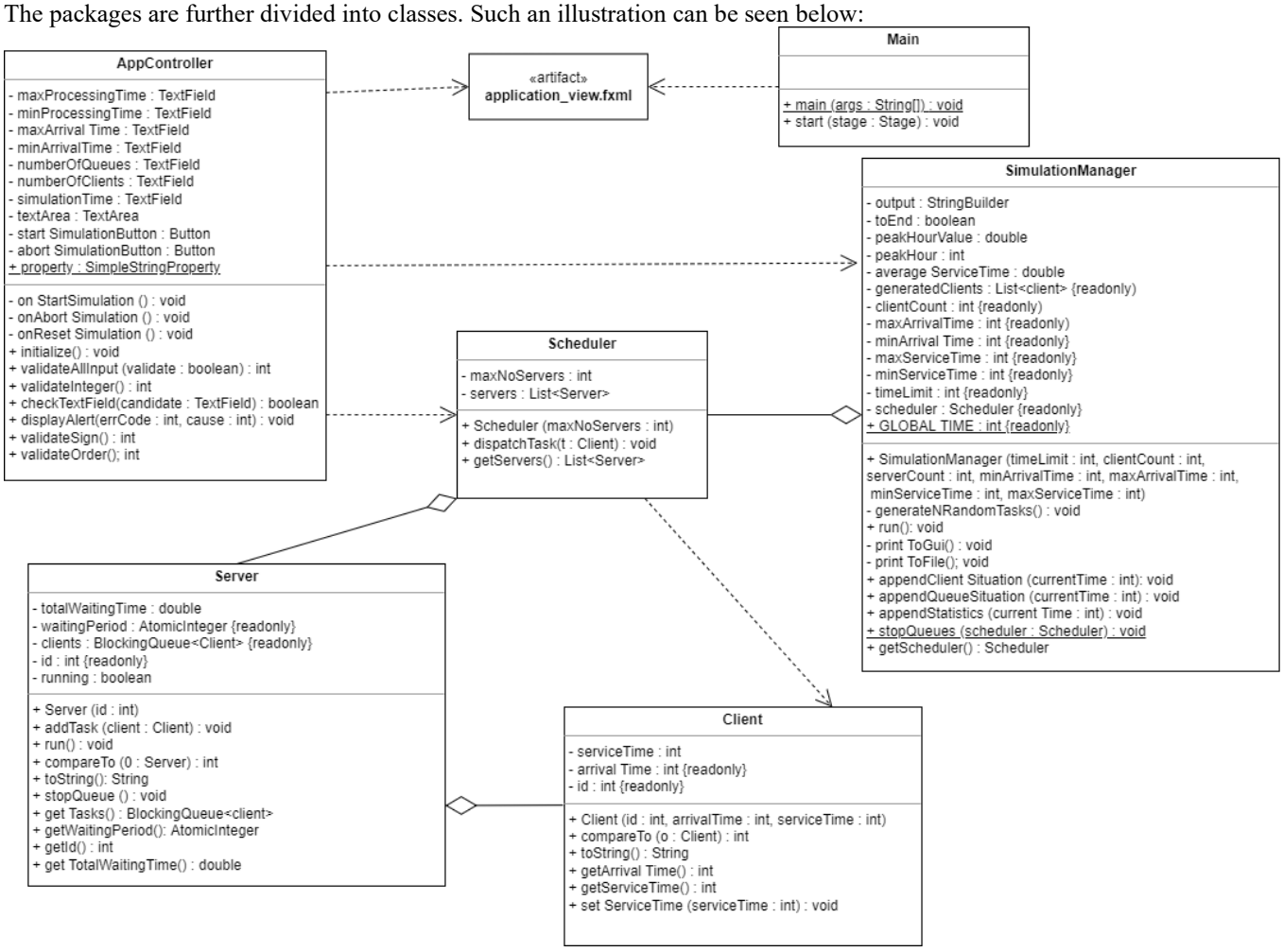
The queue simulation manager has the following inputs and outputs: number of clients, number of queues, simulation interval, minimum and maximum arrival time, and minimum and maximum service time; and at the output we have a log file and the real time simulation.

Next, the design is broken down into packages:

- Controller: this package contains the classes that control the flow of the application. Because UI applications are event-driven, and JavaFX is no exception, this package decides what explicitly happens and waits for user input.
- Business_logic: Contains the classes scheduler and simulation manager which control the flow of the actual simulation, implements a strategy, and dispatches the tasks.
- Model: contains information about the way the data is modelled internally, i.e., the way that the queues and client classes are implemented.
- View (implicit): this package includes the definition for the UI elements written in Oracle's FXML language. It specifies the controller it is linked to for the action handlers.



The packages are further divided into classes. Such an illustration can be seen below:



The data structure I considered for representing the model is a server. This server has waiting period as atomic integer (atomic operations for add, sub, etc.) and an ArrayBlockingQueue which allows for minimal synchronization since it is already synchronized as part of the Collections framework. It also has a running field which helps when it needs to be stopped. The client class has the randomly generated id, service time and arrival time.

The Server and client classes implement the comparable interface. Clients are sorted on their arrival time so it's easier to check if they should be queued or not. Also, the server class implements the comparable interface so that the scheduler can easily see which queue has the lowest waiting time (lowest value of the waiting period atomic integer). Server and simulation manager both implement the runnable interface and the run() method as they are run on an individual thread each.

The AppController class links the FXML file and the event handlers.

As far as special algorithms or techniques are concerned, the strategy for dispatching tasks was shortest queue time, or rather the queue which at that point has the shortest waiting time to get served.

4. Implementation

The **Client** class is the representation of a client/task that needs to be processed: it has an id, arrivalTime and serviceTime, and a single constructor with parameters. It also implements the comparable interface on another client object, the order relation being set on arrival time. The class also overrides the .toString() method in order to provide easy printing to command line and GUI. Relevant code:

```
public class Client implements Comparable<Client>{
    private final int id;
    private final int arrivalTime;
    private int serviceTime;

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

    @Override
    public int compareTo(Client o) {
        if(this.arrivalTime < o.arrivalTime) return -1;
        if(this.arrivalTime > o.arrivalTime) return 1;
        return 0;
    }

    @Override
    public String toString() {
        return "(" + id +","+ arrivalTime +","+ serviceTime +
                ')';
    }
```

The **Server** class is the representation of the actual queue: an ArrayBlockingQueue of clients, id, waiting period as an atomic integer (less synchronization necessary), a total waiting time that will contain the average waiting time for that queue at the end of the simulation and a Boolean field named “running” which is set to false when we want the safe stop of the queue. The constructor initializes the queue to a linkedblockingqueue, I chose this type of queue because it has dynamic size, useful for the 1000 client simulation. The addTask() method is called externally by the scheduler and synchronizes adding clients to the queue. First bit of relevant code is the run() method, which is the core functioning of the queue. In this method, the queue checks if there any clients waiting in the queue, if so, it will check the first one (.peek()) and decrease its service time by 1 every second (by default, GLOBAL_TIME is set to 1000 ms). When the service time for the client reaches 0, it is removed from the queue and the next one is processed.

```
public void run()
{
    while (running)
    {
        if(!clients.isEmpty())
        {
            try {
                Client t  = clients.peek();
                while (t.getServiceTime() > 0) {
                    Thread.sleep(SimulationManager.GLOBAL_TIME);
                    synchronized (waitingPeriod)
                    {
                        totalWaitingTime += waitingPeriod.get();
                        t.setServiceTime(t.getServiceTime()-1);
                        waitingPeriod.getAndAdd( delta: -1);
                    }
                }
                clients.take();
            } catch (InterruptedException exception) {
                break;
            }
        }
    }
}
```

The next code slice shows the implementation of the compareTo() method, toString() printing and the stopQueue() method. The stopQueue() method is called in the simulation manager at the end.

The class **Scheduler** starts the queues and creates a strategy in order to schedule the clients to the queues such that they spend the least amount of time waiting before being serviced. The server list is synchronized (vector class is synchronized in java) and the maxNoServers contains the number of servers to be deployed on threads. The dispatchTask() method is also simple: a call to Collections.sort() sorts the queues on increasing waiting time and then selects the one with the smallest waiting time and calls the addTask() method of that respective queue. The client / task will be deleted from the list in the simulation manager.

```
public class Scheduler {
    private List<Server> servers;
    private int maxNoServers;

    public Scheduler(int maxNoServers) {
        this.maxNoServers = maxNoServers;
        servers = new Vector<>();
        for(int i=1; i<=maxNoServers; i++)
        {
            Server s = new Server(i);
            servers.add(s);
            Thread t = new Thread(s);
            t.start();
        }
    }

    public void  dispatchTask(Client t)
    {
        Collections.sort(servers);
        Server min = servers.get(0);
        min.addTask(t);
    }
```

The **SimulationManager** is the most important and the most complex class in the entire application. This simulation manager advances the time and keeps track of everything that happens in the queues in order to display this information to the user. The fields are pretty self-explanatory: simulation parameters, a scheduler, a list of generated clients, and a flag that signals a premature exit to the simulation. In the constructor, the tasks are generated and placed in a synchronized data structure (vector), also the queues are implicitly started using the call to “new Scheduler”. The output StringBuilder will hold the output to be appended to the GUI text area, more on that later. The generate N random tasks method will use a rand object from the Random class in order to generate the task parameters within the bounds specified by the simulation parameters. This function does a 2-in-1 as it also calculates the average service time for all the clients as a whole. Because the client class implements the comparable interface, a call to sort() will order the clients in increasing arrival order.

```java
private void generateNRandomTasks()
{
    averageServiceTime = 0;
    for(int i=1; i<=clientCount; i++)
    {
        Random rand = new Random();
        Client t = new Client(i,rand.nextInt(minArrivalTime, bound: maxArrivalTime + 1),
                rand.nextInt(minServiceTime,  bound: maxServiceTime + 1));
        generatedClients.add(t);
        averageServiceTime += t.getServiceTime();
    }
    averageServiceTime = averageServiceTime/clientCount;
    Collections.sort(generatedClients);
}
```

I will describe the utility functions first before moving on to the run() method. Because the GUI can only be updated on the JavaFX main thread and the simulation manager runs on a different thread, this is not possible by default. What needs to be done is to use the Platform.runLater() method in order to queue this update to the GUI in order to still keep it responsive. The JavaFX main thread will update the GUI as it sees fit. The printToFile() method will generate the log file for the simulation.

```java
private void printToGui() { Platform.runLater(() -> AppController.property.setValue(output.toString())); }
private void printToFile()
{
    try {
        FileWriter myWriter = new FileWriter( fileName: "log.txt");
        myWriter.write(output.toString());
        myWriter.close();
    } catch (IOException e) {
        System.out.println("An error occurred.");
        e.printStackTrace();
    }
}
```

The appendClientSituation(), appendQueueSitaution() and appendStatistics() are very similar in behavior. These methods append to the output StringBuilder all the necessary information about the simulation and use the aforementioned toString() functions of queue and task. When there are no more tasks, the output will say that there are no more tasks to dispatch. The statistics are represented on double, floating point arithmetic being represented with 4 decimal points.

The final utility method is called stopQueues() and it does what it says, it will call the stopQueue() method, the Boolean value being set to false, and the thread will exit once it is finished processing the final task/client.

```java
public static void stopQueues(Scheduler scheduler)
{
    for(Server s: scheduler.getServers())
    {
        s.stopQueue();
    }
}
```

The first part of the run() method can be seen below. Current time is initialized to 0 and incremented by 1 in every iteration. First, the simulation manager searches for tasks with arrival time equal to current time then adds

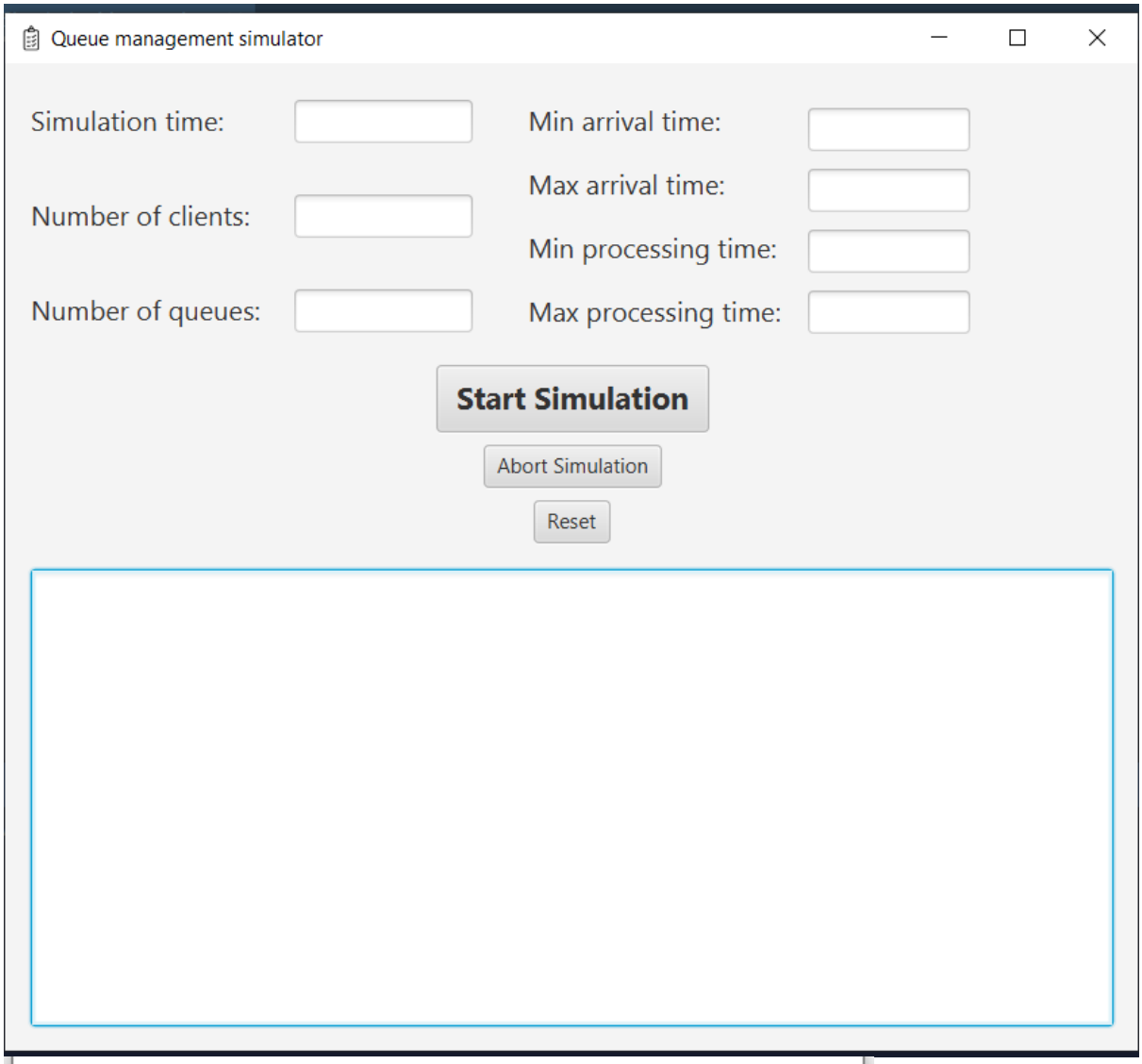
them to a list that will be deleted afterwards. Afterwards, the functions that generate the queue/client situation are called.

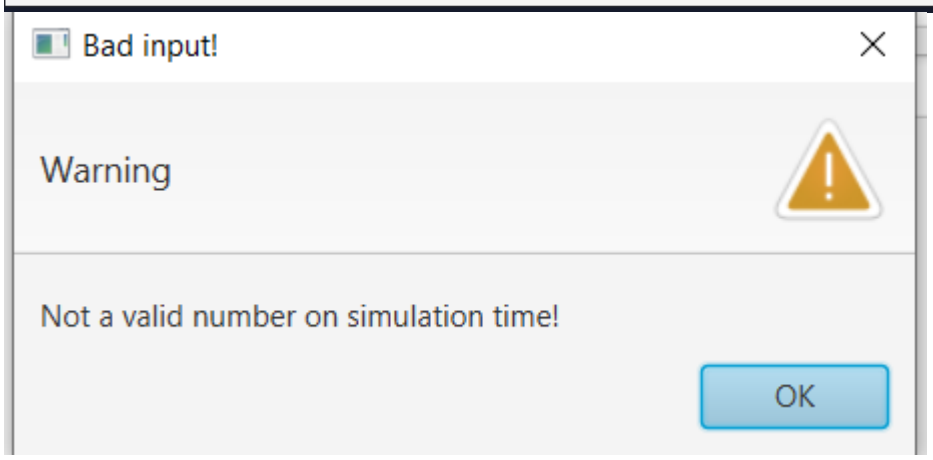
```
@Override
public void run() {
    int currentTime = 0;
    while (currentTime <= timeLimit)
    {
        List<Client> toRemove = new ArrayList<>();
        for(Client t: generatedClients)
        {
            if(t.getArrivalTime() == currentTime)
            {
                toRemove.add(t);
                scheduler.dispatchTask(t);
            }
        }
        generatedClients.removeAll(toRemove);

        appendClientSituation(currentTime);
        currentTime++;

        appendQueueSituation(currentTime);
        printToGui();
```

Next up, the thread (simulation manager) will sleep for 1 second. Then the simulation check if it's time to end, and if so, will break from this loop and go to cleanup. Cleanup follows: statistics are appended, the queues are stopped and the situation is appended to a file.

```
//SLEEP
try {
        Thread.sleep(GLOBAL_TIME);
    } catch (InterruptedException e) {
        output.append("Simulation manager was interrupted and exiting!\n");
        printToGui();
        break;
    }
//CHECK IF END OF SIMULATION
if(toEnd)
{
    output.append("Simulation is over! Exiting early...\n");
    printToGui();
    break;
}
else if(generatedClients.size() == 0 )
{
    int remainingTasks = 0;
    for(Server server: scheduler.getServers())
    {
        if(server.getTasks().size() != 0)
        {
            remainingTasks += server.getTasks().size();
        }
    }
    if(remainingTasks ==0)
    {
        toEnd = true;
    }
}
```

The **GUI** window can be seen below. The top-level element is a border pane, while the rest of the elements are a mix of HBoxes and VBoxes. Labels are used for displaying static text and a Text Area is used for displaying the simulation time. Whenever the user presses start, validation checking is done in the background. A message will be displayed to the user if the input is invalid or otherwise not interpretable by the queue simulation manager. The abort button will stop an already running simulation and the reset button will clear the Text Area and prepare the simulation manager for another simulation. Thus, the simulation manager can be used multiple times without opening and closing the application.





The **AppController** class contains all the event handlers for the application. Each text field from the graphical user interface is mapped to a variable with similar name. The action handler from the start simulation button is mapped to onStartSimulation().

In this method, the input is validated, and error messages are generated as necessary. The validateAllInput() method does multiple checks. These checks are not exactly important, but in essence each text field is assigned a certain error id which is then passed into the method that creates the alert. Thus a personalized text is created every time a field is wrong so the user knows where the error occurred. There are many checks: whether or not

the field is an integer, whether it is positive, and the app does not allow 0 values for queues and neither reverse interval for processing time nor arrival time (min has to be at max arrival / processing time).

The abort button interrupts the simulation manager cleanly and appends the statistics. The reset button should not be used while the simulation is still running as it causes a hard crash of the simulation manager, however the application should still survive this as the queues will still be closed.

```
protected void onStartSimulation() {
    if(validateAllInput(true) != 0) return;
    startSimulationButton.setDisable(true);
    app = new SimulationManager(
            Integer.parseInt(simulationTime.getText()),
            Integer.parseInt(numberOfClients.getText()),
            Integer.parseInt(numberOfQueues.getText()),
            Integer.parseInt(minArrivalTime.getText()),
            Integer.parseInt(maxArrivalTime.getText()),
            Integer.parseInt(minProcessingTime.getText()),
            Integer.parseInt(maxProcessingTime.getText())
    );
    sim = new Thread(app);
    sim.start();
}
@FXML
protected void onAbortSimulation()
{
    if(sim.isAlive()) sim.interrupt();
    abortSimulationButton.setDisable(true);
}
```

5. Results

For testing purposes, I used the GUI alongside the data provided in the assignment description. The simulations ran a single time, so the results may vary depending on the scheduler and the random numbers that are generated. However, the same trends should be seen in any test. Without further ado, here are the results:

Test 1, 4 clients, 2 queues:
Simulation is over! Exiting early... (time 31)
Queue 1 has an average waiting time of 0,688
Queue 2 has an average waiting time of 0,000

Average waiting time for the queues in total is 0,344

Average service time for a task is: 2,750
Peak hour for the simulation is time 24 with an average waiting time of 2,000 between all the queues

Test 2, 50 clients, 5 queues:
Simulation is over! Exiting early... (time 49)
Queue 1 has an average waiting time of 3,320
Queue 2 has an average waiting time of 4,120
Queue 3 has an average waiting time of 3,980
Queue 4 has an average waiting time of 3,400
Queue 5 has an average waiting time of 3,440

Average waiting time for the queues in total is 3,652

Average service time for a task is: 4,120
Peak hour for the simulation is time 39 with an average waiting time of 6,800 between all the queues

Test 3, 1000 clients, 20 queues:
Queue 1 has an average waiting time of 126,060
Queue 2 has an average waiting time of 126,000
Queue 3 has an average waiting time of 124,259
Queue 4 has an average waiting time of 123,831
Queue 5 has an average waiting time of 124,896
Queue 6 has an average waiting time of 125,378
Queue 7 has an average waiting time of 125,269
Queue 8 has an average waiting time of 124,507
Queue 9 has an average waiting time of 127,353
Queue 10 has an average waiting time of 127,100
Queue 11 has an average waiting time of 124,318
Queue 12 has an average waiting time of 124,930
Queue 13 has an average waiting time of 124,866
Queue 14 has an average waiting time of 126,055
Queue 15 has an average waiting time of 125,607
Queue 16 has an average waiting time of 125,418
Queue 17 has an average waiting time of 125,433
Queue 18 has an average waiting time of 124,920
Queue 19 has an average waiting time of 124,458
Queue 20 has an average waiting time of 124,468

Average waiting time for the queues in total is 125,256

Average service time for a task is: 5,930
Peak hour for the simulation is time 100 with an average waiting time of 206,650 between all the queues

6. Conclusions

This homework was a good introduction to thread management, and it helped me understand the synchronization mechanisms in java better, also it was a breeze to work with such mechanisms since they adhere to the java high level way of doing things so many things about threads that would usually be kind of hard were actually quite easy. ArrayBlockingQueue interface and the AtomicInteger class allow for a very easy way of scheduling all the queues. Without further ado, here are the results:

As a future improvement, I could stylize the GUI more using CSS and perhaps show the border in red for invalid input like I did in the previous assignment, perhaps provide a history of operations, and maybe even provide a better interface for displaying the queue evolution, like plotting them on a graph or showing each queue on a line and waiting clients as bubbles.

7. Bibliography

1. https://dsrl.eu/courses/pt/ and everything contained within assignment 2 support presentation, assignment 2 resources, lecture notes, etc.

2. Java Programming Masterclass by Tim Buchalka – a course available on Udemy about Java concepts and programming techniques
3. https://stackoverflow.com/questions/tagged/java
4. Stack overflow for information on java threads and how to properly schedule threads in JavaFX
5. https://docs.oracle.com/javafx/2/threads/jfxpub-threads.htm
6. https://en.wikipedia.org/wiki/Non-functional_requirement#Examples