

# Queues Management Application

## Assignment 2

- Documentation -

Student: Alexandra Ilovan

Group: 30422

Laboratory Professor: Andreea-Valeria Vesa

## TABLE OF CONTENTS

---

1. Assignment Objectives.....	3
2. Problem Analysis. Modeling. Scenarios and Use Cases.....	4
3. Design.....	7
4. Implementation.....	10
5. Results.....	12
6. Conclusions.....	15
7. Bibliography.....	16

# 1. Assignment Objectives

---

## 1.1. Main Objective

The main objective of the given assignment is to correctly design and implement a queue management simulator, by analyzing queuing-based systems and determining the most efficient methods of minimizing the time clients are waiting in queues before being served. The application simulates a series of N randomly generated clients arriving for service, entering Q queues, waiting, being served and finally leaving the queues. Upon the completion of the simulation, the program also computes the average waiting time, average service time and peak hour.

The client management is done by using a strategy that implies choosing to send the new client at the queue that takes the shortest time to process the services of the clients that are already in the queue. The queues follow the FIFO (First In First Out) model.

## 1.2. Secondary Objectives

Among the secondary objectives of the assignment are the following:

- Choosing the class attributes and configuring the behavior of the created objects such that an accurate simulation of the real world is obtained
- Accurately implementing the concepts of clients and queues from the real world as classes
- Reading the tests' input data from a file (the file contains the number of clients, the number of queues, the maximum simulation time, the arrival and service time limits that a client has to respect)
- Creating and writing the simulation logs into an output file that will contain the evolution of the clients' status (the position in the queue/outside the queue, the remaining service time), updated at the passing of every second, and the simulation results represented by the average waiting and service times, and the peak hour (the time when the most clients were waiting in queues). By interpreting the generated output log, the user can observe the evolution of the queues with ease, not needing the aid of a graphical user interface exclusively
- The usage of interfaces such as Comparable (for sorting the generated list of clients) and Runnable (for using threads)
- Defining various strategies for a more efficient process of client distribution in queues (choosing to add a client either to the queue with the shortest waiting time, or to the queue that contains the smallest number of clients already waiting in line)
- Providing an intuitive graphical user interface for watching the evolution of the management process

## 2. Problem Analysis. Modeling. Scenarios and Use Cases

### 2.1. Problem Analysis

**Stating the problem:** Improper queue management leads to high waiting times for clients and inefficient usage of resources.

Queues have the aim of helping with the organization of objects or people in various scenarios. In this application, queues are used to offer a place for a client to stay in while waiting to be served. The problem that arises is minimizing the waiting time of the clients, in order to avoid wasting precious time that could have been used more effectively.

### 2.2. Modeling

In order to solve the stated problem, the implementation of efficient queue allocation mechanisms such as the addition of clients to queues that have a minimal waiting time (see Concrete Time Strategy) or to queues that have a minimal number of clients is required (see Concrete Queue Strategy).

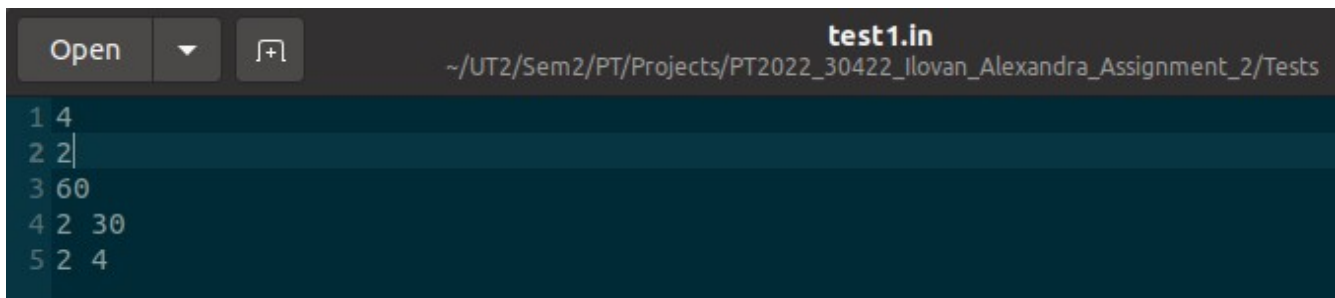
Since in the given case, the application aims to simulate the process of clients waiting to be served in a store as accurately as possible, the usage of special data structures such as atomic integers is needed (used to simulate the evolution of the queues in real time, but a more compacted version of it), as well as the introduction of multithreading (in order to be able to track the changes that occur for different queues simultaneously).

### 2.3. Scenarios and Use Cases

Upon running the application, a message that will prompt the user to choose one of three options will be displayed in the console. The three options represent the three simulation setups that were provided in the requirements file. The simulation setups are stored in three different input files (named as *test<nr>.in*) that are stored in the *Tests* directory of the project. The input data inside one of such files is provided in the following order:

- number of clients
- number of queues
- the maximum simulation time
- minimum arrival time and maximum arrival time of a client respectively (displayed as two numbers separated by a space)
- minimum service time and maximum service time of a client respectively (displayed as two numbers separated by a space)

Input file format:



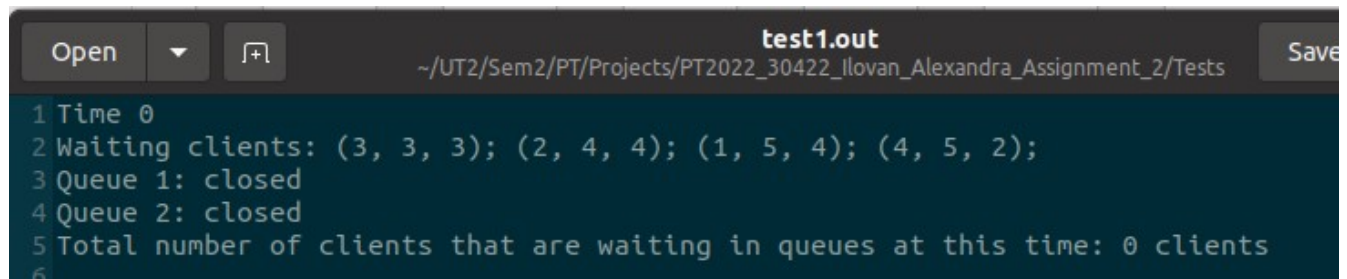
```
test1.in
~/UT2/Sem2/PT/Projects/PT2022_30422_Ilovan_Alexandra_Assignment_2/Tests
1 4
2 2
3 60
4 2 30
5 2 4
```

After choosing one of the three tests, entering its corresponding number into the console and pressing Enter, the user can expect one of two outputs.

If the user chooses to run test 1 or test 2, a graphical interface will be provided to assist in observing the evolution of the state of the queues and their respective client distribution in real time. A window that contains a real time counter, a list of waiting clients displayed in red font, and a list of Q queues with their corresponding clients in green font, will be opened. The clients will be represented as *(client ID, client arrival time, client service time)* tuples. The frame of the window will be constantly updated at each passing second and display a new client distribution. Once the simulation time limit is reached, the counter will freeze, and that means that the simulation can be stopped safely and that an output log file has been generated. The window provides three additional buttons, giving the user the option to either see the simulation results (opens a new dialog window that displays information about the average waiting and service times, as well as the peak hour of the given interval), open the generated output file containing a more detailed representation of the entire simulation in text format, made to allow the user to analyze the output data more carefully, or exit the application, stopping the simulation in the process.

If the user chooses to run the third test, however, the option of seeing the simulation in the graphical interface is not available, since illustrating a scenario with such large amounts of data is not possible. An output file will still be generated, and the user can inspect it by opening the corresponding file with the .out extension, found in the *Tests* directory.

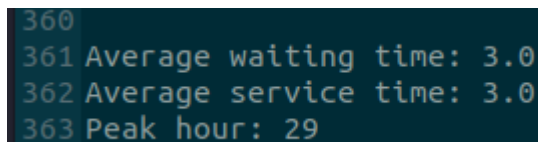
The format of the data displayed in an output file (at one given time instance) is the following:



```
test1.out
~/UT2/Sem2/PT/Projects/PT2022_30422_Ilovan_Alexandra_Assignment_2/Tests
1 Time 0
2 Waiting clients: (3, 3, 3); (2, 4, 4); (1, 5, 4); (4, 5, 2);
3 Queue 1: closed
4 Queue 2: closed
5 Total number of clients that are waiting in queues at this time: 0 clients
6
```

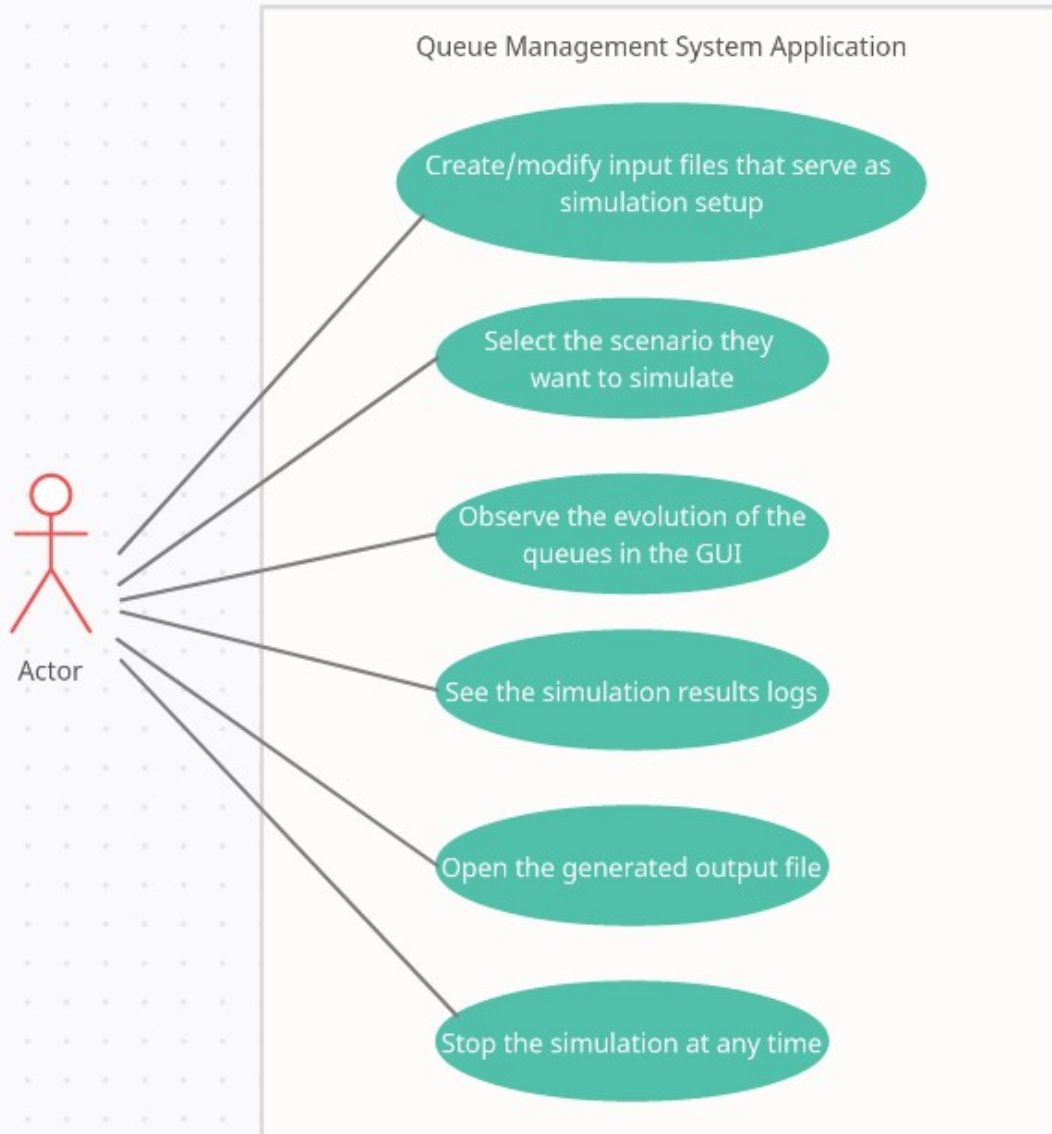
This same format is repeated for all the times until reaching the simulation time limit.

After presenting the logs for each moment of the simulation, at the end of the output file some average results are displayed, like such:



```
360
361 Average waiting time: 3.0
362 Average service time: 3.0
363 Peak hour: 29
```

## Use Cases Diagram

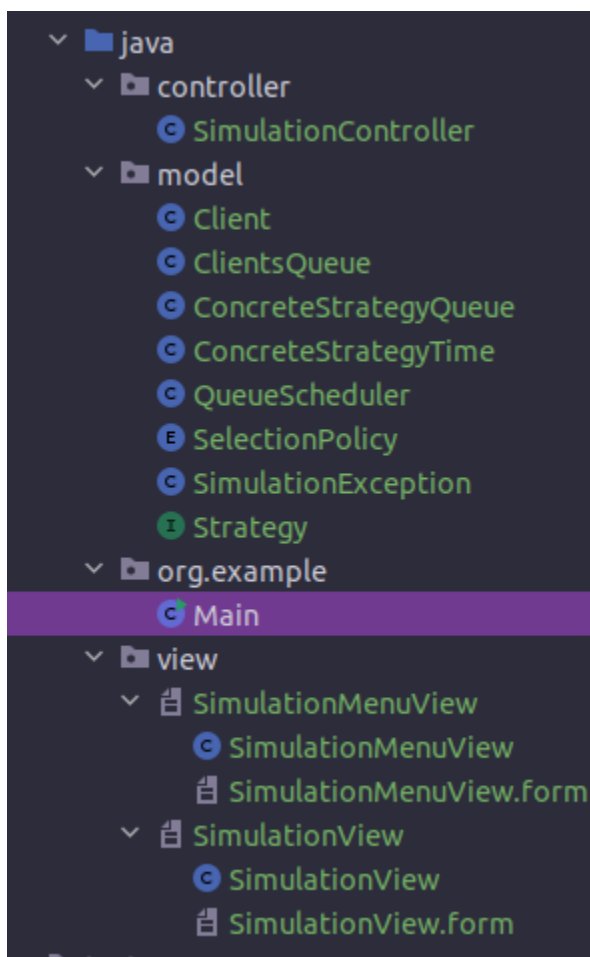


## 3. Design

### 3.1. Design Pattern. Design Decisions and Packages Distribution

The main focus of this application, when it comes to design patterns, was the usage of object-oriented design. Object-oriented design is the process of planning a system of interacting objects for the purpose of solving a software problem. Put in simpler terms, it is a paradigm of transposing objects that are part of the real world into packages and classes. The application also follows a slightly modified version of the MVC (Model View Controller) architecture.

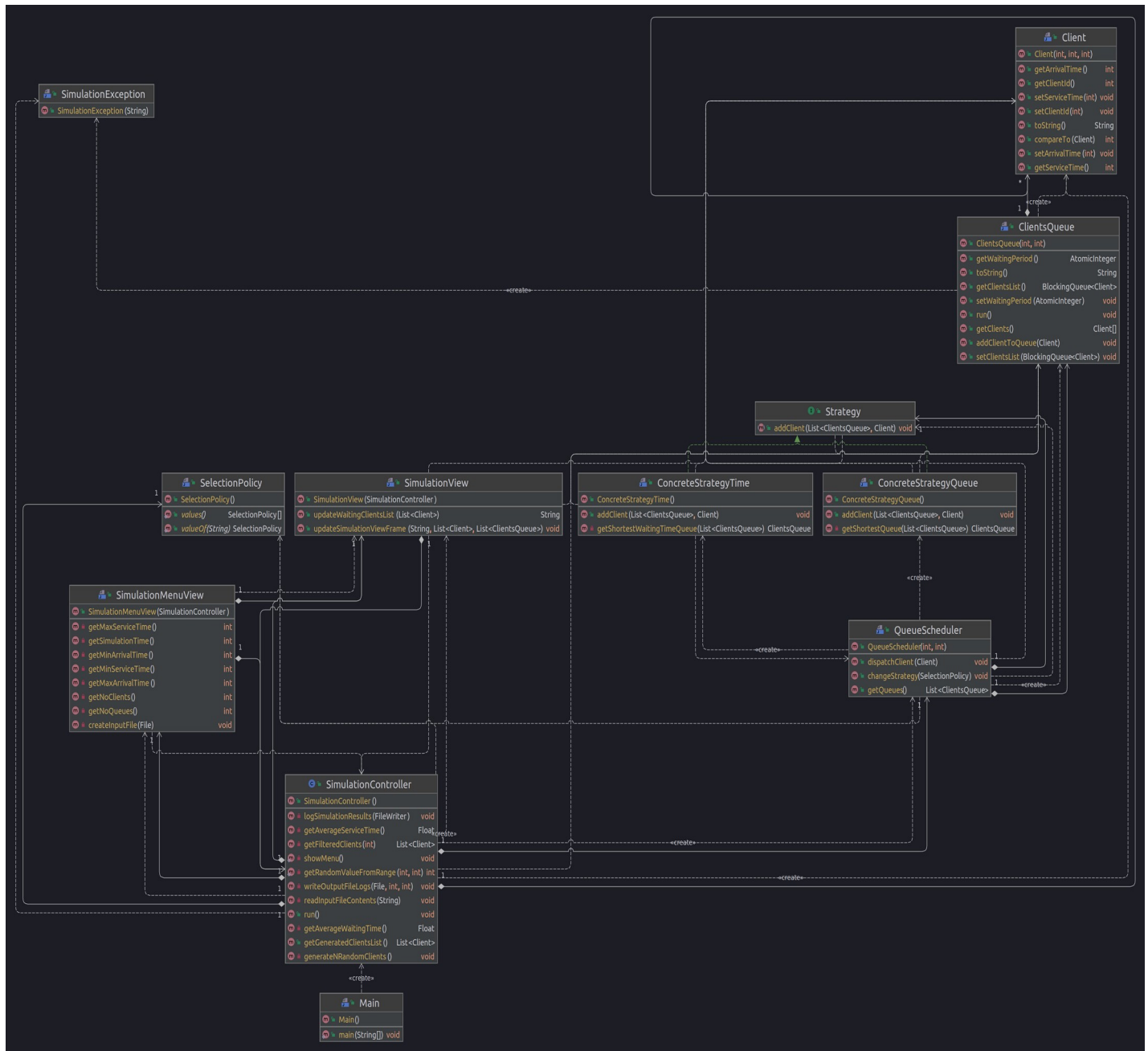
The application is structured in four main packages: **controller**, **model**, **view** and **org.example** (package containing the **Main** class), in which the following classes are included:



The **model** package contains the Strategy Interface and the implementation of the classes that lead to the proper functionality of the entire simulation system (the used strategies, represented by the **ConcreteStrategyQueue** and **ConcreteStrategyTime** classes, and the **QueueScheduler** class, used for creating and handling multiple queues, as well as for initializing their respective threads), and an enumeration **SelectionPolicy**, used for choosing the desired strategy for assigning clients to queues. The **Client** and **ClientsQueue** classes are quite self-explanatory, the first one being the object representation of a real client, and the latter being the representation of a queue from a store.

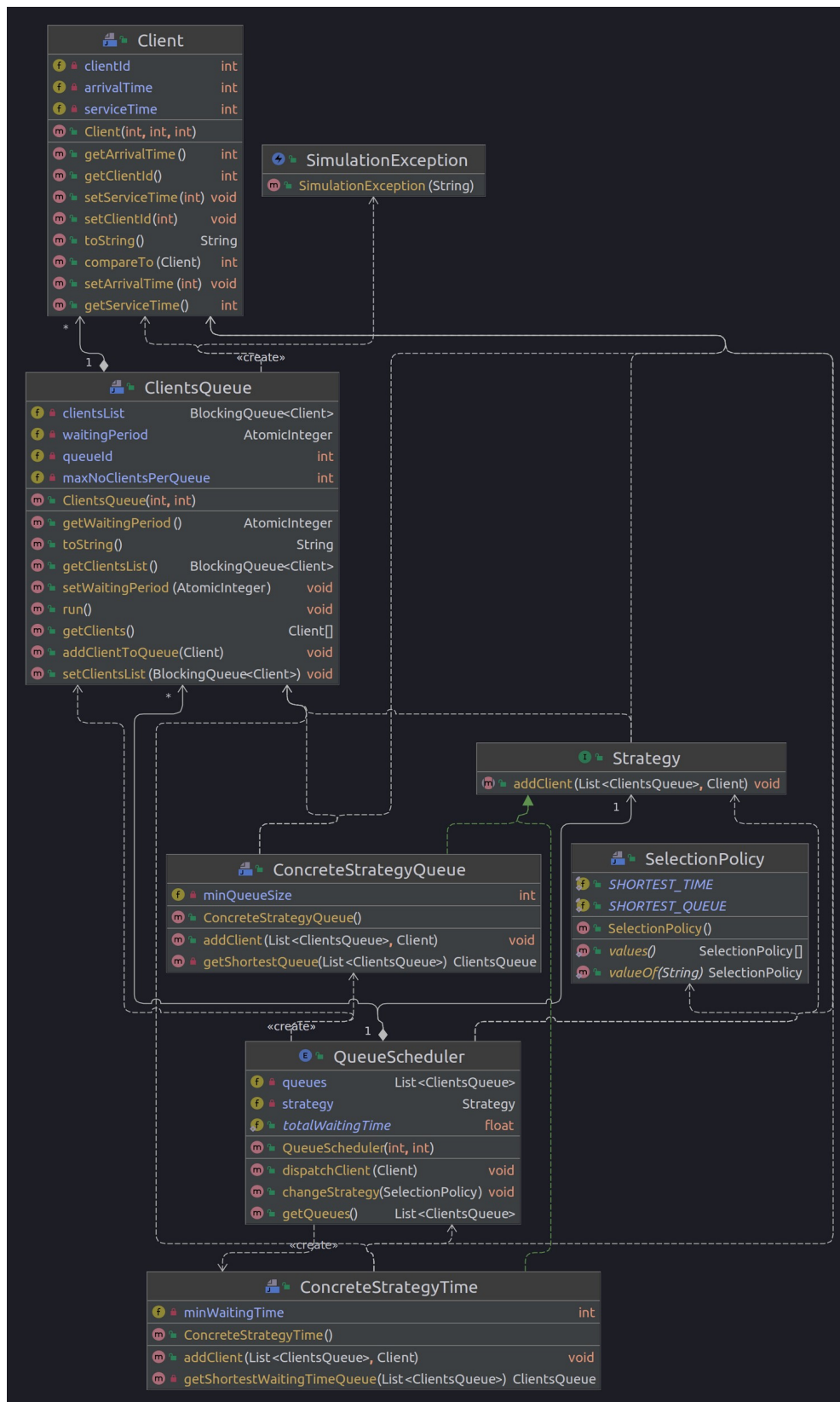
The view package contains the class **SimulationView**, which represents the graphical user interface (GUI) of the application, on which the method of data processing and the client-queue evolution is presented. The **SimulationMenuView** class has yet to be finished as a future improvement of the program. Its aim is to allow the user to input a set of input data for the simulation setup himself, but the application does not provide this functionality yet. The **SimulationController** class from the controller package is what the functionality of the program mainly revolves around, since connects all the pieces together (input and output file management, the random client generator, and client ordering/ filtering, interactions with the user, establishing the ties with the GUI, thread management and so on).

## 3.2. UML Diagram





In the following image is a more detailed representation of the relationships that take place within the model package alone:



## 4. Implementation

---

### Main Class Implementation

The Main class contains the main() method, in which an instance of SimulationController is defined and launched towards execution in a thread.

### SimulationController Class Implementation

This class represents the controller of the application, as it deals with reading and writing data into files, as well as generating the data that is going to be processed. Among the main attributes of the class are the following:

- noClients – the total number of clients, or tasks, that have to be generated
- noQueues – the total number of queues, or servers
- maxSimulationTime – the time limit of the simulation; when surpassing this limit, the execution stops, despite having or not having finished serving all the clients
- minArrivalTime and maxArrivalTime – the limited range from which a random arrival time of a client has to be generated
- minServiceTime and maxServiceTime – the limited range from which a random processing time of a client's order has to be generated
- peakHour – stores the value of the time at which the most clients were waiting in line for their turn to come
- averageWaitingTime – stores the value of an average time it takes all the clients to wait for their turn in a queue
- averageServiceTime – stores the average time it take to process all the clients' orders

There are also some additional attributes used for file operation management, as well as queue scheduling (selectionPolicy, which represents the strategy of client selection when being added to a queue – in this case it is always equal to SHORTEST\_TIME, queueScheduler, which is an instance of the class that deals with the client scheduling and their distribution to queues).

The constructor of the class begins by calling a method that displays the menu of options the user can choose, and depending on the user input, configures the input and output files of the tests. It then creates an instance of the QueueScheduler object, generates N random clients based on the processed input data from the files, and, in the case that the user chose either one of options 1 or 2, it will also create an instance of SimulationView.

The random client generating process is done through the methods getRandomValueFromRange(int minLimit, int maxLimit) (which returns a random number from the [minLimit, maxLimit] range) and generateNRandomClients().

The method getFilteredClients(int currentTime) returns a list of the clients that have their arrival times equal to the current time, so that the program will know to insert those clients in the most efficient way in the queues.

The method readInputFileContents(String inputFilePath) receives as argument a String which represents the absolute or relative path of the input file and starts storing the read values in an ArrayList, so that it can later pass the values to the corresponding main attributes of the class. The method works by reading the file line by line, and by splitting the line (using the method split(regex), where regex = " ") whenever it encounters a space character. This splitting procedure was required in order to extract the arrival and service time limits.

The method writeOutputFileLogs(File outputFile, int realTime, int logCode) receives as arguments the output file the logs and results will be written to, the current time of the simulation and a special code (logCode) used for differentiating the data that is written depending on its value.

The run() method carries the most relevance, since it manages the execution of the thread that contains the instance of the class from the main() method. This method selects the clients to be added to a queue in order of their arrival times. The method runs while there are still clients whose orders need to be processed or until the time limit has been reached.

## **ConcreteStrategyQueue & ConcreteStrategyTime Class Implementation**

ConcreteStrategyQueue offers a strategy of client selection that implies choosing the queue with the least waiting clients, while ConcreteStrategyTime chooses the queues with a minimum processing time, which is also the strategy the given assignment requires us to use. Both of these implement the Strategy Interface, which describes the behavior of the strategies through its only method addClient().

## **Scheduler Class Implementation**

The constructor of this class receives the maximum number of queues and the maximum number of clients per queue, and generates the respective queues along with a thread for each one, following to be launched towards execution. The changeStrategy() method can be used in future developments for choosing a different strategy of queue selection.

## **ClientsQueue Class Implementation**

When it comes to attributes, this class contains a queue ID (queueID), the maximum number of clients allowed per queue (maxNoClientsPerQueue), a list of the clients that are waiting in the queue (of BlockingQueue<Client> type) and an AtomicInteger (waitingPeriod) in which the waiting time of the queue in question is computed at every step. The class implements the Runnable Interface and overrides the run() method. In this method all the threads are being synchronized by calling sleep(1000), making the thread sleep for one second in real time, and after each second that passes, the service time of the client at the head of the queue is being decremented by 1.

## **Client Class Implementation**

The Client class contains information about the client (ID, arrival time, service time) and getters and setters that help retrieve or modify this information. The class implements the Comparable Interface, so that it can sort the clients in the ascending order of their arrival times. The sorting operation is done once all the random clients are generated.

## **SimulationView Class Implementation**

The SimulationView class has all the components of the visible on the panel as attributes, and its constructor configures the frame settings needed for its display on the screen. It creates three action listeners: one for the seeLogs button (opens a message dialog that displays the average waiting time, average service time and peak hour), one for the openLogFile button (which opens the output log file by finding its path in the computer of the user), and one for the exit button, which stops the simulation and exits the application. The updateSimulationViewFrame() method is the one that updates the frame of the window accordingly as time progresses, so that the user can follow the simulation with more ease.

## 5. Results

The assignment requirements do not specify the need for JUnit testing classes to exist. In the following images, an example of the results obtained when running the first test are shown:

For this input:

```
test1.in
1 4
2 2|
3 60
4 2 30
5 2 4
```

The following output is obtained:

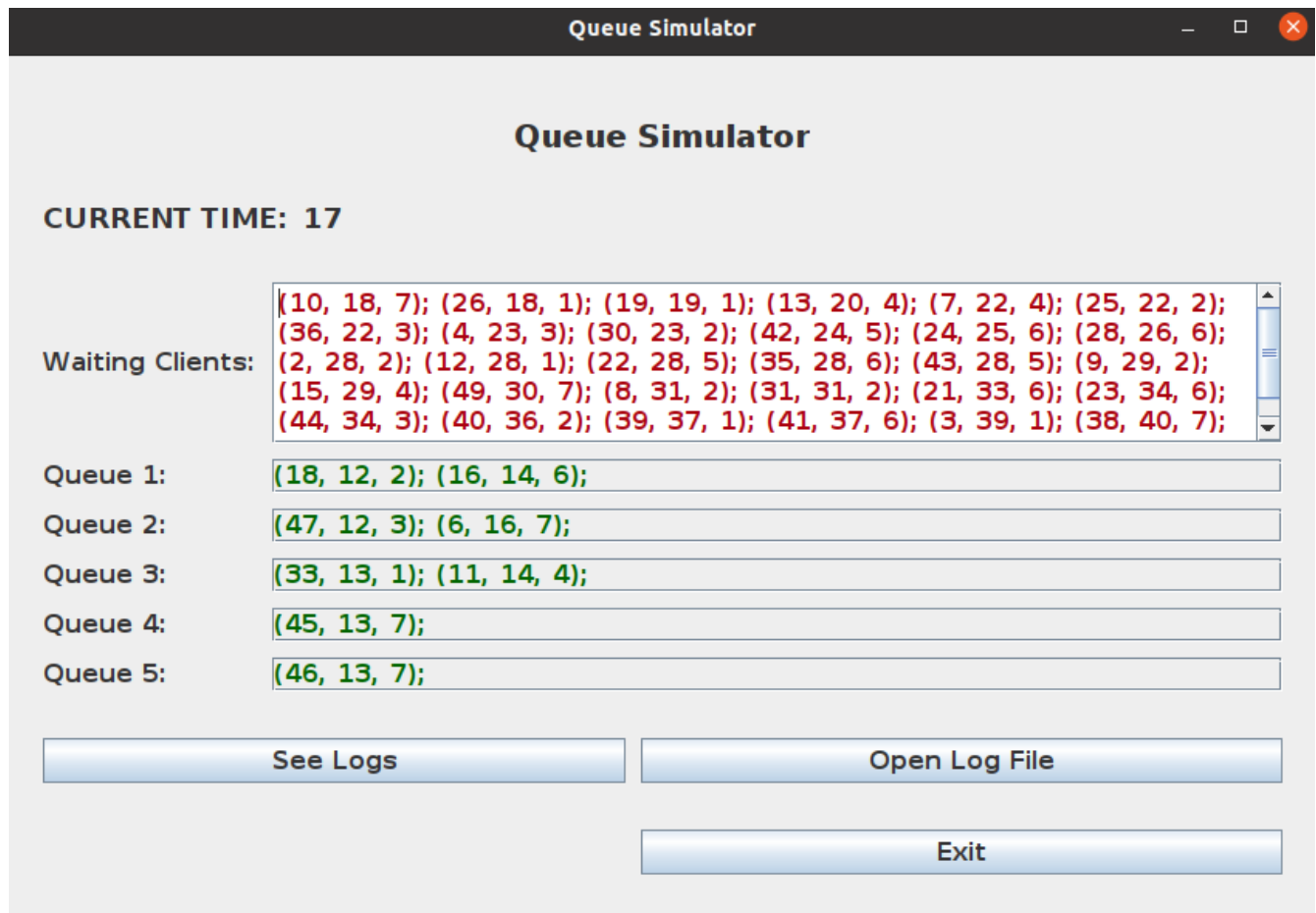
```
1 Time 0
2 Waiting clients: (2, 3, 4); (1, 9, 4); (4, 17, 2); (3, 28, 2);
3 Queue 1: closed
4 Queue 2: closed
5 Total number of clients that are waiting in queues at this time: 0 clients
6
7 Time 1
8 Waiting clients: (2, 3, 4); (1, 9, 4); (4, 17, 2); (3, 28, 2);
9 Queue 1: closed
10 Queue 2: closed
11 Total number of clients that are waiting in queues at this time: 0 clients
12
13 Time 2
14 Waiting clients: (2, 3, 4); (1, 9, 4); (4, 17, 2); (3, 28, 2);
15 Queue 1: closed
16 Queue 2: closed
17 Total number of clients that are waiting in queues at this time: 0 clients
18
19 Time 3
20 Waiting clients: (1, 9, 4); (4, 17, 2); (3, 28, 2);
21 Queue 1: (2, 3, 4);
22 Queue 2: closed
23 Total number of clients that are waiting in queues at this time: 1 clients
24
25 Time 4
26 Waiting clients: (1, 9, 4); (4, 17, 2); (3, 28, 2);
27 Queue 1: (2, 3, 3);
28 Queue 2: closed
29 Total number of clients that are waiting in queues at this time: 1 clients
30
31 Time 5
32 Waiting clients: (1, 9, 4); (4, 17, 2); (3, 28, 2);
33 Queue 1: (2, 3, 2);
34 Queue 2: closed
35 Total number of clients that are waiting in queues at this time: 1 clients
36
37 Time 6
38 Waiting clients: (1, 9, 4); (4, 17, 2); (3, 28, 2);
39 Queue 1: (2, 3, 1);
40 Queue 2: closed
41 Total number of clients that are waiting in queues at this time: 1 clients
42
43 Time 7
44 Waiting clients: (1, 9, 4); (4, 17, 2); (3, 28, 2);
45 Queue 1: closed
46 Queue 2: closed
47 Total number of clients that are waiting in queues at this time: 0 clients
48
49 Time 8
50 Waiting clients: (1, 9, 4); (4, 17, 2); (3, 28, 2);
```

```

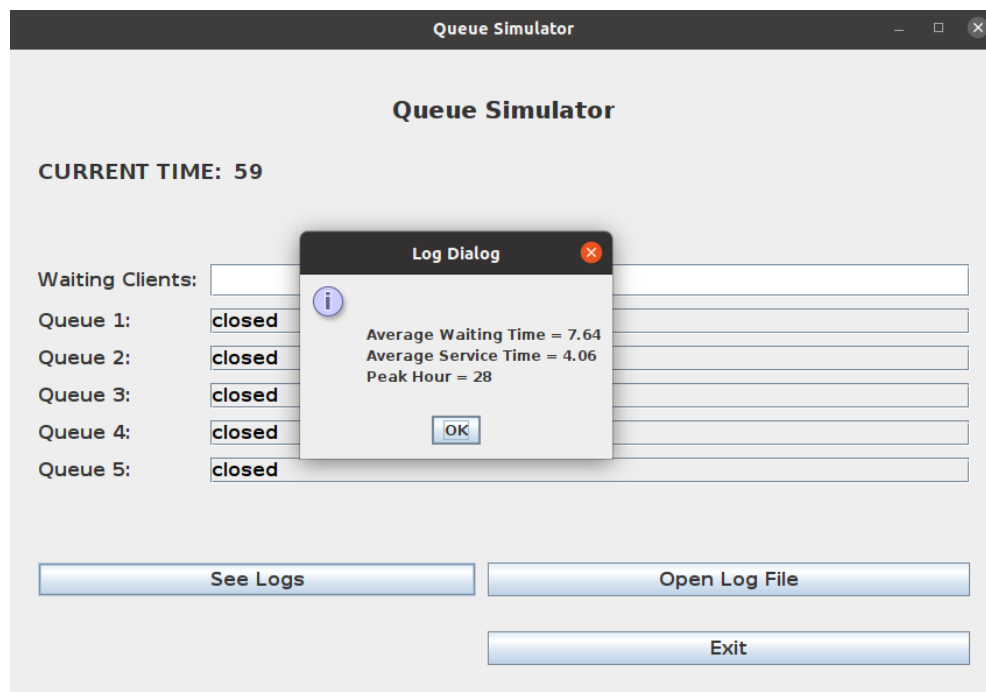
49 Time 8
50 Waiting clients: (1, 9, 4); (4, 17, 2); (3, 28, 2);
51 Queue 1: closed
52 Queue 2: closed
53 Total number of clients that are waiting in queues at this time: 0 clients
54
55 Time 9
56 Waiting clients: (4, 17, 2); (3, 28, 2);
57 Queue 1: (1, 9, 4);
58 Queue 2: closed
59 Total number of clients that are waiting in queues at this time: 1 clients
60
61 Time 10
62 Waiting clients: (4, 17, 2); (3, 28, 2);
63 Queue 1: (1, 9, 3);
64 Queue 2: closed
65 Total number of clients that are waiting in queues at this time: 1 clients
66
67 Time 11
68 Waiting clients: (4, 17, 2); (3, 28, 2);
69 Queue 1: (1, 9, 2);
70 Queue 2: closed
71 Total number of clients that are waiting in queues at this time: 1 clients
72
73 Time 12
74 Waiting clients: (4, 17, 2); (3, 28, 2);
75 Queue 1: (1, 9, 1);
76 Queue 2: closed
77 Total number of clients that are waiting in queues at this time: 1 clients
78
79 Time 13
80 Waiting clients: (4, 17, 2); (3, 28, 2);
81 Queue 1: closed
82 Queue 2: closed
83 Total number of clients that are waiting in queues at this time: 0 clients
84
85 Time 14
86 Waiting clients: (4, 17, 2); (3, 28, 2);
87 Queue 1: closed
88 Queue 2: closed
89 Total number of clients that are waiting in queues at this time: 0 clients
90
91 Time 15
92 Waiting clients: (4, 17, 2); (3, 28, 2);
93 Queue 1: closed
94 Queue 2: closed
95 Total number of clients that are waiting in queues at this time: 0 clients
96
...
354
355 Time 59
356 Waiting clients: No clients in line
357 Queue 1: closed
358 Queue 2: closed
359 Total number of clients that are waiting in queues at this time: 0 clients
360
361 Average waiting time: 3.0
362 Average service time: 3.0
363 Peak hour: 29

```

When running a test and using the graphical interface as well, the process looks like this:



If the user chooses to see the log simulation results as well, the following is shown:



## 6. Conclusions

---

By working on this assignment, I have gained a significant amount of valuable knowledge, such as understanding multithreading (how threads work and how to manage and synchronize them). I have also improved my ability to connect real life concepts with coding, which made the working process be more enjoyable since I could actually make such challenging connections. I also practiced using the object-oriented design pattern for the first time.

Some possible further implementations for the project include adding the feature to allow the user to select the simulation setup variables, and implementing more queue and client selection strategies.

## 7. Bibliography

---

- the provided support presentation

[https://www.w3schools.com/java/java\\_files\\_read.asp](https://www.w3schools.com/java/java_files_read.asp)

<https://www.geeksforgeeks.org/how-do-i-generate-random-integers-within-a-specific-range-in-java/>

[https://en.wikipedia.org/wiki/Object-oriented\\_design](https://en.wikipedia.org/wiki/Object-oriented_design)

<https://www.javatpoint.com/how-to-open-a-file-in-java>