Fundamental Programming Techniques

Assignment 2

Queues Simulator

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA, ROMANIA

Student: Filip Denisa-Mariana
Group: 30422
Semigroup: 1
Lecturer: Dr. Eng. Prof. Salomie Ioan
Teaching Assistant: Stan Ciprian
Faculty of Automation and Computer Science, TUCN
2020-2021

Table of Contents

1. Assignment Objective

The main objective of this assignment is to design and implement a simulation application aiming to analyse queuing based systems for determining and minimizing clients' waiting time.

The secondary objectives of this assignment are:

Secondary Objective	Chapter in which it is detailed
Analyze the problem and identify requirements	2
Design the queue simulator	3
Implement the queue simulator	4

1.1. Description of secondary objectives

- **Analyze the problem and identify requirements** – in the beginning, a problem is formulated into natural language. Prior to the design process, research into the theoretical background of the problem is needed. Afterwards, the problem needs to be broken down into functional and non-functional requirements, alongside with the possible scenarios and use cases that arise.
- **Design the queues simulator** – the solution needs to firstly be designed so that its implementation is one of the best ones possible. There are many steps to be completed before starting to implement a problem, such as choosing the optimal architectural patterns and design patterns, establishing the relationships between classes, the algorithms and data structures to be used and so on.
- **Implement the queues simulator** – during this step, the classes with their respective fields and methods are created according to the previously chosen design.

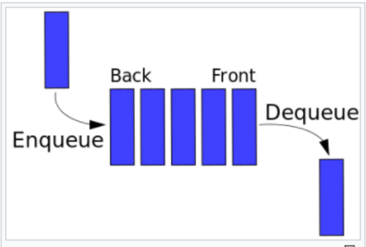
2. Problem analysis, modeling, scenarios, use cases

2.1. Problem Analysis

The problem identified is the management of queue-based systems, that aim to minimize the total time a “client” spends waiting until they receive their service. The main objective of a queue is to provide a place for a client to wait before being served.

To understand the specifics of the given problem, a theoretical background of the required terms is given.

In computer science, a queue is a collection of entities that are maintained in a sequence and can be modified by the addition of entities at one end of the sequence and the removal of entities from the other end of the sequence. The end of the sequence at which elements are added is called the back, tail or read of the queue and the end at which elements are removed is called the head or from of the queue. (Wikipedia)



Representation of a FIFO (first in, first out) queue

Such queues are commonly used to model real world domains. Specifically, in the context of the identified problem for this assignment, an element/entity from a queue is associated with a client that lines up to wait for a service.

Another important notion that stays at the foundation of this project is **Java Concurrency**. The Java programming language and the Java virtual machine (JVM) have been designed to support concurrent programming, and all execution takes place in the context of threads.In computer science, a **thread of execution** is the smallest sequence of programmed instructions that can be managed independently by the scheduler. Objects and resources can be accessed by many separate threads; each thread has its own path of execution but can potentially access any object in the program. (Wikipedia)

The most important aspect of working with threads is **thread safety**. The programmer must ensure that the read and write access to objects is properly synchronized between threads, so that no shared resources get modified concurrently. Thread synchronization ensures that objects are modified by only one thread at a time and that threads are prevented from accessing partially updated objects during modification by another thread. The Java language has built-in constructs to support this coordination.

2.2. Modeling

The identified problem is modelled using a set of input parameters inserted by an external user prior to the start of the simulation. The input data is described by the following:

- Number of clients (N);
- Number of queues (Q);
- Simulation interval ($t_{simulation}^{MAX}$);
- Minimum and maximum arrival time ($\mathrm{t}_{\mathrm{arrival}}^{\mathrm{MIN}} \leq \mathrm{t}_{\mathrm{arrival}} \leq \mathrm{t}_{\mathrm{arrival}}^{\mathrm{MAX}}$);
- Minimum and maximum service time ($\mathrm{t}_{\mathrm{service}}^{\mathrm{MIN}} \leq t_{service} \leq \mathrm{t}_{\mathrm{service}}^{\mathrm{MAX}}$);

These input parameters will be inserted by the user in the application's user interface. According to the input data, N clients will stand in a waiting queue, waiting to be served at one of the Q queues generated. Each client i is defined by the following tuple: (ID_i, $t^{i}_{arrival}$, $t^{i}_{service}$), with the following constraints:

- $1 \le ID_i \le N$
- $t^{MIN}_{arrival} \le t^{i}_{arrival} \le t^{MAX}_{arrival}$
- $t^{MIN}_{service} \le t^{i}_{service} \le t^{MAX}_{service}$

Each one of the Q queues is also defined by an ID ($1 \le ID_i \le Q$). The simulation of the queues starts from 0 and goes to $t^{MAX}_{simulation}$. The external user will also choose, through the graphical user interface, the strategy by which a client chooses at queue to go to. The two possible strategies, that aim to minimize a client's waiting time, thus always choose the shortest available queue, are:

- *smallest number of clients:* In this case, a queue is defined by the number of clients in its sequence. The shortest queue is the one with the smallest number of clients.
- *shortest waiting time*: In this case, a queue is defined by the total waiting time of the clients in its sequence (the total waiting time is obtained by summing up the service time of all the clients from the queue). The shortest queue is the one with the shortest waiting time.

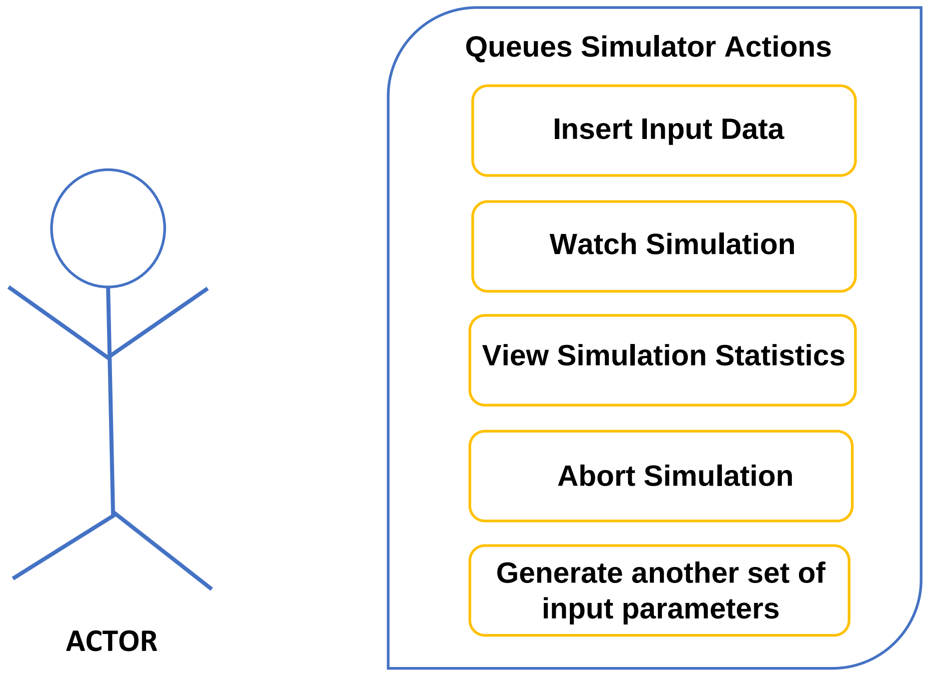
Finally, the simulation will end when either $t^{MAX}_{simulation}$has been reached or when all of the N generated clients have been served.

2.3. Scenarios

During a simulation, a few different scenarios can arise. In the preferred case, the user will insert valid input parameters, press the *Done* button to change to the Simulation Scene, press the button *Start Simulation* to generate all the data and to start the simulation. During the simulation, a user can only do two things: wait for the simulation to finish whilst watching the real time evolution of the queues, or press the *Back* button, that will change the scene back to the initial one, where another set of input parameters can be inserted. After the simulation is done, the user can either press the *View simulation statistics* button, that will start another JavaFX scene (containing the computed statistics of the simulation) or can press the *Back* button and be taken to the initial scene.

However, a user can also introduce invalid input parameters, which demands the verification and validation of the user input to be of utmost importance. In this case, the validation of the input data takes place, and an appropriate error message is generated. In another scenario, the user opens the queues simulator scene and closes it immediately.

2.4. USE CASES



Use case: Observe Queues Simulation and view statistics of simulation
Primary actor: user
Main Success Scenario:

- the user inputs the required input parameters from their keyboard, in the first scene of the GUI;
- the user presses the *Done* button;
- the user is redirected to the scene that displays the real time evolution of the simulation;
- the user presses the *Start Simulation* button;
- the N clients and Q queues are generated and the simulation starts from the second 0 – the user watches the clients receiving their services;
- when the simulation is done, the user can presses the *View Simulation Statistics* button;
- the user is redirected to a new scene, containing the three computed statistics of the finished simulation.

Alternative scenario:

- the user inserts incorrect input data;
- an alert dialog window pops up, informing the user of the place where the inserted data is invalid;
- the user has to insert the input parameters again, until all of them meet the requirements;

Alternative scenario:

- in the Simulation scene, the user presses the *Back* or *X* button of the window;
- the user is redirected to the primary scene, Input Data, and the background thread is interrupted;
- the user can then insert another set of input parameters.

After defining the main use case, the following functional and non-functional requirements were identified:

Functional Requirements	Non-functional requirements
insertion of input parameters by user	intuitively named fields
pressing the Done button, by the user	a Back button that takes the user to the previous window
generation of the clients and queues	a pleasant to look at interface
unfolding the simulation	a scene responsible for displaying the simulation statistics
displaying the real time evolution of the simulation	the ability to reinsert the input parameters and start another simulation
handling exceptions, such as invalid input data	

3. Design

3.1. Design Decisions

To divide my project into intuitive packages, I have chosen to implement the MVC (Model View Controller) architectural pattern. MVC is a interactive system pattern that divides a project into three categories: input, output and process. I will describe each of them individually:

- Under the view package reside three .fxml file, each corresponding to the GUI of the Queues Simulator phases:
  - an window for inserting the input parameters;
  - an window for displaying the real time evolution of the queues
  - an window for displaying the statistics of the simulation, after it is finalized.
- Under the controller package lie three Controller java classes, since each view is associated with only one controller;
- Under the model package I have created my own classes that model the problem given in natural language into code. This package is further divided into Enums, Comparators, Strategy, Simulation, Server and Task directories.

The main classes of the project are **Client** and **Queue**, alongside **Scheduler** and **SimulationManager**. The classes Client and Queue model the real-life behaviour of their real-life concepts. These classes will be further described under the Data Structures and Implementation sections. The **RandomClientGenerator** class generates clients with random values for their fields, according to the input parameters received.

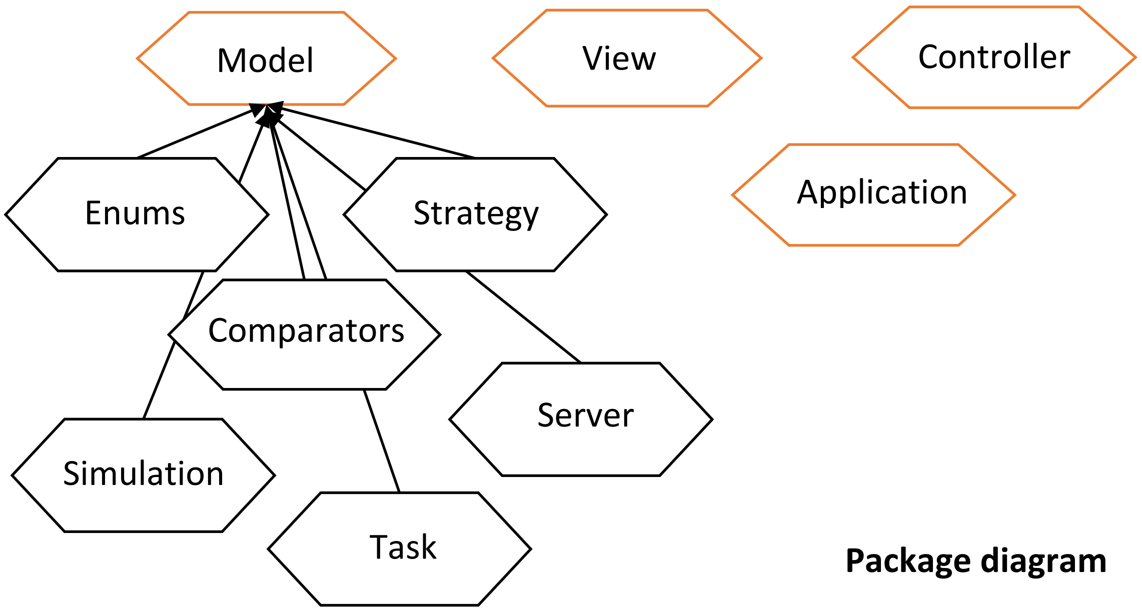
The **Scheduler** class is responsible for sending clients to queues in order to receive their service. This is done through the **Strategy behavioural design pattern**. This pattern allows the behaviour or algorithm of a class to be changed at runtime. To integrate this pattern into my project, I have created one interface, called **Strategy**. This interface is implemented by another two classes – **ConcreteStrategyQueue** and **ConcreteStrategyTime**.

The **SimulationStatistics** class is tasked with computing the statistics of a finished simulation, the data being of interest to the external user.

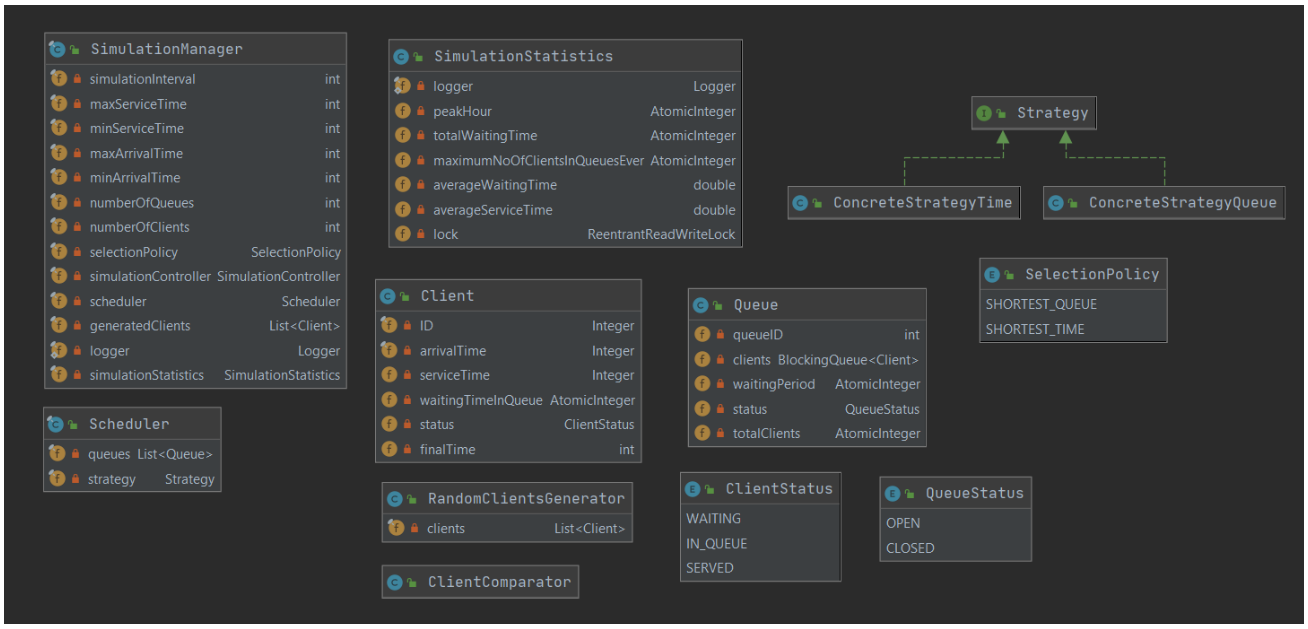
The **SimulationManager** is the main thread of the application, that is responsible for putting every action into motion. It starts each thread represented by the Queue and through the Scheduler class, handles the oncoming clients until the simulation is finished.

Splitting the application into the aforementioned classes helped to avoid their clustering with too many tasks. It also improves the over-all understandment and readability of the project.

3.2. UML Diagrams

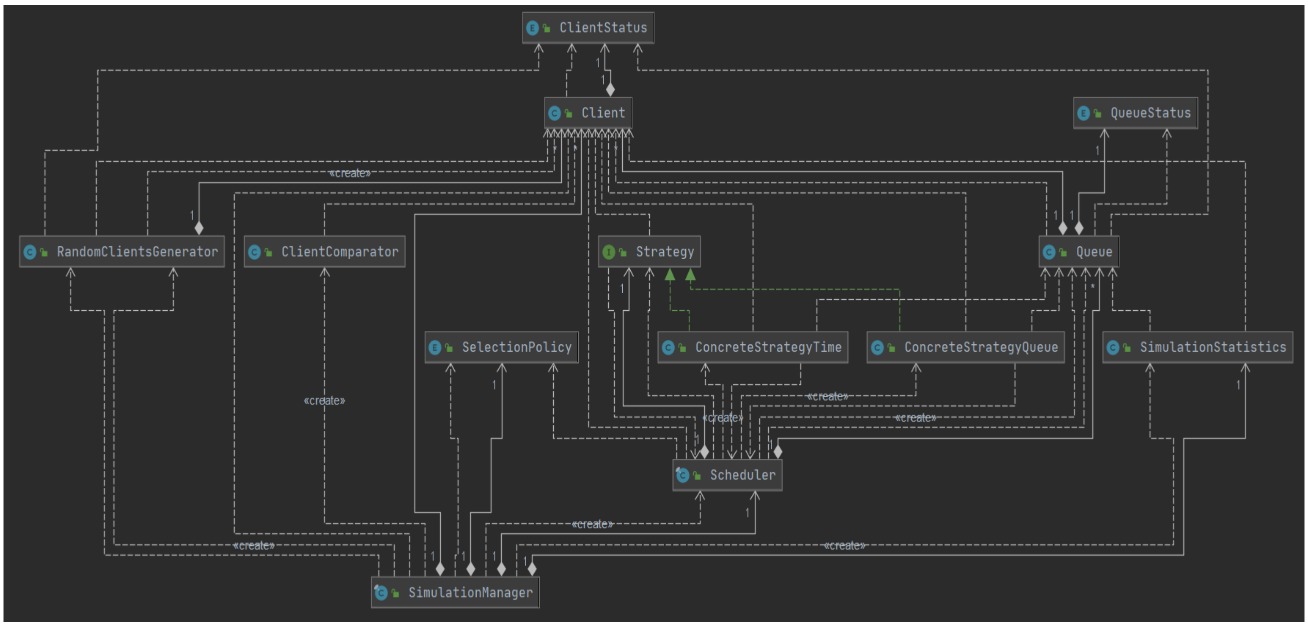


Package diagram



```
SimulationManager
  simulationInterval                    int
  maxServiceTime                        int
  minServiceTime                        int
  maxArrivalTime                        int
  minArrivalTime                        int
  numberOfQueues                        int
  numberOfClients                       int
  selectionPolicy           SelectionPolicy
  simulationController SimulationController
  scheduler                       Scheduler
  generatedClients             List<Client>
  logger                             Logger
  simulationStatistics SimulationStatistics

Scheduler
  queues   List<Queue>
  strategy    Strategy

SimulationStatistics
  logger                                        Logger
  peakHour                               AtomicInteger
  totalWaitingTime                       AtomicInteger
  maximumNoOfClientsInQueuesEver         AtomicInteger
  averageWaitingTime                            double
  averageServiceTime                            double
  lock                          ReentrantReadWriteLock

Client
  ID                          Integer
  arrivalTime                 Integer
  serviceTime                 Integer
  waitingTimeInQueue    AtomicInteger
  status                 ClientStatus
  finalTime                       int

RandomClientsGenerator
  clients   List<Client>

ClientComparator

Queue
  queueID                            int
  clients        BlockingQueue<Client>
  waitingPeriod            AtomicInteger
  status                     QueueStatus
  totalClients             AtomicInteger

Strategy
ConcreteStrategyTime
ConcreteStrategyQueue

SelectionPolicy
  SHORTEST_QUEUE
  SHORTEST_TIME

ClientStatus
  WAITING
  IN_QUEUE
  SERVED

QueueStatus
  OPEN
  CLOSED
```

Class Diagram with Fields



Class Diagram with Dependencies

3.3. Data Structures

All throughout the project, I have used the List and ArrayList Collections, replacing the conventional arrays. I have chosen to do so because an ArrayList is a variable length, resizing each them an element is removed or added to the Collection. At the same time, the removal of objects is faster for the ArrayList.

Moreover, to ensure thread safety, I have used the BlockingQueue Collection for holding the clients in a queue until they were server. ConcurrentMap and ConcurrentHashMap were also useful for mapping the UI elements (such as labels, groups or VBoxes) to their equivalent structures from the Model package.

Regarding my own data structures, I have created Queue and Client. These objects correspond to their real-life notions and are the center of the project. They are the objects used for every action performed.

3.4. Class and Interface Design

Each class contains some of the following:

- private fields;
- constructors (with and without arguments);
- methods;
- the overriden toString method;
- the getters and setters were generated through the Project Lombok library.

The interfaces Strategy defines the method that will be overriden by the classes that implement it. The classes Queue and SimulationManager, that implement the Runnable interface, also provide their own take of the run() method.

3.5. Relationships

There exist Is-A relationships between the Strategy interface and the classes that implement it.

Between the classes Queue and Client classes there is a Has-A relationship (aggregation). Since Queues have a List of Clients waiting, the relationship is one to many (one queue can have multiple clients at a time). There also are some dependency relationships between classes, especially between SimulationManager and the other classes, since SimulationManager puts every resource into motion.

3.6. Algorithms

For this project, there weren't any complicated algorithms that had to be used. The queue at which the next client will wait can be chosen in two ways:

- **smallest number of clients** – the number is obtained by computing the total number of clients already enqueued and that haven't been served yet.
- **shortest waiting time** – the number is obtained by summing up the service times of all of the clients already enqueued. To this sum, the service time of the current client is also added.

For the simulation statistics, the following computations were used:

- **peak hour** – the total number of clients currently in queues is computed at every second and it is compared with the maximum value met so far. If the new number is greater than the maximum, the maximum changes its value and the peak hour (i.e. second) is remembered in variable. This process is carried out the entire simulation.
- **average service time** – the service times of all of the generated initial clients are summed up into a variable and then divided to N (the total number of clients);
- **average waiting time:**
    - the waiting time is characterized by the addition of the sum of the service time of the clients already in queue and the service time of the first waiting client.
    - the waiting time is computed for every client;
    - these waiting times are added at the end of the simulation and the sum is divided to N (the total number of clients).

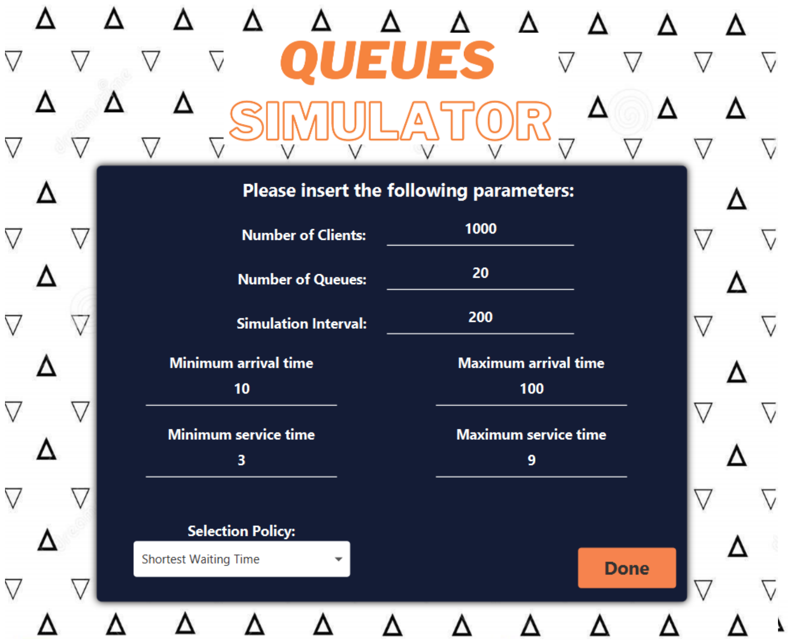
For each operation performed on polynomials, an algorithm is used. Whereas addition, subtraction, multiplication, integration and differentiation are quite trivial, the division operation is more complicated.

3.7. User Interfaces

The Graphical User Interface was implemented using JavaFX and its tool, SceneBuilder. I have chosen the JavaFX software platform for creating and delivering desktop applications because it is constantly improved and it will replace the standard GUI library for Java SE in the forseeable future.

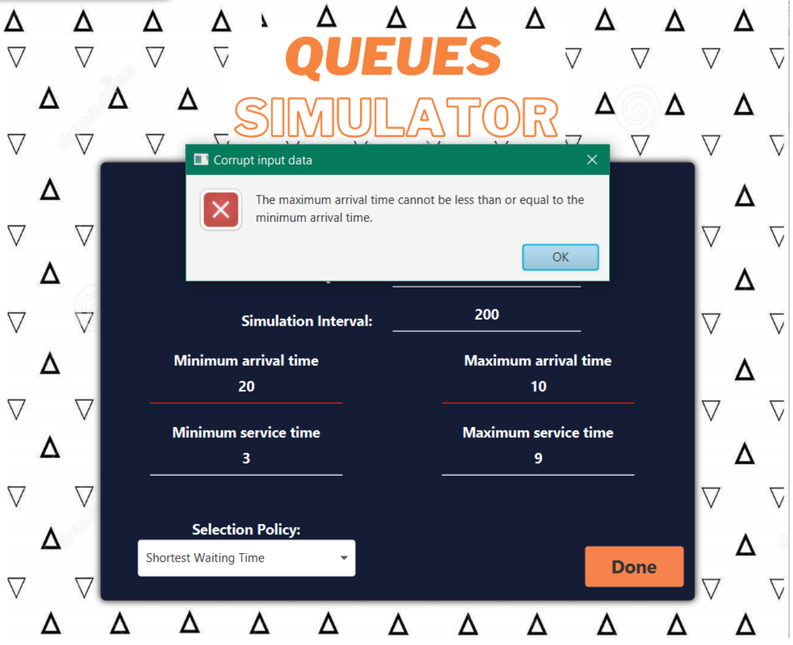
I have opted for a simple and intuitive design, with three windows that carry out the entire application:

- **Input Data** – the initial scene that allows the user to insert the desired input parameters;
- **Queues Simulation** – the following scene that carries out the real-time evolution of the simulation, according to the prior inserted parameters;
- **Simulation Statistics** – the final window, that displays the three computed statistics after the simulation is over.



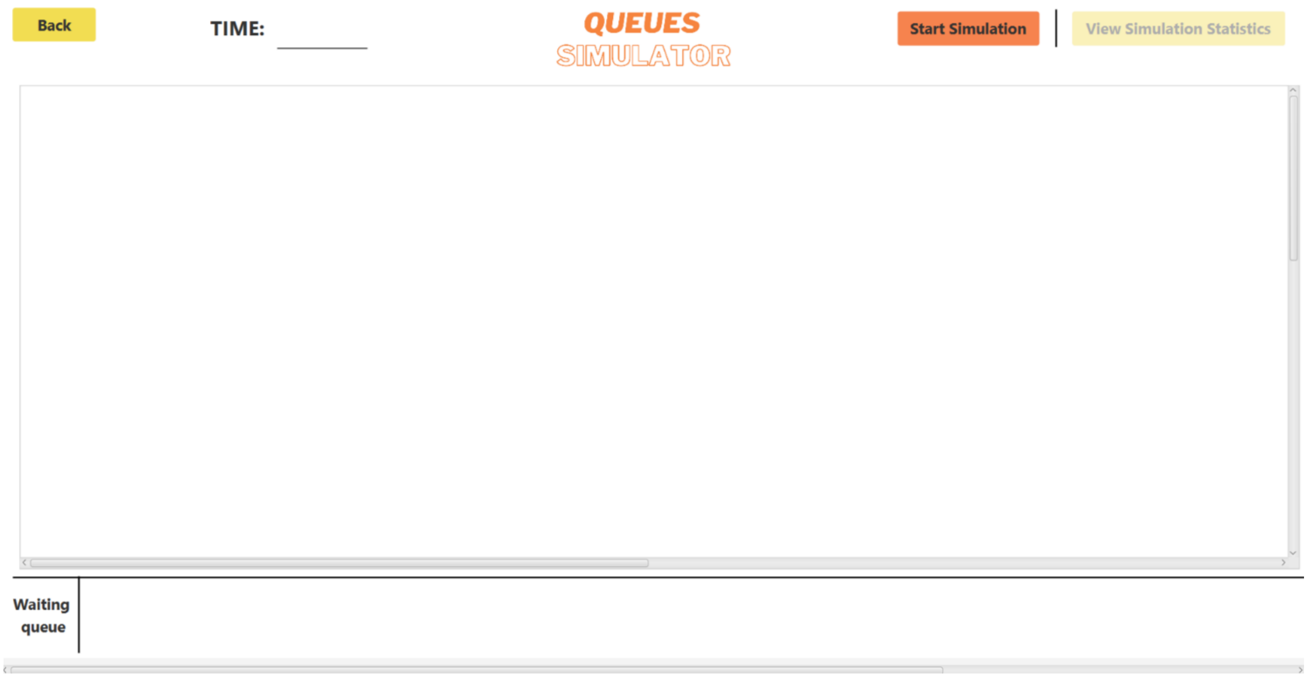
Please insert the following parameters:
Number of Clients: 1000
Number of Queues: 20
Simulation Interval: 200
Minimum arrival time: 10
Maximum arrival time: 100
Minimum service time: 3
Maximum service time: 9
Selection Policy: Shortest Waiting Time

the first scene of the application, after some input data has already been inserted

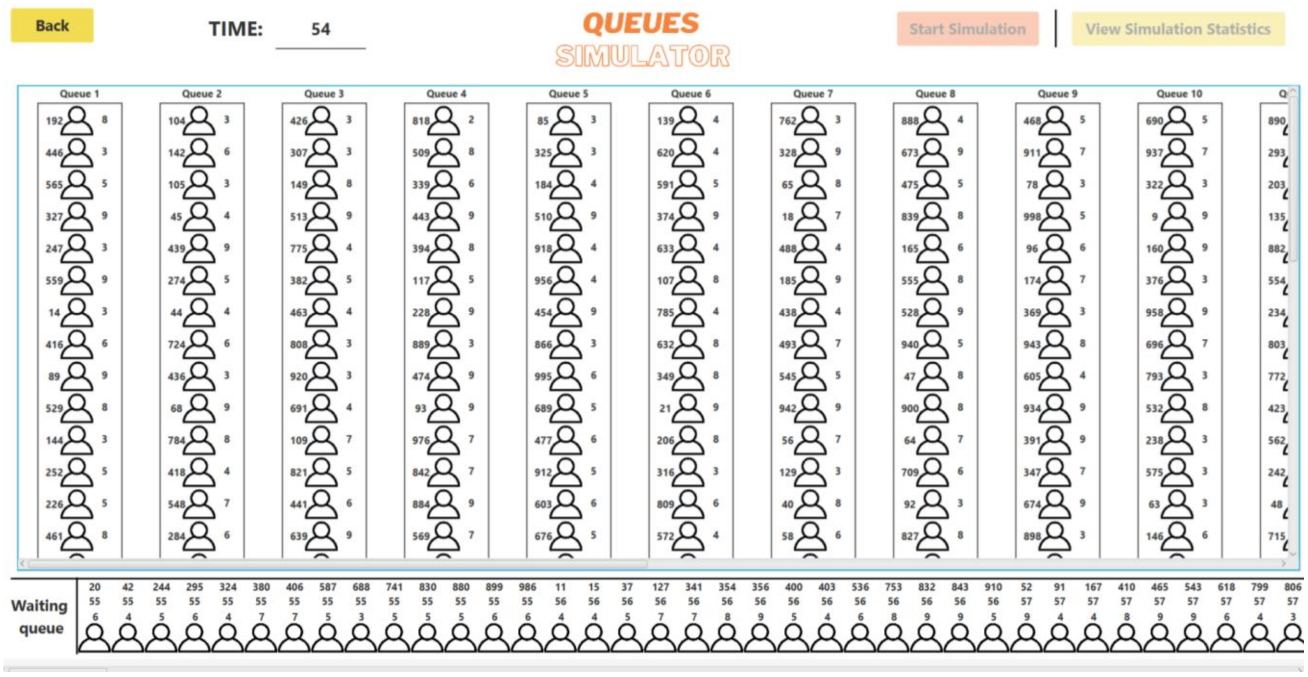


Corrupt input data
The maximum arrival time cannot be less than or equal to the minimum arrival time.
Simulation Interval: 200
Minimum arrival time: 20
Maximum arrival time: 10
Minimum service time: 3
Maximum service time: 9
Selection Policy: Shortest Waiting Time

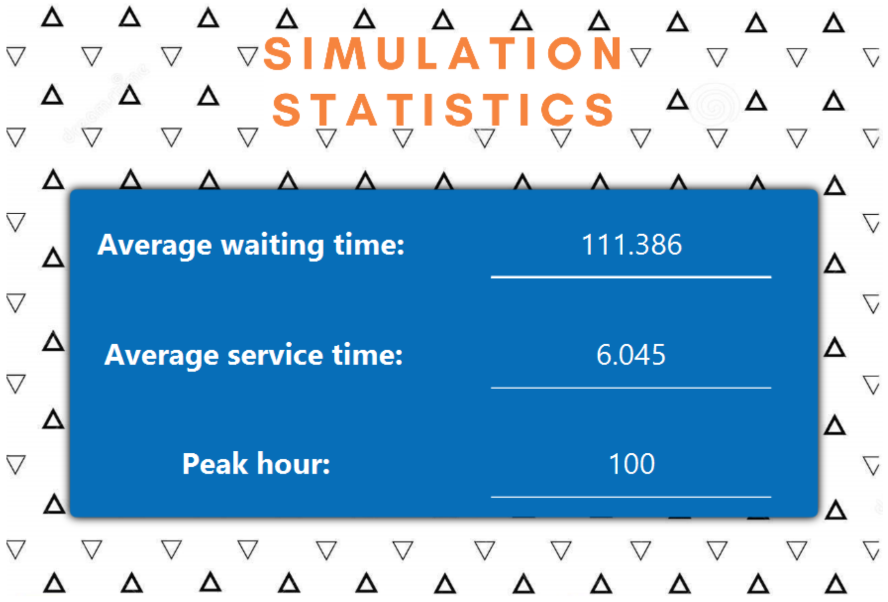
the first scene of the application, after an error was encountered due to invalid input data



The second scene of the application, before the Start Simulation has been pressed – an empty canvas. The Queues and waiting clients weren't generated yet. The View Simulation Statistics button is disabled.



The second scene of the application, after the Start Simulation button has been pressed. This is second 54 of the simulation, with clients already waiting in line and with other clients still waiting in the waiting queue. The Start Simulation button was disabled. The View Simulation Statistics button will be enabled only when the simulation is over.


SIMULATION STATISTICS

Average waiting time: 111.386

Average service time: 6.045

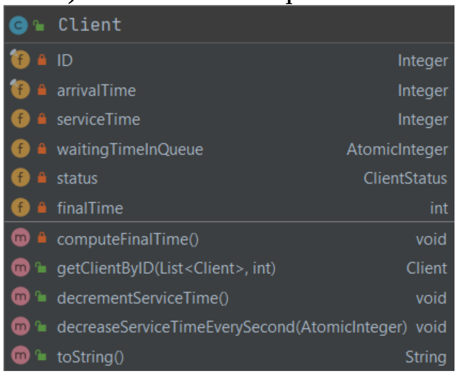
Peak hour: 100

the third and final scene of the application, displaying the three computed statistics of the prior simulation

4. Implementation

I will present each class sequentially, describing their most important fields and methods:

1) **Client** – the equivalent of a task;



```
Client
ID                                          Integer
arrivalTime                                 Integer
serviceTime                                 Integer
waitingTimeInQueue                          AtomicInteger
status                                      ClientStatus
finalTime                                   int
computeFinalTime()                          void
getClientByID(List<Client>, int)            Client
decrementServiceTime()                      void
decreaseServiceTimeEverySecond(AtomicInteger) void
toString()                                  String
```

Fields:

- **ID** – the ID of the client;
- **arrivalTime** – the second at which the client leaves the waiting queue to line up at one of the service queues;
- **serviceTime** – the amount of seconds it takes for a client to be served, when at the front of a service queue;
- **waitingTimeInQueue** – the amount of second a client waits in a service queue, before being served;
- **finalTime** – the amount of seconds it takes for a client to leave the service queues (*finalTime = arrivalTime + waitingTimeInQueue + serviceTime).*

Methods:

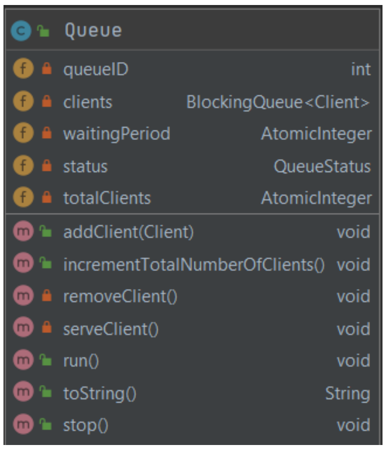
- **decreaseServiceTimeEverySecond()** – decreases the service time by one and then sleep the Thread for a second – this process is repetead for serviceTime x seconds;
- **computeFinalTime()** – computes the final time of service of a client;
- **getClientByID()** – finds a client from a list, based on its ID;
- **toString()** – takes a Client and returns its string equivalent (ID, arrivalTime, serviceTime).

2) **Queue** – the equivalent of a server;

- each queue is a separate thread, responsible for managing the clients that enqueue to it. A queue sleeps for serviceTime seconds whenever a client is at the front. Afterwards, it dequeues the first client and continues the cycle with the next one, until the queue is empty.

Fields:

- **queueID** – the unique ID that identifies a queue;



Queue

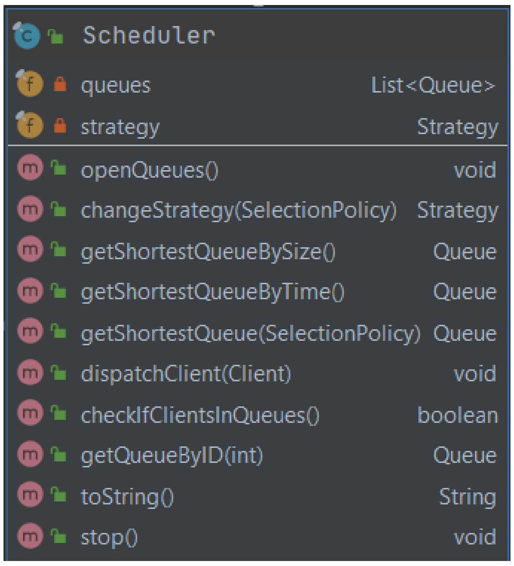
queueID	int
clients	BlockingQueue<Client>
waitingPeriod	AtomicInteger
status	QueueStatus
totalClients	AtomicInteger
addClient(Client)	void
incrementTotalNumberOfClients()	void
removeClient()	void
serveClient()	void
run()	void
toString()	String
stop()	void

- **clients** – a Blocking Queue of the clients that currently wait in this queue;
- **waitingPeriod** – the amount of seconds a clients spends waiting in this queue, based on the list of clients;
- **status** – the status of the queue (CLOSED or OPEN);
- **totalClients** – the total amount of clients that have been waiting in a queue, in the span of the entire simulation.

Methods:

- **addClient()** – adds a new client to the clients list;
- **incrementTotalNumberOfClients()** – increases the total number of clients by one, when a new client is added;
- **removeClient()** – removes a client from the clients list, after it has been served;
- **serveClient()** – serves a client for serviceTime seconds, decreasing the serviceTime by one each second.
- **run()** – overriden method – it states the actions a queue thread performs during its lifetime;
- **stop()** – stops the thread execution, by changing the queue statust from OPEN to CLOSED.

3) **Scheduler** – an immutable (final) class



Scheduler

queues	List<Queue>
strategy	Strategy
openQueues()	void
changeStrategy(SelectionPolicy)	Strategy
getShortestQueueBySize()	Queue
getShortestQueueByTime()	Queue
getShortestQueue(SelectionPolicy)	Queue
dispatchClient(Client)	void
checkIfClientsInQueues()	boolean
getQueueByID(int)	Queue
toString()	String
stop()	void

Fields:

- **queues** – a list of queues, generated at runtime;
- **strategy** – the strategy chosen by the user, according to which clients will be send to queues during the simulation;

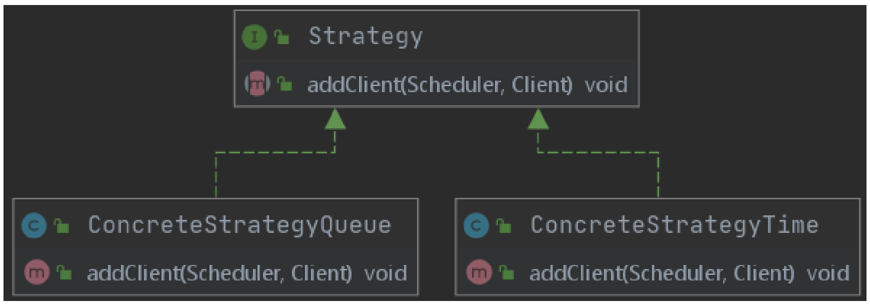
Methods:

- **openQueues()** – starts the threads of the queues from the queues list;
- **changeStrategy()** – initializes the strategy according to the SelectionPolicy enum (to be described later);
- **getShortestQueueBySize()** – gets the shortest queue in terms of number of clients;
- **getShortestQueueByTime()** – gets the shortest queue in terms of waiting time;
- **getShortestQueue()** – calls one of the two previously mentioned functions, according to the SelectionPolicy enum, and returns the shortest queue;
- **dispatchClient()** – calls the addClient method of the respective strategy;
- **checkIfClientsInQueues()** – if there are still clients in queues, it returns true. If there aren’t any clients in queues and the simulation time is done, the application will close;
- **getQueueByID()** – return a queue from the queues list, based on its QueueID;
- **stop()** – calls the stop method of each queue from the queues list.

4) **Strategy** – Interface (classes ConcreteStrategyTime, ConcreteStrategyQueue implement it)

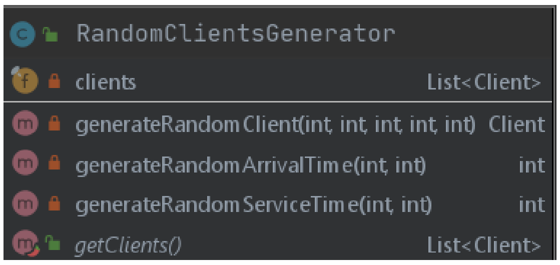
- implemented through the **Strategy Behavioural Design Pattern**

Methods:


Strategy
addClient(Scheduler, Client) void
ConcreteStrategyQueue
addClient(Scheduler, Client) void
ConcreteStrategyTime
addClient(Scheduler, Client) void

- **addClient()** – adds a client to the shortest available queue.
- ➢ ConcreteStrategyQueue establises the shortest queue based on the number of clients;
- ➢ ConcreteStrategyTime established the shortest queue based on the waiting time.

5) **RandomClientsGenerator**



```
RandomClientsGenerator
clients                                        List<Client>
generateRandomClient(int, int, int, int, int)  Client
generateRandomArrivalTime(int, int)            int
generateRandomServiceTime(int, int)            int
getClients()                                   List<Client>
```

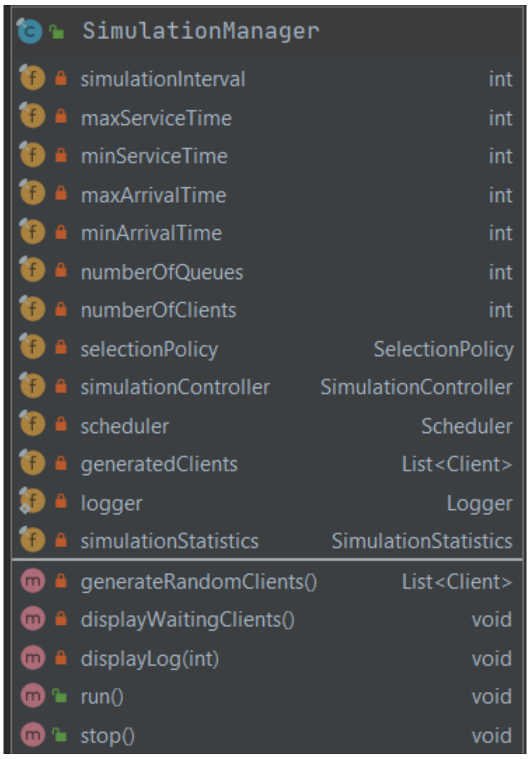
Fields:

- **clients** – a list of randomly generated objects of class Client.

Methods:

- **generateRandomArrivalTime()** – generates a random arrival time, bounded by $t_{arrival}^{MIN}$ and $t_{arrival}^{MAX}$
- **generateRandomServiceTime()** – generates a random service time, bounded by $t_{service}^{MIN}$ and $t_{service}^{MAX}$.
- **generateRandomClient()** – generates a Client with random parameters.

These methods are not implemented in the interface. They are overriden in each class that implements the interface, the compiler deciding at runtime which version to perform (Strategy pattern).



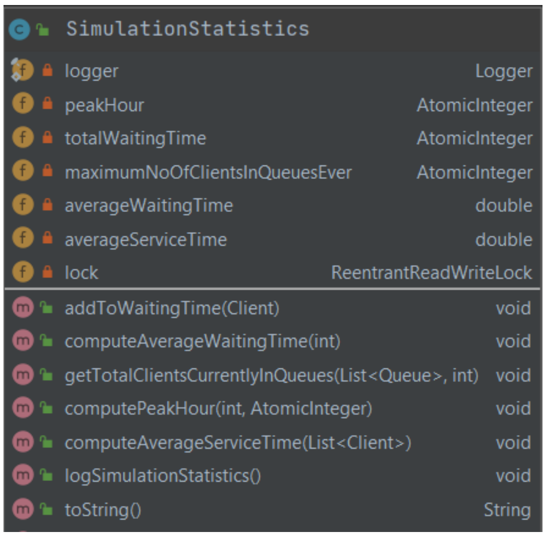
```
SimulationManager
simulationInterval        int
maxServiceTime            int
minServiceTime            int
maxArrivalTime            int
minArrivalTime            int
numberOfQueues            int
numberOfClients           int
selectionPolicy           SelectionPolicy
simulationController      SimulationController
scheduler                 Scheduler
generatedClients          List<Client>
logger                    Logger
simulationStatistics      SimulationStatistics
generateRandomClients()   List<Client>
displayWaitingClients()   void
displayLog(int)           void
run()                     void
stop()                    void
```

6)**SimulationManager** – an immutable class

- the SimulationManager is the main background Thread of the application. This thread starts Q queues threads and starts the simulation altogether, performing a cycle of action each second, until $t_{simulation}^{MAX}$ has been reached. It assigns clients from the waiting queue to the service queues, whenever a client's arrival time is equal to the current time of the simulation.

Methods:

- **generateRandomClients()** – generates a list of numberOfClients randomly created clients;
- **displayWaitingClients()** – displays the tuples of parameters that define the clients from the waiting queue;
- **displayLog()** – logs the status of the service queues and of the waiting queue, for the currentTime second;
- **run()** – overrides the thread method;
- **stop()** – calls the stop() method of Scheduler and interrupts the current Thread.



```
SimulationStatistics
logger                                          Logger
peakHour                                        AtomicInteger
totalWaitingTime                                AtomicInteger
maximumNoOfClientsInQueuesEver                  AtomicInteger
averageWaitingTime                              double
averageServiceTime                              double
lock                                            ReentrantReadWriteLock
addToWaitingTime(Client)                        void
computeAverageWaitingTime(int)                  void
getTotalClientsCurrentlyInQueues(List<Queue>, int) void
computePeakHour(int, AtomicInteger)             void
computeAverageServiceTime(List<Client>)         void
logSimulationStatistics()                       void
toString()                                      String
```

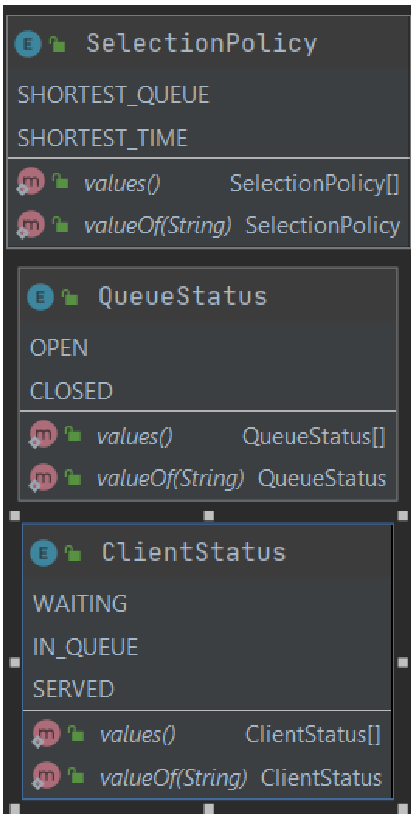
7) SimulationStatistics

Fields:

- **logger** – of class Logger, it logs the simulation statistics;
- **peakHour** – the hour when the most clients were lined up in the Q queues;
- **totalWaitingTime** – the total waiting time of the clients that were created during the simulation;
- **maximumNoOfClientsInQueuesEver** – the maximum number of clients that were found in the service queues at the same time;
- **averageWaitingTime**
- **averageServiceTime**

Methods:

- **addToWaitingTime()** – add the waiting time of a client to totalWaiting time;
- **computeAverageWaitingTime()** – totalWaitingTime / N;
- **getTotalClientsCurrentlyInQueues()** – counts the number of clients that are lined up in a service queue, during a second of the simulation;
- **computePeakHour()** – computes the peak hour of the simulation;
- **computeAverageServiceTime** – totalServiceTime / N;
- **logSimulationStatistics()** – writes the toString() representation of the SimulationStatistics class, with the help of a logger.



```
SelectionPolicy
SHORTEST_QUEUE
SHORTEST_TIME
values()              SelectionPolicy[]
valueOf(String)       SelectionPolicy

QueueStatus
OPEN
CLOSED
values()              QueueStatus[]
valueOf(String)       QueueStatus

ClientStatus
WAITING
IN_QUEUE
SERVED
values()              ClientStatus[]
valueOf(String)       ClientStatus
```

8) **SelectionPolicy** – Enumerator

Values:

- SHORTEST_QUEUE – based on the number of clients;
- SHORTEST_TIME – based on the waiting time of a queue.

9) **QueueStatus** – Enumerator

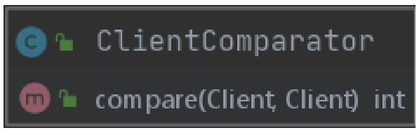
Values:

- OPEN – a queue is ready to receive clients and to continue to operate, serving them;
- CLOSED – a queue closes, being unavailable for clients.

10) **ClientStatus** – Enumerator

Values:

- WAITING – a client's state when it is found in the waiting queue;
- IN_QUEUE – a client's state when it is lined up in a service queue, still waiting to be served;
- SERVED – a client's state after its service has been received.


ClientComparator
compare(Client, Client) int

11) **ClientComparator** – implements Comparator Interface

- compares two clients, first by their arrival time, and then by their ID.

12) BadInputException – extends Exception

This class is a custom exception, that gets thrown whenever a bad input was detected (either by the previously explained regular expressions, or from manually added data to constructors). The exception is handled in the PolynomialCalculatorController, where a custom alert pops up in a dialog window if the user inserts invalid characters.

The GUI

For the GUI I have used three .fxml files, containing the view (the aspect) of the Graphical User Interface. For each of these .fxml files, I have a Controller class, that combines the model and the view of the project.

For containers, I have used AnchorPanes, ScrollPanes and GridBoxes. I have found that it is easier to arrange controls on the canvas with GridBoxes and they get better resized if the dimensions of the windows are manually changed by the user.

The service queues are implemented with the help of VBoxes, that automatically resize when new elements are added into them. The waiting queue was modeled using a HBox, for the same purposes. The ScrollPane allows the user to see all the generated clients and waiting queues during a simulation.

As for controls, I have Text Fields, Labels, ComboBoxes and Buttons. The buttons are responsible for every action that can be performed by the user. When a button is pressed, its associated function starts running and the code is executed. I have also used the JFoenix take on some of the original JavaFx control, because they are more stylish.

5. Results

The results of the following tests can be found in the text files Test1Logs.txt, Test2Logs.txt and Test3Logs.txt.

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]
selectionPolicy = shortest waiting time

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]
selectionPolicy = shortest waiting time

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]
selectionPolicy = shortest waiting time

6. Conclusions

After completing the Queues Simulator, I have managed to get an understanding of how Java Concurrency works and of how Java Threads must be handled to assure the best functionality of the project. I have worked for the first time with a Logger, and I have found out that its detailed messages and its diversity is to be preferred over simple System output prints. I have gained a deeper knowledge of some Java features, such as working with streams instead of doing computations manually.

I have enjoyed seeing how Java Concurrency works in the context of a real-life example.

Further developments:

- displaying more statistics of a simulation;
- allowing the user to pause the simulation and then start it from the second it was stopped on;
- using the Executor framework for a better Thread handling;
- using the Task class for better communication between the JavaFX Thread and the background ones.

7. Bibliography

Fundamental Programming Techiniques – *Lecture Slides*: Dr. Eng. Prof. Salomie Ioan

Fundamental Programming Techiniques – *Assignment Two Support Presentation*: Dr. Eng. Prof. Salomie Ioan

Fundamental Programming Techniques – *PT2021_Assignment2*: Dr. Eng. Prof. Salomie Ioan

Wikipedia – *Queue (abstract data type)*: https://en.wikipedia.org/wiki/Queue_(abstract_data_type)

Wikipedia – *Java concurrency*: https://en.wikipedia.org/wiki/Java_concurrency

Wikipedia – *Thread (computing):* https://en.wikipedia.org/wiki/Thread_(computing)

Oracle – *Platform:*
https://docs.oracle.com/javafx/2/api/javafx/application/Platform.html#runLater%28java.lang.Runnable%29

Loggly – *Logging Basics*: https://www.loggly.com/ultimate-guide/java-logging-basics/

Logging Service – *Frequently Asked Questions*: https://logging.apache.org/log4j/log4j-2.0/faq.html#config_sep_appender_level