

MINISTRY OF EDUCATION AND SCIENTIFIC RESEARCH



TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

Programming Techniques
Assignment 2

Queue Simulation

Teacher: prof. Ioan Salomie
Teacher Assistant: Cristina Bianca Pop
Student: Bucur Alexandra
Group: 30424

2020-2021

Contents

1. Assignment objective.....	3
2. Problem analysis, modeling, scenarios, use cases.....	4
Analyzing the problem.....	4
Modelling.....	4
Scenarios and Use cases.....	4
3. Design	6
Design decisions	6
Relationships, packages	6
UML diagrams	8
Data Structures.....	8
Class Design.....	9
Algorithms	10
User Interfaces	11
4. Implementation	14
Client class	14
Server Class	14
Scheduler Class.....	14
TimeBounds Class	15
SimulationManager Class	15
Controller Class	15
Strategy, ConcreteStrategyTime, ConcreteStrategyQueue Class	15
App Class.....	16
5. Results.....	17
6. Conclusions.....	19
Further implementation.....	19
7. Bibliography	20

1. Assignment objective

Main objective:

Design and implement a simulation application aiming to analyze queuing based systems for determining and minimizing clients' waiting time.

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 is interested in minimizing the time amount their "clients" are waiting in queues before they are served. One way to minimize the waiting time is to add more servers; more queues in the system.

The application should simulate (by defining a simulation time *tsimulation*) a series of clients arriving for service, entering some queues, waiting, being served and finally leaving the queues. All clients are generated when the simulation is started, and are characterized by three parameters: ID (a number between 1 and N), *tarrival* (simulation time when they are ready to go to the queue; i.e. time when the client finished shopping) and *tservice* (time interval or duration needed to serve the client; i.e. waiting time when the client is in front of the queue).

The application tracks the total time spent by every client in the queues and computes the average waiting time, but also a peak hour and the average service time.

The following are considered as input data from the user.

- Number of clients (N);
- Number of queues (Q);
- Simulation interval (*tsimulationMAX*);
- Minimum and maximum arrival time ($tarrivalMIN \leq tarrival \leq tarrivalMAX$);
- Minimum and maximum service time ($tserviceMIN \leq tservice \leq tserviceMAX$);

Sub-objectives:

- Analyze the problem and identify requirements
- Design the queue simulator
- Implement the queue simulator
- Test the queue simulator

1. Analyze the problem and identify requirements

This part will analyze the requirements and decide the modelling, scenarios and use cases. As functional requirements the user, which is also the primary actor of the project, should be able to insert all the necessary data for performing the queue simulation and then see the resulted output in a period of time. It will be presented in detail in [part 2](#) of the documentation.

2. Design the queue simulator

This part will analyze the design of the queue simulator. It will decide on a certain design pattern, decide on package configurations and the design of the classes.

It will also present the OOP design of the application, the UML class and package diagrams, the data structures used, the defined interfaces and the algorithms used (if applicable) will be presented.

This part will be presented in detail in [part 3](#).

3. Implement the queue simulator

This part will present in detail the implementation of the project. Each class will be described with important fields and methods. The implementation of the user interface will be described. This will be explained later, in [part 4](#).

4. Test the queue simulator

This part will present the result of the testing of the application. It will be presented later, in [part 5](#).

2. Problem analysis, modeling, scenarios, use cases

Analyzing the problem

General overview

This application should simulate customers waiting to receive a service (e.g. supermarket, bank, etc.) just like in the real world, they have to wait in queues, each queue processing clients simultaneously.

The idea is to analyze how clients can be served in a certain simulation interval, by inserting some parameters in an appropriate interface, designed especially for this scope.

Modelling

The user will be able to insert some parameters and then see the simulation of some queues in real-time simulation. All customers are randomly generated, depending on the input values, such as number of clients, simulation interval and time bound for arrival and service time.

The user can set different parameters:

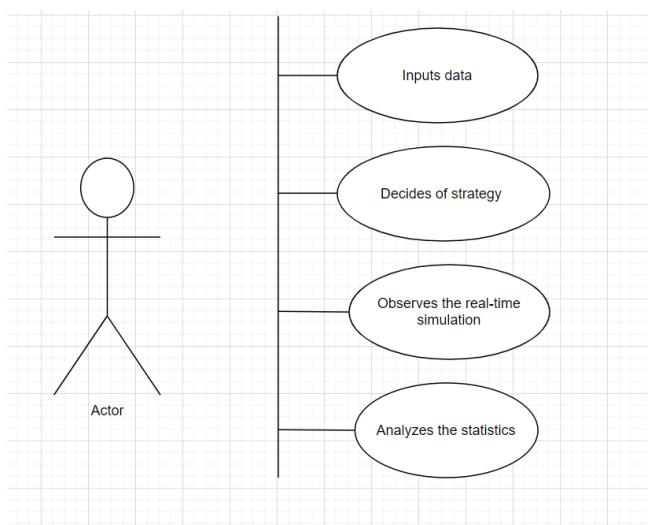
- The maximum number of queues available
- The maximum number of customers
- A simulation interval (in seconds)
- Minimum and maximum service time: the number of seconds needed for a client to be processed.
- Minimum and maximum arrival time: seconds between which a customer can enter the queues.
- The user can select the strategy on which the queues will be processed (shortest waiting period or shortest queue).

Based on the input introduced by the user, the result of the chosen operation will be displayed in the interface, in the designated place

- The real-life simulation of the queues (in seconds).
- The average waiting time for total.
- The “peak hour” => when the most clients were served.
- The average service time (in seconds)

If the data that is introduced is not correct, it will display a proper error message.

Scenarios and Use cases



The use case represents the basic form of functioning of the program. At first the user(actor) will input the data, after this, the user will choose between the available mode of operations and will see the result of the queue simulation. If one of the data is not correct, the application will signal this.

Shortest queue strategy

- **Use Case:** view simulation of a certain number of queues(shortest queue strategy)

Primary Actor: user

Main Success Scenario:

1. The user inserts all the necessary data, one at the time in the graphical user interface.
3. The user selects the shortest queue strategy
4. The queue simulator performs the real-time development of the queues and displays the result, also the statistics at the end

Alternative Sequence: Incorrect data

- The user inserts incorrect numbers (e.g. negative numbers/ letters instead of numbers)
- An error message will be displayed for the data that was incorrect
- The scenario returns to the current field that has to be inputted(in step 1)

Shortest time period strategy

- **Use Case:** view simulation of a certain number of queues(shortest time period strategy)

Primary Actor: user

Main Success Scenario:

1. The user inserts all the necessary data, one at the time in the graphical user interface.
3. The user selects the shortest time strategy
4. The queue simulator performs the real-time development of the queues and displays the result, also the statistics at the end

Alternative Sequence: Incorrect data

- The user inserts incorrect numbers (e.g. negative numbers/ letters instead of numbers)
- An error message will be displayed for the data that was incorrect
- The scenario returns to the current field that has to be inputted(in step 1)

3. Design

Design decisions

- This project has a MVC design pattern., which divides the application into three areas: processing, output and input.

Context

- Many software systems deal with finding data from a repository and displaying the data to the users through a graphical user interface (GUI)
- Disadvantages:
 - The GUI changes more often than the business logic implementation -> if they are implemented in the same class then each time the GUI changes the business logic is changed
 - The business logic cannot be reused
 - The code is complex and difficult to maintain

Model components

- The central component of the pattern. It is the application's dynamic data structure, independent of the user interface. It directly manages the data, logic and rules of the application.
- main idea: encapsulates core data and functionality.

View components

- Any representation of information such as a chart, diagram or table. Multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants.
- main idea: display information to the user - obtains the data it displays from the model

Controller

- Each view has an associated controller component. Controllers receive input, usually as events that denote mouse movement, activation of mouse buttons or keyboard input. Events are translated to service requests, which are sent either to the model or to the view.
- main idea: accepts input and converts it to commands for the model or view.

Relationships, packages

Java packages help in organizing multiple modules and group together related classes and interfaces.

In object-oriented programming development, model-view-controller (MVC) is the name of a methodology or design pattern for successfully and efficiently relating the user interface to underlying data models. The MVC pattern is widely used in program development with programming languages such as Java, Smalltalk, C, and C++.

The application, which is implementing the Model-View-Controller architectural design pattern, will have 4 packages: Controller Package, Model Package, View Package and Main Package.

The Main package:– contains a single class, named App, which contains the main() method and will be the one that calls and produces the page for the user interface.

The Controller package: -Will have class, MainController and SimulationManager

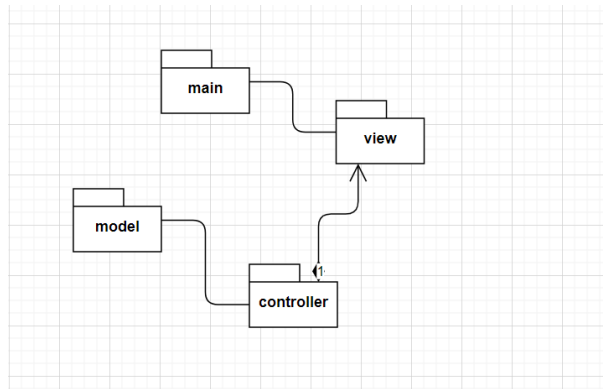
-This package will deal with linking the user interface (available in View package) and the implementation in the backend of the application (available in Model package) and dealing with the main thread that will take care of the servers.

- it interconnects the model and the view

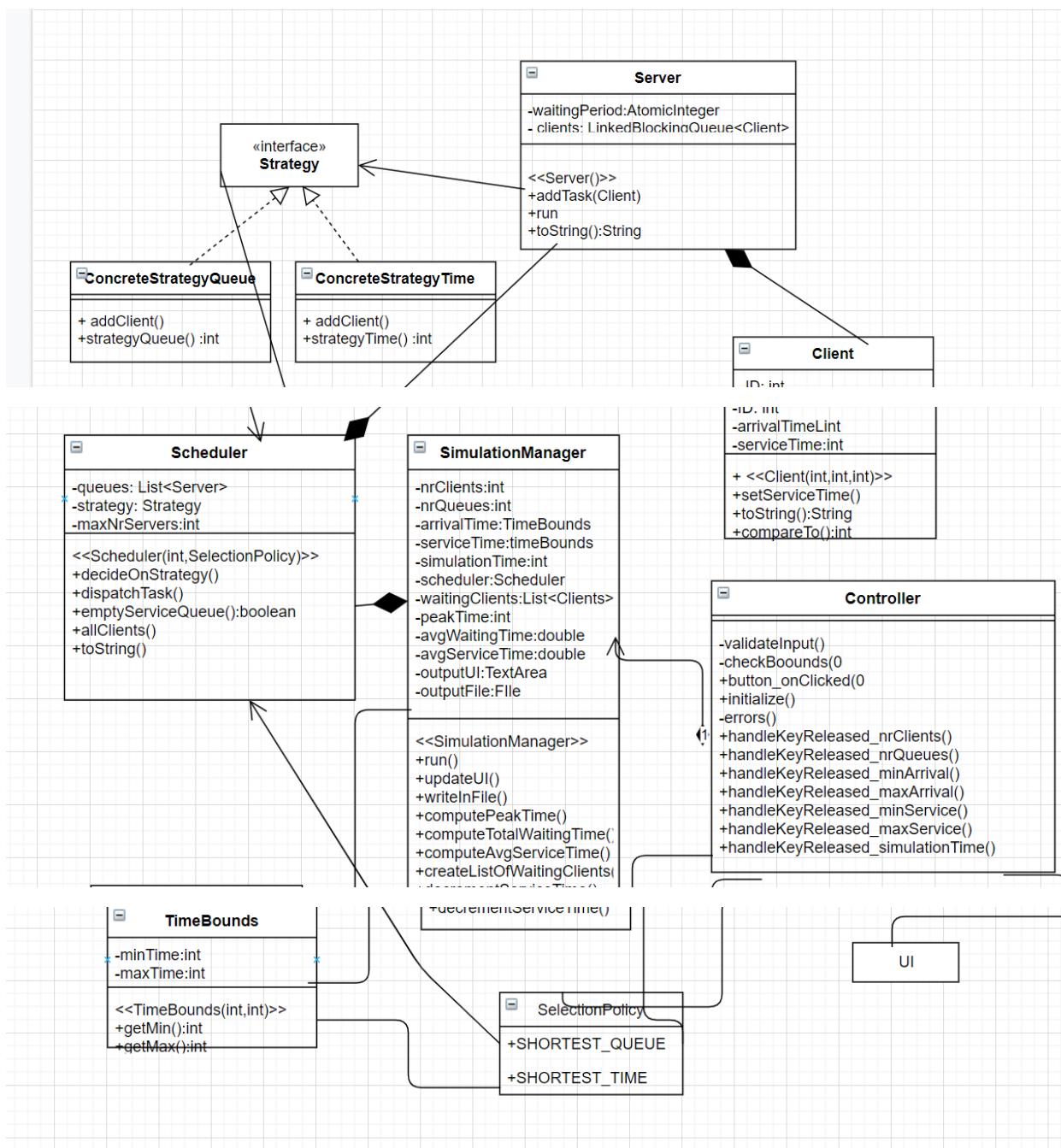
The View package: – Contains a single class which represents the appearance of the user interface (GUI)

The Model package:

- Contains classes that deal with the model of the problem
- Define the client class and the servers(queues).
- decides on a strategy of the clients' way of entering a queue.



UML diagrams



Data Structures

The data structures that have been used in this application are either primitive data types, such as integers or doubles.

The application also uses a newly created class, called TimeBounds that represents a period of time(an interval), with a minValue and a maxValue. I use this for the arrival and service bounds. I also have a class Client that I use as an incorporated type for the queues(servers) => list of clients.

I also use thread safe types of data, such as AtomicInteger and LinkedBlockingQueue in order to be sure that all the data is synchronized.

For the list implementation, I chose to use an LinkedBlockingQueue type, instead of a simple array, because they are more efficient from the point of view of memory management and provide a faster access to their content. This type of data is also a thread safe type, so we won't worry about the data being incorrect or being fetched a little but too late. The code is also easier to understand and cleaner. Moreover, the size of an LinkedBlockingQueue is not fixed so we do not have to about incrementing the length of the array or getting an "Overflow" exception.

Class Design

The whole idea of splitting your program into classes is based on a general rule named divide and conquer. The application divides a problem into smaller problems and then those small classes solve the simple and well-known problems.

Because I followed the MVC architecture, my program consists of 4 parts: Main package, Controller Package, Model Package, View Package.

1) The Model package – contains the logic of the application

Client Class

Fields:

ID: int

arrivalTime:int

serviceTime:int

Constructors:

– public Client (int ID, int arrivalTime, int serviceTime) : the constructor that initializes the client with the transmitted fields

Methods:

-public void setServiceTime(): decrements the service time for a customer(client), during processing time

-public void toString() : displays one client on the screen

-public void compareTo(Object o): compares 2 clients

Server Class

Fields:

waitingTime: AtomicInteger

clients : LinkedBlockingQueue<Client>

Constructor:

-public Server() : default constructor

Methods:

-public BlockingQueue<Client> getClient() : returns the list of clients

- public void addTask() : adds a new client in the current queue and increases the waiting time
- public void toString() : displays one queue
- public void run(): simulates the behaviour of one client having in mind his service time and removes it when it is processed

Scheduler class

```
private List<Server> queues;
private int maxNrServers;
private int maxTasksPerServer;
private Strategy strategy;
```

Constructors:

public Scheduler(int maxNrServers, int maxTasksPerServer, SelectionPolicy policy) => decided on the strategy and initializes the queues

Methods:

```
public void decideOnStrategy(): decides on the strategy
public void dispatchTask(): adds a new client in one of the servers
public List<Server> getQueues(): simple getter
public boolean emptyServiceQueues(): verifies if all queues are empty
public String toString() : displays all servers
public int allClients()
```

SimulationPolicy Enum

-this enum is used for returning the actual strategy

TimeBounds Class

-class used to simulate an interval for the random generation of the clients

ConcreteStrategyQueue & ConcreteStrategyTime Classes

-classes that implement the interface Strategy, decides on which queue will the next client go to

Interface: Strategy

-has only one method, addTask method, that adds a client into a designated queue(method implemented in other classes)

screen Class

-JavaFX code, displays all buttons, labels, text areas on the screen.

3) The Control package– this contains the linking between the model and the view

Controller Class

-deals with events for buttons and labels, but also validating data.

Most important method: validateInput()

-has a subclass **ProcessService** that helps displaying labels for a certain amount of time

SimulationManager Class

-deals with generating n random clients with all the data

-takes care of the main thread, that simulated the real time output of the queues simulations

4) The Main package

App class

–App class which runs the start() method in order to “turn on” the application.

Algorithms

The Server class contains a run method:

This method deals with a thread and decides on what the current queue will contain => a client that has no more service time will be removed from the queue.

```
public void run() {
    int sleepPeriod = 1;
    if(this.clients.peek() != null) {
        sleepPeriod = this.clients.peek().getServiceTime();
    }
    while (true){
        try{
            this.waitingPeriod.decrementAndGet();
            Thread.sleep(1000 * sleepPeriod);
        } catch (InterruptedException e){
            System.out.println("Something woke me");
        }
        if (clients.peek() != null && clients.peek().getServiceTime() == 0) {
            clients.remove();
        }
    }
}
```

Other important methods are the ones in the SimulationManager, the run method that inserts a new client when the current time is equal with the arrival time of the client.

User Interfaces

At the beginning the interface looks like this. The user will now be able to insert all the data necessarily for the simulation. Only the first textbox is enabled in order to make the user insert a valid input.

Queue simulator

Number of clients: 10
 Number of queues: 3
 Simulation time: 20

Minimum arrival time: 5
 Maximum arrival time: 16
 Minimum service time: 1
 Maximum service time: 7

Shortest queue
 Shortest time

Start simulation

After all input has been introduced in a correct manner, a choiceBox will appear in order to decide on the policy of the simulation => client will chose the shortest queue or the shortest waiting time.

During the simulation, the user will be able to see the current moment and the appearance of the queues.

Number of clients: 4
 Number of queues: 3
 Simulation time: 12

Minimum arrival time: 1
 Maximum arrival time: 3
 Minimum service time: 1
 Maximum service time: 5

Time= 1
 Waiting clients: [(1,2,4), (2,3,5), (3,3,4), (4,3,4)]
 Queue 1 :closed
 Queue 2 :closed
 Queue 3 :closed

Time= 2
 Waiting clients: [(2,3,5), (3,3,4), (4,3,4)]
 Queue 1 :{1,2,4}
 Queue 2 :closed
 Queue 3 :closed

Time= 3

Shortest queue

Start simulation

Number of clients: assf
 Number of queues:
 Simulation time:
 Please insert a number

If the input data is not correct, it will show an attention message. It will show a message if the number is negative, also.

Average service time: 5

Peak time: 3

Average waiting time: 5

At the end the peak value, avg service time and avg waiting time will be displayed.

Details about JAVAFX can be found at the links at [section 7.1](#)

4. Implementation

Client class

Client	
setServiceTime()	void
toString()	String
compareTo(Client)	int
arrivalTime	int
ID	int
serviceTime	int

The Client class represent the smaller class in the project and deals with the information of one client. There are only 3 methods here, but the important ones are setServiceTime and compareTo. The first decreases by 1 the service time of a client, during the real-life simulation and the other method will compare 2 client based on their arrival time.

Server Class

Server	
addTask(Client)	void
run()	void
toString()	String
waitingPeriod	AtomicInteger
clients	BlockingQueue<Client>

This class represents a queue of clients and the corresponding waiting time for each client. Method addTask will increase the size of the queue and add a new client, and the waitingPeriod of the queue will be increased with the service time of the newly added client.

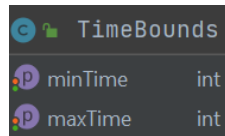
The run method will check when the service time for a client is 0 and remove it from the queue, but also put the current thread to sleep for a period equal to the one of the service time of the first client. More about threads and concurrency in [section 7.2](#)

Scheduler Class

Scheduler	
decideOnStrategy(SelectionPolicy)	void
dispatchTask(Client)	void
emptyServiceQueues()	boolean
toString()	String
allClients()	int
queues	List<Server>
strategy	Strategy

This class decides on how the next client will be introduced, or more precisely where (in which queue/server) it will be introduced. It holds a list with all the servers and the method emptyServiceQueue will check if all the queues are empty or not, returning a corresponding Boolean. Based on the SelectionPolicy, the Scheduler will decide on the appropriate strategy, which will be decided in the first method that can be seen on the left.

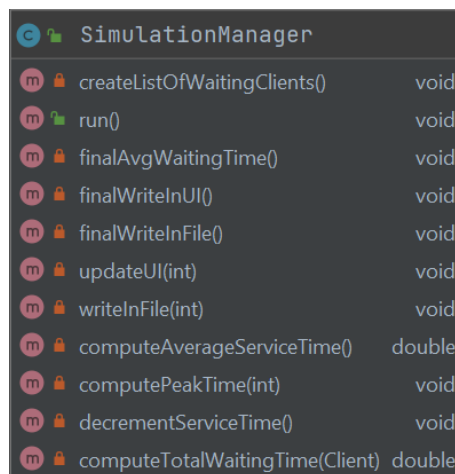
TimeBounds Class



TimeBounds	
minTime	int
maxTime	int

This class is a relatively simple class, with only 2 fields and setters or getters, but also the constructor that has both fields.

SimulationManager Class

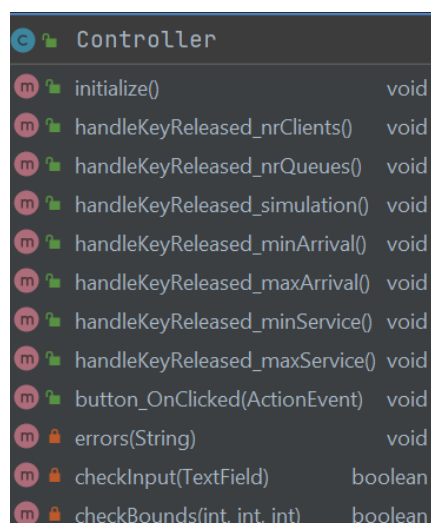


SimulationManager	
createListOfWaitingClients()	void
run()	void
finalAvgWaitingTime()	void
finalWriteInUI()	void
finalWriteInFile()	void
updateUI(int)	void
writeInFile(int)	void
computeAverageServiceTime()	double
computePeakTime(int)	void
decrementServiceTime()	void
computeTotalWaitingTime(Client)	double

This class will generate n random clients based on the input introduced by the user. It will also command the output on the interface. The main method(run) will programmatically introduce new clients in the designated queue when the arriving time is the one as the currentTime. More simply, the run method will deal with the real time simulation and display at the end of the simulation the peak hour, the average time of service and the average time of waiting. The rest of the methods deal with displaying the data in a file or on the designated user interface.

There are a few more methods that calculate the peak hour and avg time.

Controller Class



Controller	
initialize()	void
handleKeyReleased_nrClients()	void
handleKeyReleased_nrQueues()	void
handleKeyReleased_simulation()	void
handleKeyReleased_minArrival()	void
handleKeyReleased_maxArrival()	void
handleKeyReleased_minService()	void
handleKeyReleased_maxService()	void
button_OnClicked(ActionEvent)	void
errors(String)	void
checkInput(TextField)	boolean
checkBounds(int, int, int)	boolean

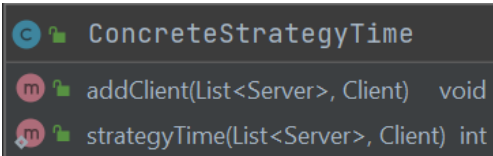
This class deals with all the changes that will appear on the graphical user interface. The public methods are events for the buttons, labels, text areas.

There is one more method that validates the input(all should be numbers for example, positive etc.)

The checkBound is also a validation method in order to see that the interval inserted by the user is a valid one.

Strategy, ConcreteStrategyTime, ConcreteStrategyQueue Class

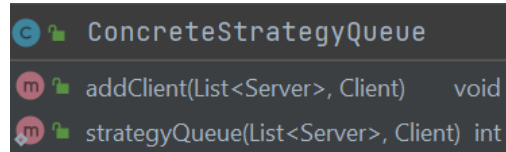
Strategy class is actually an interface that is implemented by ConcreteStrategyTime and ConcreteStrategyQueue classes.



```
ConcreteStrategyTime
addClient(List<Server>, Client) void
strategyTime(List<Server>, Client) int
```

The 2 methods decide on which queue should a new client be inserted, here based on the shortest period of waiting time.

The second method returns the index of the queue that will be increased.

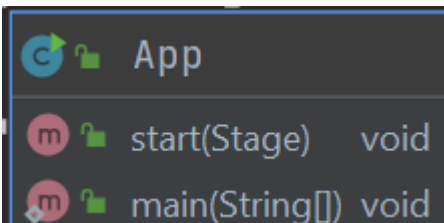


```
ConcreteStrategyQueue
addClient(List<Server>, Client) void
strategyQueue(List<Server>, Client) int
```

The 2 methods decide on which queue should a new client be inserted, here based on the shortest length of the queue.

The second method returns the index of the queue that will be increased.

App Class



```
App
start(Stage) void
main(String[]) void
```

This class contains the main() of the application. At runtime the application will start from this point.

5. Results

Test 1

N = 4

Q = 2

tsimulationMAX = 60 seconds

[*tarrivalMIN*, *tarrivalMAX*] = [2, 30]

[*tserviceMIN*, *tserviceMAX*] = [2, 4]

<p>Time= 0 Waiting clients: [(1,8,3), (2,14,3), (3,16,4), (4,28,4)] Queue 1 :closed Queue 2 :closed</p> <p>Time= 1 Waiting clients: [(1,8,3), (2,14,3), (3,16,4), (4,28,4)] Queue 1 :closed Queue 2 :closed</p> <p>Time= 2 Waiting clients: [(1,8,3), (2,14,3), (3,16,4), (4,28,4)] Queue 1 :closed Queue 2 :closed</p> <p>Time= 3 Waiting clients: [(1,8,3), (2,14,3), (3,16,4), (4,28,4)] Queue 1 :closed Queue 2 :closed</p> <p>Time= 4 Waiting clients: [(1,8,3), (2,14,3), (3,16,4), (4,28,4)] Queue 1 :closed Queue 2 :closed</p> <p>Time= 5 Waiting clients: [(1,8,3), (2,14,3), (3,16,4), (4,28,4)] Queue 1 :closed Queue 2 :closed</p> <p>Time= 6 Waiting clients: [(1,8,3), (2,14,3), (3,16,4), (4,28,4)] Queue 1 :closed Queue 2 :closed</p> <p>Time= 7 Waiting clients: [(1,8,3), (2,14,3), (3,16,4), (4,28,4)] Queue 1 :closed Queue 2 :closed</p>	<p>Time= 11 Waiting clients: [(2,14,3), (3,16,4), (4,28,4)] Queue 1 :closed Queue 2 :closed</p> <p>Time= 12 Waiting clients: [(2,14,3), (3,16,4), (4,28,4)] Queue 1 :closed Queue 2 :closed</p> <p>Time= 13 Waiting clients: [(2,14,3), (3,16,4), (4,28,4)] Queue 1 :closed Queue 2 :closed</p> <p>Time= 14 Waiting clients: [(3,16,4), (4,28,4)] Queue 1 : (2,14,3) Queue 2 :closed</p> <p>Time= 15 Waiting clients: [(3,16,4), (4,28,4)] Queue 1 : (2,14,2) Queue 2 :closed</p> <p>Time= 16 Waiting clients: [(4,28,4)] Queue 1 : (2,14,1) Queue 2 : (3,16,4)</p> <p>Time= 17 Waiting clients: [(4,28,4)] Queue 1 :closed Queue 2 : (3,16,3)</p> <p>Time= 18 Waiting clients: [(4,28,4)] Queue 1 :closed Queue 2 : (3,16,2)</p> <p>Time= 19 Waiting clients: [(4,28,4)] Queue 1 :closed Queue 2 : (3,16,1)</p>	<p>Time= 23 Waiting clients: [(4,28,4)] Queue 1 :closed Queue 2 :closed</p> <p>Time= 24 Waiting clients: [(4,28,4)] Queue 1 :closed Queue 2 :closed</p> <p>Time= 25 Waiting clients: [(4,28,4)] Queue 1 :closed Queue 2 :closed</p> <p>Time= 26 Waiting clients: [(4,28,4)] Queue 1 :closed Queue 2 :closed</p> <p>Time= 27 Waiting clients: [(4,28,4)] Queue 1 :closed Queue 2 :closed</p> <p>Time= 28 Waiting clients: [] Queue 1 : (4,28,4) Queue 2 :closed</p> <p>Time= 29 Waiting clients: [] Queue 1 : (4,28,3) Queue 2 :closed</p> <p>Time= 30 Waiting clients: [] Queue 1 : (4,28,2) Queue 2 :closed</p> <p>Time= 31 Waiting clients: [] Queue 1 : (4,28,1) Queue 2 :closed</p> <p>Time= 32 Waiting clients: []</p>
---	--	---

<p>Time= 8 Waiting clients: [(2,14,3), (3,16,4), (4,28,4)] Queue 1 : (1,8,3) Queue 2 :closed</p> <p>Time= 9 Waiting clients: [(2,14,3), (3,16,4), (4,28,4)] Queue 1 : (1,8,2) Queue 2 :closed</p> <p>Time= 10 Waiting clients: [(2,14,3), (3,16,4), (4,28,4)] Queue 1 : (1,8,1) Queue 2 :closed</p>	<p>Time= 20 Waiting clients: [(4,28,4)] Queue 1 :closed Queue 2 :closed</p> <p>Time= 21 Waiting clients: [(4,28,4)] Queue 1 :closed Queue 2 :closed</p> <p>Time= 22 Waiting clients: [(4,28,4)] Queue 1 :closed Queue 2 :closed</p>	<p>Queue 1 :closed Queue 2 :closed</p> <p>Average service time: 3.5 Peak time: 16 Average waiting time: 0</p>
---	---	---

Test 2

N = 50

Q = 5

tsimulationMAX = 60 seconds

[*tarrivalMIN*, *tarrivalMAX*] = [2, 40]

[*tserviceMIN*, *tserviceMAX*] = [1, 7]

Test 3

N = 1000

Q = 20

tsimulationMAX = 200 seconds

[*tarrivalMIN*, *tarrivalMAX*] = [10, 100]

[*tserviceMIN*, *tserviceMAX*] = [3, 9]

The tests results are available in the .txt files.

6. Conclusions

This project was a good exercise in remembering the OOP concepts learned in the first semester, but also learning some new concepts regarding design patterns, and the most important one, working with Threads. Even though I have not properly worked with threads before, now, due to this project I now understand how they work and how to be used.

This assignment helped me to improve my knowledge about queues processing and how they are implemented on a system, I learned about Threads and about synchronization.

Further implementation

- Make the interface more appealing.
- Be able to stop the simulator at a certain moment and inspect the queues details, size, current waiting time.
- Focus on displaying the customers(clients) in a more appealing manner(with icons of a person).

7. Bibliography

1. JAVAFX:
 - 1.1 <https://docs.oracle.com/javase/8/javafx/api/javafx/scene/control/TextArea.html>
 - 1.2 <https://www.tutorialspoint.com/how-to-create-a-text-area-in-javafx>
2. Concurrency and threads
 - 2.1 <https://www.javacodegeeks.com/2013/01/java-thread-pool-example-using-executors-and-threadpoolexecutor.html>
 - 2.2 https://www.tutorialspoint.com/java/util/timer_schedule_period.htm
 - 2.3 <https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html>
 - 2.4 <https://docs.oracle.com/javase/8/docs/api/java/util/Timer.html>
 - 2.5 <https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html>
 - 2.6 Fundamental Programming Techniques -Assignment 2: Support Presentation