Documentation

Assignment 2
QUEUES MANAGEMENT APPLICATION USING THREADS AND SYNCHRONIZATION MECHANISMS

Student name: Bojan Darian-Adelin

Group: 30423/1

CONTENTS

ASSIGNMENT OBJECTIVE

The objective of the assignment is to design and implement a management application which assigns clients to queues such that the waiting time is minimized. To solve this assignment, we are going to use threads and synchronization mechanisms, such that we can represent the average waiting time, the average service time, peak hour, and the simulation-time in a graphical interface.

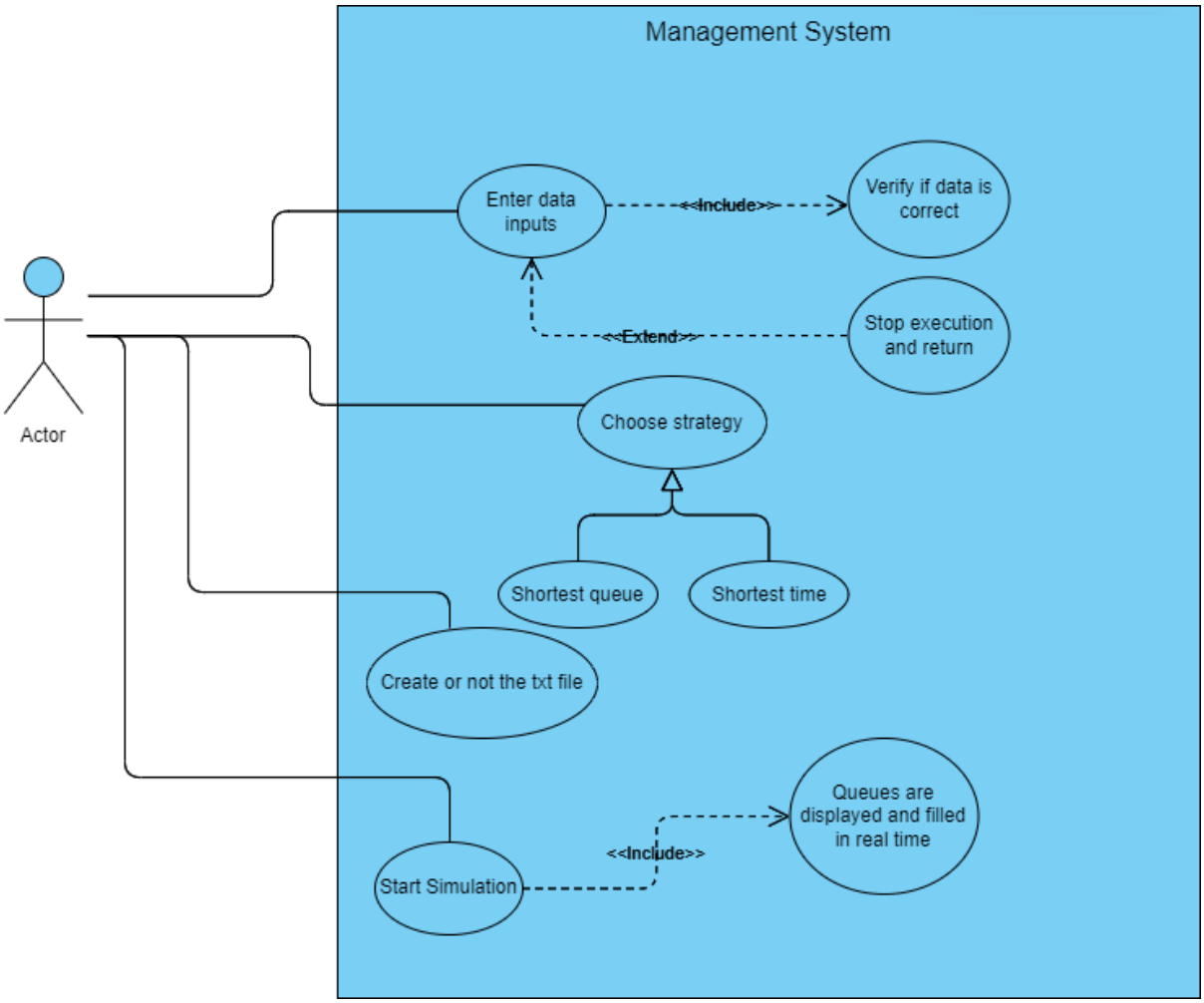
To achieve this task, it is necessary to satisfy the subsequent sub-objectives:

- Use object-oriented programming style (encapsulation, appropriate classes etc.).
- Implement a graphical user interface in Java Swing.
- Implement the random client generator.
- Usage of appropriate synchronized data structures to assure thread safety.
- Good organization of the code
- Log of events displayed in a graphical user interface/.txt file.
- Multithreading.
- Have 2 strategies: shortest queue and shortest time.
- Java naming conventions.

PROBLEM ANALYSIS, MODELING, SCENARIOS, USE CASES

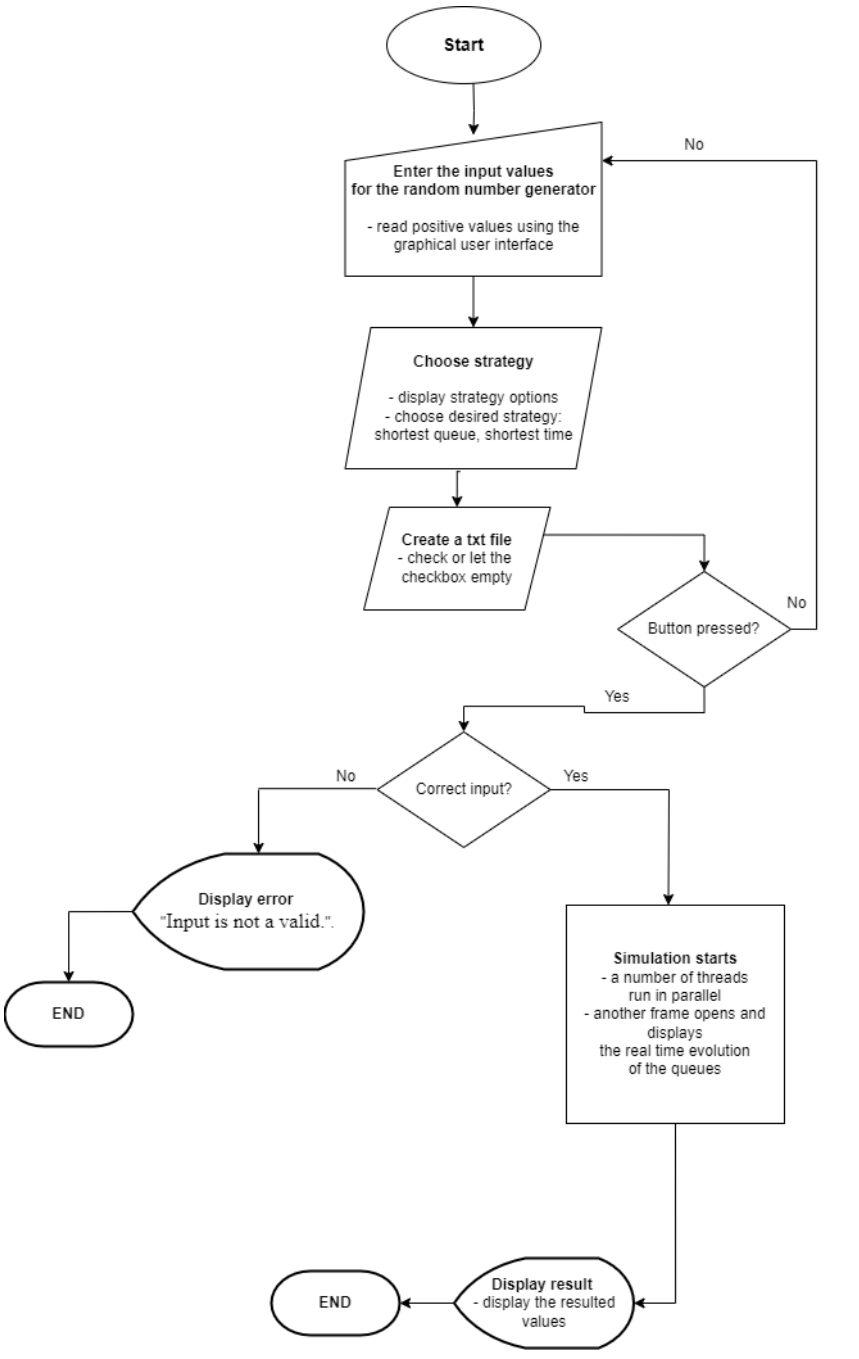
In order to solve this task, it is important to take into consideration the class model and the use cases such that we can have a clear view of the future development.

By starting with the use case diagram, we can gain a clearer understanding of the application we are developing. The user has the following interactions with the graphical user interface: entering the inputs for the random task generator, choosing the strategy for dividing clients to queues, enabling the creation of a txt file with the results or disabling it, and starting the execution of the threads.



Firstly, the inputs are written by the user in the graphical interface. They select a method and indicate whether they wish to generate a new file with the results. Subsequently, the program checks for errors (negative values or any non-number value). In case of negative values, or strings instead of numbers the execution will not start, and an error message will be displayed (“Input is not valid.”). Then, another frame starts with the real-time queues’ evolution.

The following **flowchart** outlines the actions within the application:



DESIGN OF THE APPLICATION

Classes, interfaces, and their responsibility:

- Main - initializes the application's graphical interface for a simulation and sets up an exit mechanism when the interface is closed.
- SimulationFrame – creates a GUI where the user can insert desired inputs.
- QueueViewer – frame for real-time queues' evolution
- SimulationManager – uses the inputs from the SimulationFrame, then creates a certain number of threads with the tasks generated by the random client generator.
- Strategy – an interface linked to TimeStrategy and ShortestQueueStrategy
- TimeStrategy – a class which has the main method of adding tasks according to a greedy algorithm which selects the shortest time.
- ShortestQueueStrategy – a class which has the main method of adding tasks according to an algorithm which selects the least number of people which wait in a queue.
- Scheduler – assigns tasks to servers according to the strategy chosen.
- Task – used to create objects of type (id, arrivalTime, serviceTime) which will be useful in the SimulationManager and Server classes.
- Server – the most important class, where threads start; implements Runnable and updates important fields in QueueViewer like average waiting time, average service time, or peak hour.

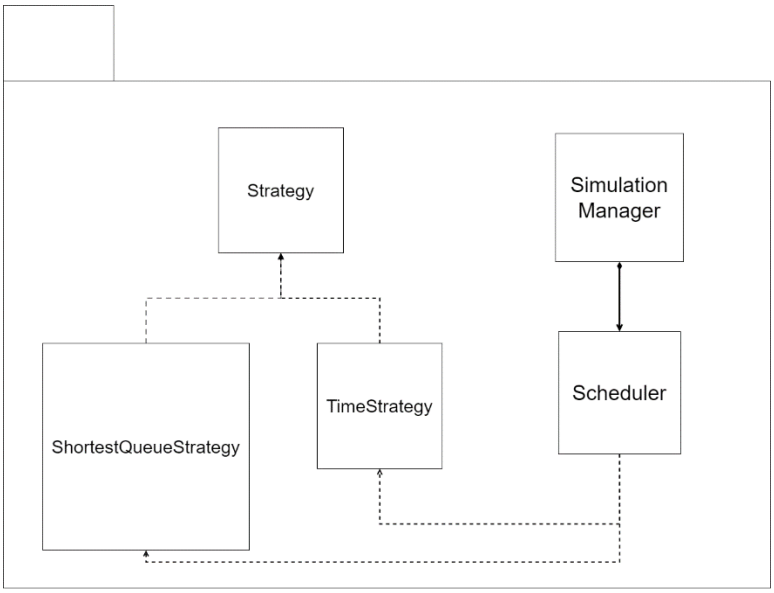
There are four packages in the application: BusinessLogic, GUI, MainApp and Model. Each one of them includes classes which have a common objective.

BusinessLogic contains 4 classes and an interface: Scheduler, ShortestQueueStrategy, TimeStrategy, SimulationManager, and Strategy. These classes are responsible for diving tasks correctly to servers based on the logic implemented in the TimeStrategy or ShortestQueueStrategy.

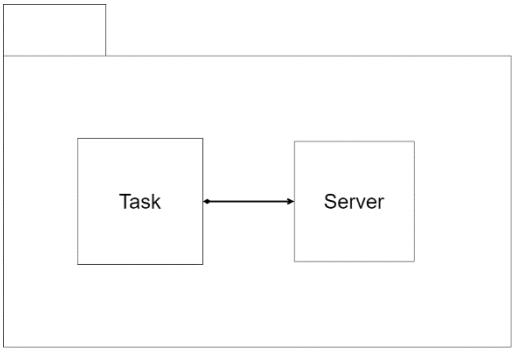
GUI package includes QueueViewer and SimulationFrame. The latter is the main frame where the user can insert inputs and then continue by starting the distribution of tasks to servers. After everything is set up, the QueueViewer opens a new frame where the user can see the real-time progression of the queues.

Model package has the classes Task and Server. Both are just a representation of the objects used in BusinessLogic. Task objects represent the clients and their waiting time, while Server serves as an object representation of the queue.

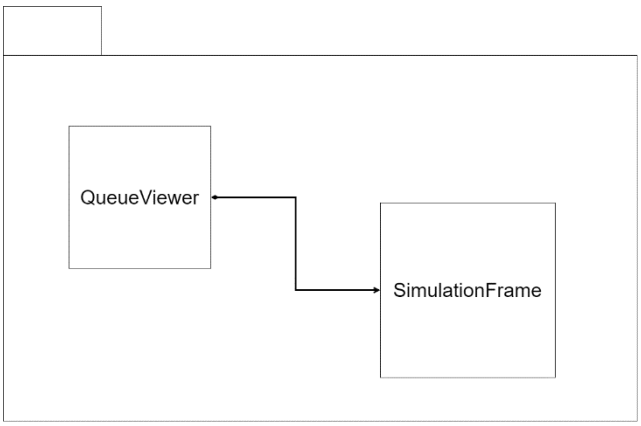
Lastly, the MainApp package contains the Main class which instantiates a new frame for the user to insert the inputs.



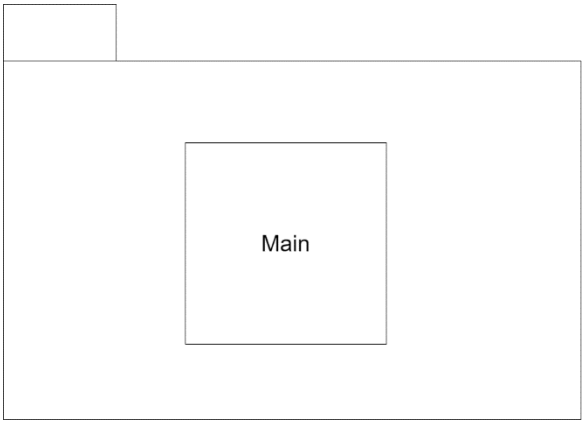
BusinessLogic



Model



GUI



MainApp

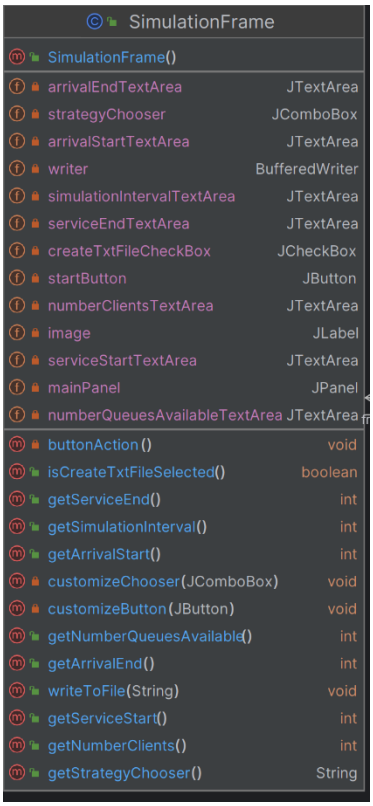
Interactions

- SimulationManager interacts with Scheduler, QueueViewer, Task, Server by managing the simulation process, generating random tasks, creating servers, starting processing, and updating UI elements.
- Scheduler interacts with Server, Task, Strategy, ShortestQueueStrategy, TimeStrategy having the responsibility of dispatching tasks to servers based on the selected strategy.
- QueueViewer interacts only with SimulationManager. It displays the progress of each client in the queues, updates UI elements such as average waiting time, average service time, and peak hour. Peak hour represents the time when the servers are most occupied.
- ShortestQueueStrategy and TimeStrategy interact with Server and Task.
- Server processes tasks, updates progress, and communicates with the SimulationManager and QueueViewer.
- Task has a relationship of association with Server class.

Important data structures used in this application:

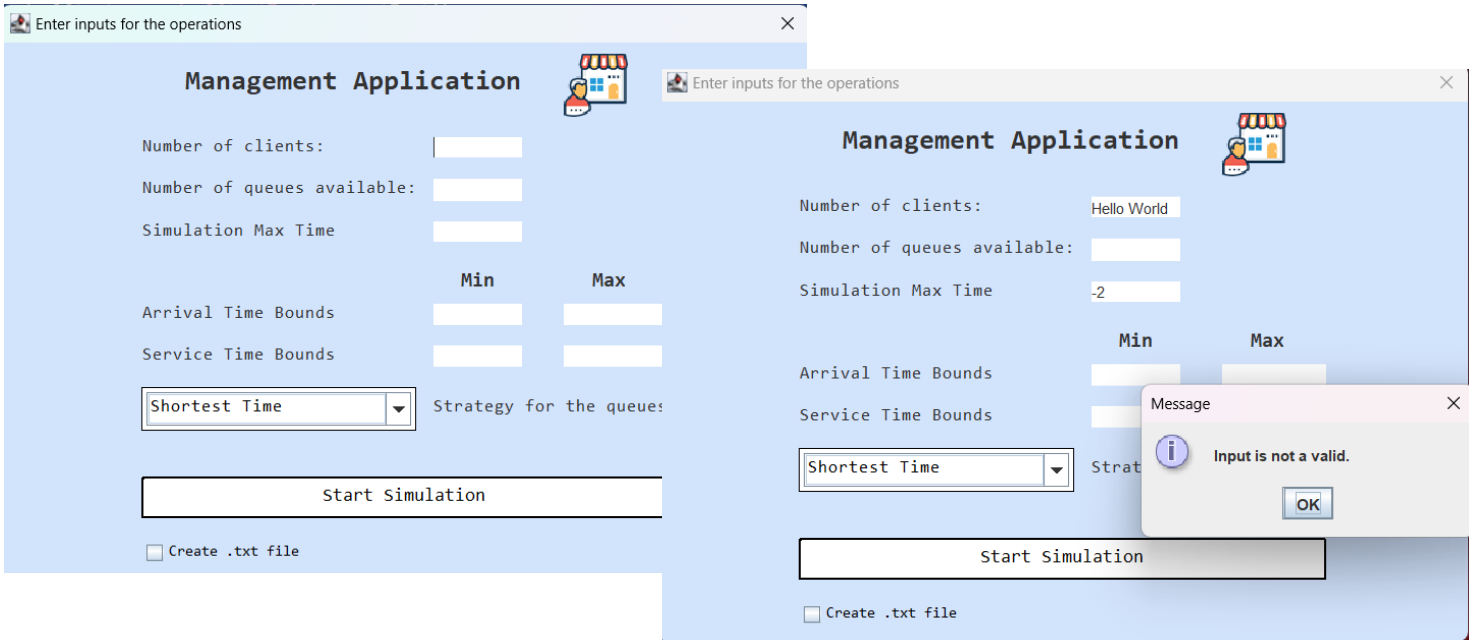
- LinkedBlockingQueue<Task> tasks – We are using an optionally-bounded blocking queue implementation because it allows multiple threads to access and modify the queue concurrently, by ensuring that tasks are processed in a synchronized and efficient manner.
- List printedTimes – A static synchronized list with the main goal of preventing duplicates of the real-time simulation process.
- ArrayList<Integer> waitingPeriodsForClients – stores waiting periods for individual clients in the server. It is useful in the ShortestQueueStrategy.
- private final List<Task> tasks – used in SimulationManager to store generated tasks.

IMPLEMENTATION

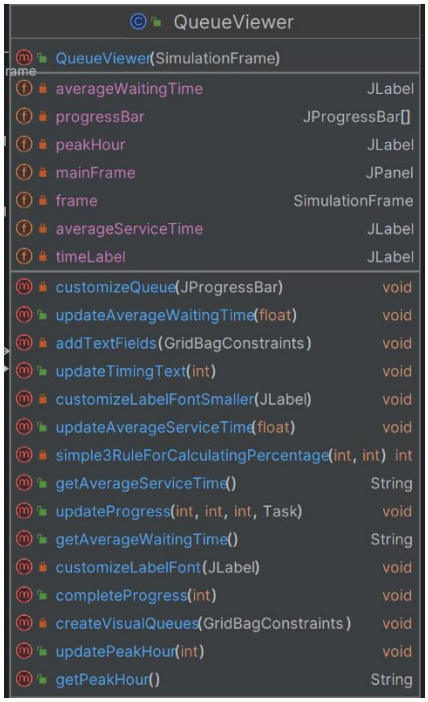


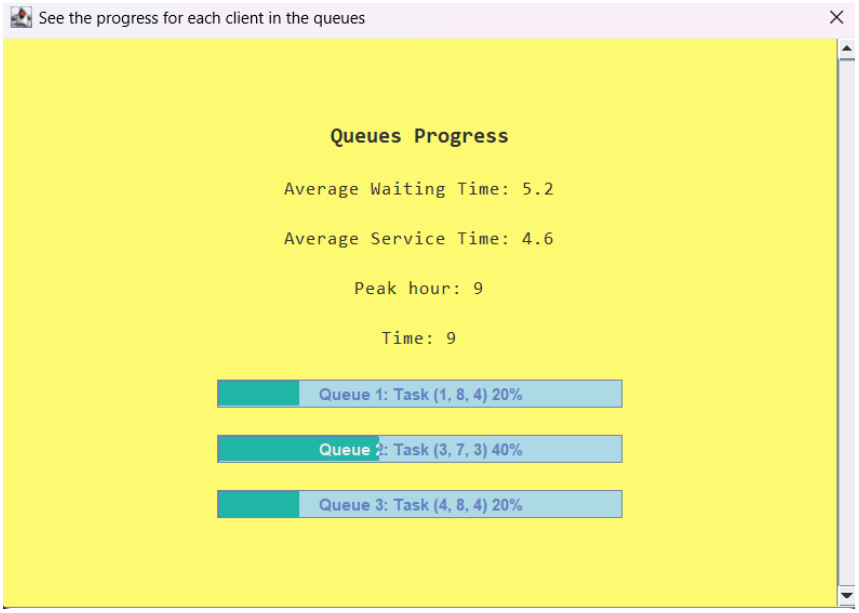
SimulationFrame represents the main graphical user interface of the application. Its objective is to construct and display a dialog window for the user interface. SimulationFrame serves as the intermediary for user interactions with the application's interface, preparing the inputs for the next phase. The interface includes seven fields for entering integer expressions, one JComboBox which lets the user choose the desired strategy and a button to display the evolution of the queues by calling the QueueViewer class. Each field value is tested before passing values to SimulationManager and starting the process. If the conversion from String to Integer is done successfully, then the integer values are tested again in the SimulationMaanger before starting the queue evolution process.

The UI of the project is simple and effectively allows users to enter desired inputs. If the values are not of the desired type, an error message will be displayed:

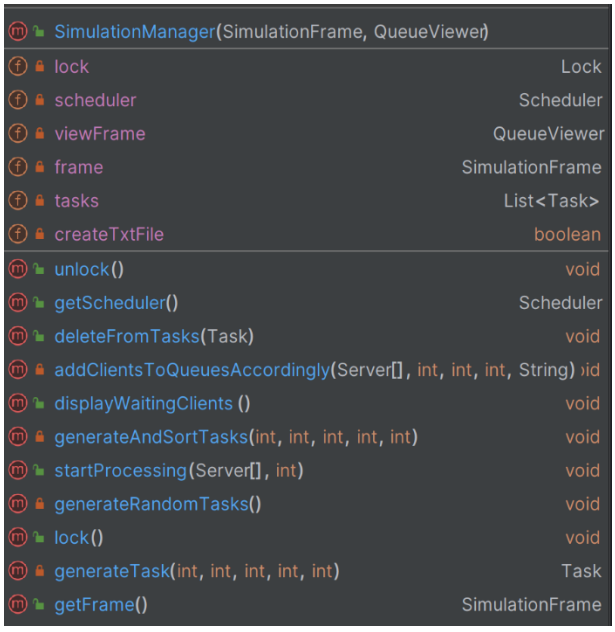


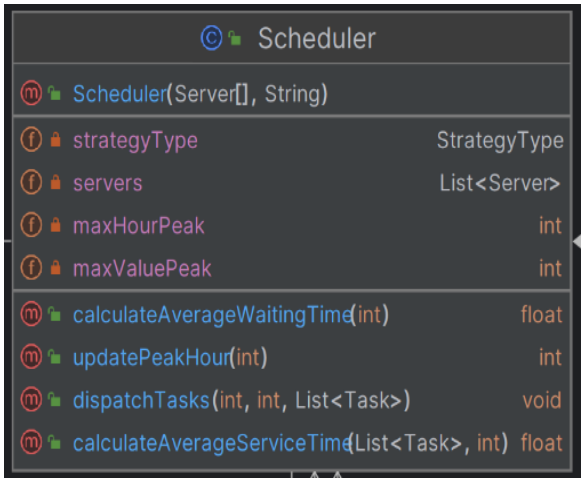
The class QueueViwer represents the class for the evolution of the queues. After everything is tested, a series of threads are launched, and the user can see in real-time the progress of the queues based on the desired strategy. This class is only a GUI and does not provide any special algorithms for finding important results.





The most important class, SimulationManager, serves as a central component responsible for orchestrating and managing the simulation process. Its objectives are orchestration of simulation process, communication with GUI, interaction with Scheduler, simulation control, error handling. In this class, the input is tested before starting the process from the QueueViewer. The inputs should be all positive integers and arrivalTimeStart <= arrivalTimeEnd, same rule being applied for serviceTimeStart and serviceTimeEnd. It represents the main control component of the application, dividing tasks to servers with the help of other classes (Scheduler, Strategy etc.).





The Scheduler class has the main objective of adding to servers the tasks received from the SimulationManager. Based on the chosen strategy, the scheduler class dispatches the tasks and calls a a new Strategy class, which is either ShortestQueueStrategy or TimeStrategy. Both of them have the purpose of minimizing the average waiting time. Waiting time is calculated as the sum of gaps between the arrival time and the time the client will enter the queue plus the service time.

ShortestQueueStrategy is the class where the tasks are given and stored in specific servers according to the principle of choosing the queue with the least number of people. The following algorithm shows the logic behind the task allocation:

public void addTask(List<Server> servers, Task task) {

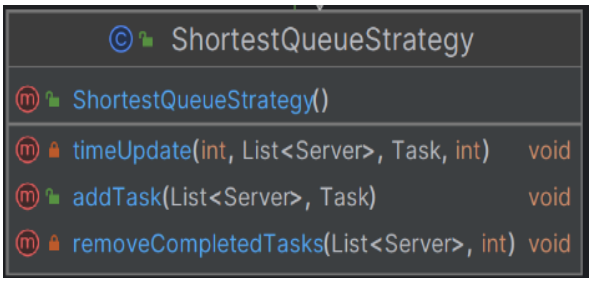
```
    if(!servers.isEmpty())
        {
            //check based on number of people
            removeCompletedTasks(servers, task.getArrivalTime());


            int smallestNumberOfWaiters =
servers.get(0).getWaitingPeriodsForClients().size();
            int timeForSmallestNumberOfWaiters =
servers.get(0).getWaitingPeriod().get();
            int index = 0;
            for(int i = 1; i < servers.size(); i++)
            {
                int numberOfPeopleCurrentServer =
servers.get(i).getWaitingPeriodsForClients().size();
                if(numberOfPeopleCurrentServer < smallestNumberOfWaiters)
                {
                    timeForSmallestNumberOfWaiters =
servers.get(i).getWaitingPeriod().get();
                    smallestNumberOfWaiters = numberOfPeopleCurrentServer;
```

```
                    index = i;
                }
            }
            timeUpdate(timeForSmallestNumberOfWaiters, servers, task, index);

servers.get(index).addElementToArray(servers.get(index).getWaitingPeriod().get());//add
waiting period for last client added
            servers.get(index).addTask(task); //add the client
        }
    }
```

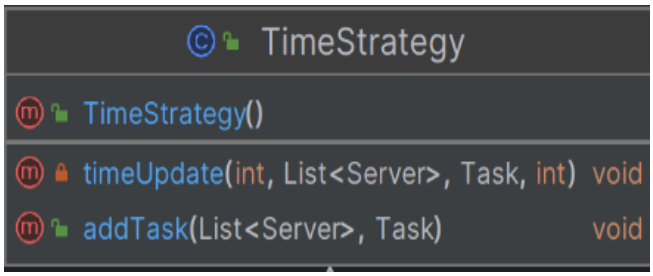
First, the algorithm updates the number of people if it's the case. Then it selects the least number of people. Finally, it updates the time and, also, the number of people which are waiting in the specific queue.



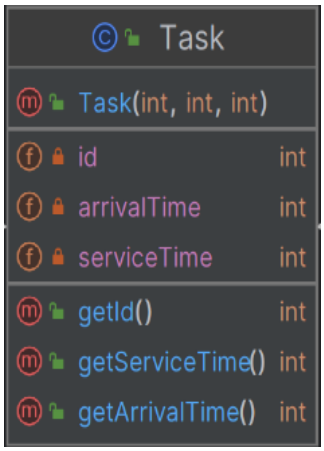
The TimeStrategy, on the other hand, calculates the next steps for adding tasks according to the minimum time a client needs to wait. It also updates the waiting time for other clients for the next tasks which we want to add.

```
    public void addTask(List<Server> servers, Task task) {
        if(!servers.isEmpty())
        {
            int smallestTime = servers.get(0).getWaitingPeriod().get();
            int index = 0;
            for(int i = 1; i < servers.size(); i++)
            {
                int waitingPeriodCurrentServer =
servers.get(i).getWaitingPeriod().get();
                if(waitingPeriodCurrentServer < smallestTime)
                {
                    smallestTime = waitingPeriodCurrentServer;
```

```
                index = i;
            }
        }
        timeUpdate(smallestTime, servers, task, index);
        servers.get(index).addTask(task);
    }
}
```



Task class is useful in the random client generator functionality from the SimulationManager. Its sole purpose is to create tasks which correspond to clients. The clients will wait in queues according to the desired strategy. It uses encapsulation and it behaves like a record.



The class which implements Runnable and where threads are run is the Server class. Here, in the run method the tasks are processed and executed parallelly. There are as many servers as the number of queues wanted by the user. After one task is finished, another one is taken from the LinkedBlockingQueue data structure which stores the next clients.

```
private void processTask(int simulationTimeThread, int currentArrivalTime, boolean
arrived, Task task, int currentServiceTime, int personalizedServiceTime, boolean end,
boolean wasInWhile) {

        while (numberOfPeople > 0 && simulationTimeThread <= simulationMaxInterval &&
!end) {
```

```
            try {simulationManager.lock();
                if (!arrived && simulationTimeThread >= currentArrivalTime)
                { arrived = true; on = true; simulationManager.deleteFromTasks(task);}
            } finally {
                simulationManager.unlock();
            }
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            printAndSet(simulationTimeThread);
            queueViewer.updateProgress(serverIndex, currentServiceTime,
personalizedServiceTime, task);
            if (arrived) {
                if (displayTxtFile) {
                    try {
                        simulationManager.getFrame().writeToFile("(" + task.getId() + ",
" + currentArrivalTime + ", " + (currentServiceTime - personalizedServiceTime) + ") in
queue: " + serverIndex);
                    } catch (IOException e) {
                        throw new RuntimeException(e);
                    }
                }
                personalizedServiceTime++;
            }
            if (personalizedServiceTime >= currentServiceTime)
            {on = false; end = true; numberOfPeople--;}
            wasInWhile = true;simulationTimeThread++;
        }
        continueOrEnd(wasInWhile, simulationTimeThread);
    }

    @Override
    public void run() {
        Task task;
```

```
        try {
            task = tasks.take();
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        int currentServiceTime = task.getServiceTime();
        int currentArrivalTime = task.getArrivalTime();
        int simulationTimeThread = timeSet.get();
        int personalizedServiceTime = 0;
        boolean arrived = false;
        boolean end = false;
        boolean wasInWhile = false;

        processTask(simulationTimeThread, currentArrivalTime, arrived, task,
currentServiceTime, personalizedServiceTime, end, wasInWhile);
    }
```

This code shows how the threads are executed. Firstly, the relevant data is retrieved, then the processTask method is called. Until the task is finished, it continues by calculating the simulation time, and during specific times, it updates the number of people, writes that a client has entered/is in the queue and so on.

RESULTS

Three tests were made to check the correctness of the application. As the task suggests, I included the input data from the tables below alongside the results in the project repository.

Test 1	Test 2	Test 3
N = 4 Q = 2 $t_{simulation}^{MAX}$ = 60 seconds $[t_{arrival}^{MIN}, t_{arrival}^{MAX}] = [2, 30]$ $[t_{service}^{MIN}, t_{service}^{MAX}] = [2, 4]$	N = 50 Q = 5 $t_{simulation}^{MAX}$ = 60 seconds $[t_{arrival}^{MIN}, t_{arrival}^{MAX}] = [2, 40]$ $[t_{service}^{MIN}, t_{service}^{MAX}] = [1, 7]$	N = 1000 Q = 20 $t_{simulation}^{MAX}$ = 200 seconds $[t_{arrival}^{MIN}, t_{arrival}^{MAX}] = [10, 100]$ $[t_{service}^{MIN}, t_{service}^{MAX}] = [3, 9]$

Here is the log of events for the first test with the results. The rest of the tests are in a separate folder in the gitlab repository.

Test1:

```
Time: 0

Waiting clients: (2, 12, 3), (4, 13, 4), (3, 15, 3), (1, 30, 2)

Time: 1

Waiting clients: (2, 12, 3), (4, 13, 4), (3, 15, 3), (1, 30, 2)

Time: 2

Waiting clients: (2, 12, 3), (4, 13, 4), (3, 15, 3), (1, 30, 2)

Time: 3

Waiting clients: (2, 12, 3), (4, 13, 4), (3, 15, 3), (1, 30, 2)

Time: 4

Waiting clients: (2, 12, 3), (4, 13, 4), (3, 15, 3), (1, 30, 2)

Time: 5

Waiting clients: (2, 12, 3), (4, 13, 4), (3, 15, 3), (1, 30, 2)

Time: 6

Waiting clients: (2, 12, 3), (4, 13, 4), (3, 15, 3), (1, 30, 2)

Time: 7

Waiting clients: (2, 12, 3), (4, 13, 4), (3, 15, 3), (1, 30, 2)

Time: 8

Waiting clients: (2, 12, 3), (4, 13, 4), (3, 15, 3), (1, 30, 2)

Time: 9

Waiting clients: (2, 12, 3), (4, 13, 4), (3, 15, 3), (1, 30, 2)

Time: 10

Waiting clients: (2, 12, 3), (4, 13, 4), (3, 15, 3), (1, 30, 2)

Time: 11

Waiting clients: (2, 12, 3), (4, 13, 4), (3, 15, 3), (1, 30, 2)

Time: 12

Waiting clients: (4, 13, 4), (3, 15, 3), (1, 30, 2)
```

```
(2, 12, 3) in queue: 1

Time: 13

Waiting clients: (3, 15, 3), (1, 30, 2)

(2, 12, 2) in queue: 1

(4, 13, 4) in queue: 2

Time: 14

Waiting clients: (3, 15, 3), (1, 30, 2)

(4, 13, 3) in queue: 2

(2, 12, 1) in queue: 1

Time: 15

Waiting client: (1, 30, 2)

(4, 13, 2) in queue: 2

(3, 15, 3) in queue: 1

Time: 16

Waiting client: (1, 30, 2)

(3, 15, 2) in queue: 1

(4, 13, 1) in queue: 2

Time: 17

Waiting client: (1, 30, 2)

(3, 15, 1) in queue: 1

Time: 18

Waiting client: (1, 30, 2)

Time: 19

Waiting client: (1, 30, 2)

Time: 20

Waiting client: (1, 30, 2)

Time: 21

Waiting client: (1, 30, 2)

Time: 22
```

```
Waiting client: (1, 30, 2)

Time: 23

Waiting client: (1, 30, 2)

Time: 24

Waiting client: (1, 30, 2)

Time: 25

Waiting client: (1, 30, 2)

Time: 26

Waiting client: (1, 30, 2)

Time: 27

Waiting client: (1, 30, 2)

Time: 28

Waiting client: (1, 30, 2)

Time: 29

Waiting client: (1, 30, 2)

Time: 30

(1, 30, 2) in queue: 2

Time: 31

(1, 30, 1) in queue: 2

Time: 32

Simulation Results:

Average Waiting Time: 3

Average Service Time: 3

Peak hour: 13
```

Because there were no people arriving earlier than expected, they did not have to wait any more, therefore the waiting time is equal to the service time. This example was used with the time strategy.

For the same input data, the following log of events was generated using the shortest queue strategy:

```
Time: 0
Waiting clients: (1, 14, 4), (2, 18, 3), (4, 20, 2), (3, 29, 2)
Time: 1
Waiting clients: (1, 14, 4), (2, 18, 3), (4, 20, 2), (3, 29, 2)
Time: 2
Waiting clients: (1, 14, 4), (2, 18, 3), (4, 20, 2), (3, 29, 2)
Time: 3
Waiting clients: (1, 14, 4), (2, 18, 3), (4, 20, 2), (3, 29, 2)
Time: 4
Waiting clients: (1, 14, 4), (2, 18, 3), (4, 20, 2), (3, 29, 2)
Time: 5
Waiting clients: (1, 14, 4), (2, 18, 3), (4, 20, 2), (3, 29, 2)
Time: 6
Waiting clients: (1, 14, 4), (2, 18, 3), (4, 20, 2), (3, 29, 2)
Time: 7
Waiting clients: (1, 14, 4), (2, 18, 3), (4, 20, 2), (3, 29, 2)
Time: 8
Waiting clients: (1, 14, 4), (2, 18, 3), (4, 20, 2), (3, 29, 2)
Time: 9
Waiting clients: (1, 14, 4), (2, 18, 3), (4, 20, 2), (3, 29, 2)
Time: 10
Waiting clients: (1, 14, 4), (2, 18, 3), (4, 20, 2), (3, 29, 2)
Time: 11
Waiting clients: (1, 14, 4), (2, 18, 3), (4, 20, 2), (3, 29, 2)
Time: 12
Waiting clients: (1, 14, 4), (2, 18, 3), (4, 20, 2), (3, 29, 2)
Time: 13
Waiting clients: (1, 14, 4), (2, 18, 3), (4, 20, 2), (3, 29, 2)
Time: 14
Waiting clients: (2, 18, 3), (4, 20, 2), (3, 29, 2)
(1, 14, 4) in queue: 1
Time: 15
Waiting clients: (2, 18, 3), (4, 20, 2), (3, 29, 2)
(1, 14, 3) in queue: 1
```

```
Time: 16
Waiting clients: (2, 18, 3), (4, 20, 2), (3, 29, 2)
(1, 14, 2) in queue: 1
Time: 17
Waiting clients: (2, 18, 3), (4, 20, 2), (3, 29, 2)
(1, 14, 1) in queue: 1
Time: 18
Waiting clients: (4, 20, 2), (3, 29, 2)
(2, 18, 3) in queue: 2
Time: 19
Waiting clients: (4, 20, 2), (3, 29, 2)
(2, 18, 2) in queue: 2
Time: 20
Waiting client: (3, 29, 2)
(2, 18, 1) in queue: 2
(4, 20, 2) in queue: 1
Time: 21
Waiting client: (3, 29, 2)
(4, 20, 1) in queue: 1
Time: 22
Waiting client: (3, 29, 2)
Time: 23
Waiting client: (3, 29, 2)
Time: 24
Waiting client: (3, 29, 2)
Time: 25
Waiting client: (3, 29, 2)
Time: 26
Waiting client: (3, 29, 2)
Time: 27
Waiting client: (3, 29, 2)
Time: 28
Waiting client: (3, 29, 2)
Time: 29
(3, 29, 2) in queue: 1
Time: 30
(3, 29, 1) in queue: 1
```

```
Time: 31
Simulation Results:
Average Waiting Time: 2.75
Average Service Time: 2.75
Peak hour: 20
```

CONCLUSION

In conclusion, the management application project provides a user-friendly tool for visualizing the process of diving clients to certain queues according to a rule specified by the user. Through encapsulation and modular design, the project ensures clean code organization and ease of maintenance.

This project helped in understanding the concepts of threads and synchronization mechanisms by minimizing the average waiting time for the clients.

As future enhancements, the application can be improved by creating another button in the QueueViewer which has the aim of changing the strategy during a certain time, from shortest queue to shortest time. The application can also be enhanced in terms of strategies as well: adding other ways to distribute people (a strategy aimed to have the waiting time, excluding the service time, as similar as possible to other clients; priority-based scheduling, Deadline-driven Scheduling).

BIBLIOGRAPHY

Lecture materials and other resources: https://dsrl.eu/courses/pt/

Understanding concurrency:
https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

Understanding threads using Runnable:
https://www.geeksforgeeks.org/runnable-interface-in-java/

Understanding the usage and purpose of LinkedBlockingQueue:

https://www.geeksforgeeks.org/linkedblockingqueue-class-in-java/

Reentrant lock: https://www.geeksforgeeks.org/reentrant-lock-java/