DOCUMENTATION

ASSIGNMENT *2*

STUDENT NAME: Hoza Violeta Maria

GROUP: 30424

CONTENTS

1. Assignment Objective....3
2. Problem Analysis, Modeling, Scenarios, Use Cases....3
3. Design....5
4. Implementation....6
5. Results....8
6. Conclusions....9
7. Bibliography....9

1. Assignment Objective

i) ***Main Objective***: Designing and implementing an application aimed at analyzing queue-based systems. This application should simulate customers waiting to receive a service (e.g. supermarket, bank, etc.). Just like in the real world, they must wait in queues, each queue processing clients simultaneously. The idea is to analyze how many clients can be served in a certain simulation interval, by entering parameters in an intuitive, user-friendly, application graphical interface.

ii) Sub-objectives:

• Analyzing the given problem and finding the requirements for solving it
• Designing and the implementation of the queue management system
• Determine the peak hour, average waiting time and the average service time
• Designing a user-friendly graphical interface

2. Problem Analysis, Modeling, Scenarios, Use Cases

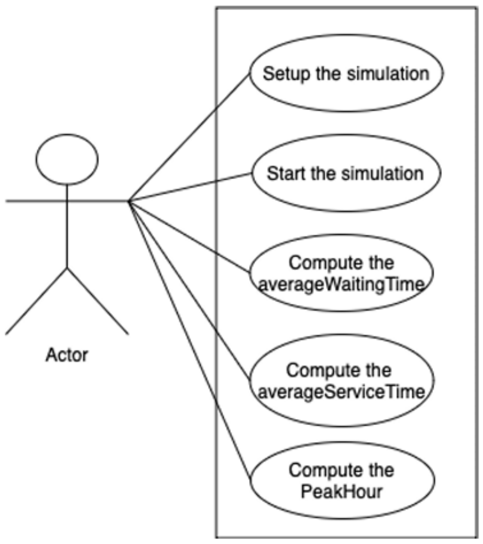
i) Functional requirements

- Must have: Simulation of a queue in real time with a given number of clients, queues, and maximum simulation time. Saving the results in a log file. Queues working as separate threads.
- Should have: GUI that receives data from the user and uses it to process the simulation. Data should be validated. Real time simulation of queue evolution. Calculation of peak hour, average waiting time and the average service time.
- Could have: Selection of desired strategy: shortest waiting time, lowest number of clients, etc.
- Won't have: Visualization of clients that left the queues.

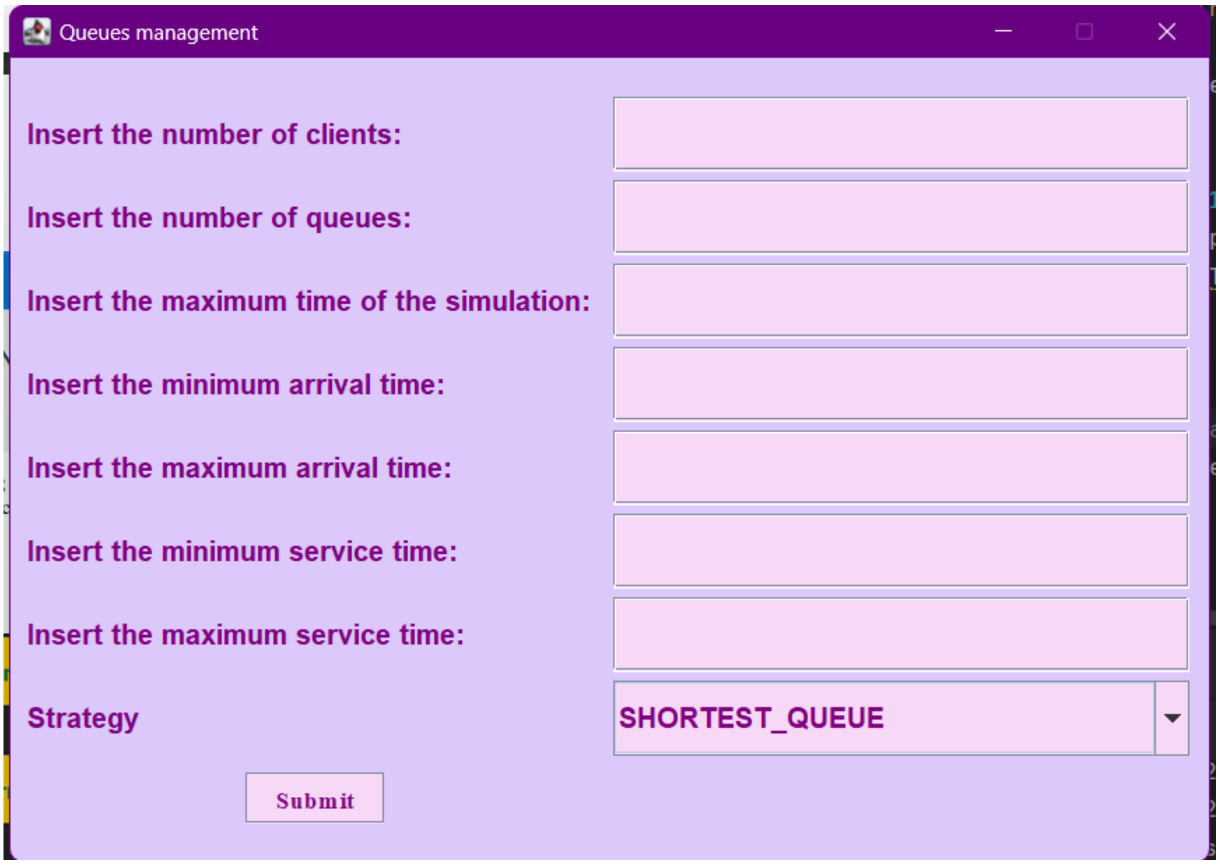
ii) Non-functional requirements:

- The user interface should be simple and intuitive, with labels that indicate what the application is expecting from the user.
- Compact design.

iii) Use cases:



- Main actor: the user
- The user enters the data in the text areas. The values are then validated by the application. If the validation passed, the simulation starts with the given parameters.

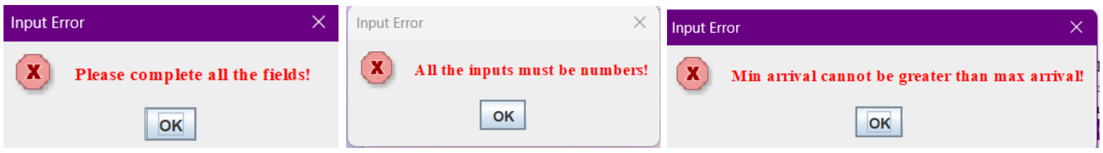


Queues management

Insert the number of clients:	
Insert the number of queues:	
Insert the maximum time of the simulation:	
Insert the minimum arrival time:	
Insert the maximum arrival time:	
Insert the minimum service time:	
Insert the maximum service time:	
Strategy	SHORTEST_QUEUE

- The user enters non – numerical values in the text areas. A notification appears on the screen letting the user know that there is a wrong input.

- The user enters numerical values, but the logic tests are not passed (e.g., the total simulation time is smaller than the maximum arrival time). A similar notification message appears and informs the user about the invalidity of the provided data.


Input Error
Please complete all the fields!
OK

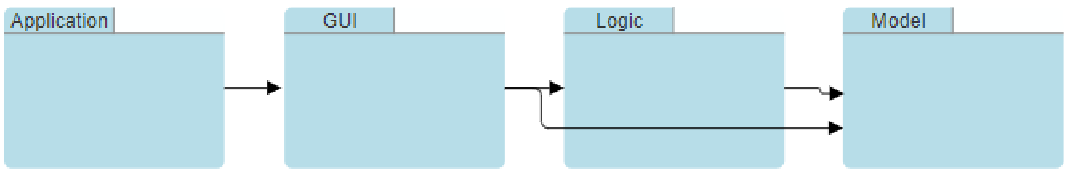
Input Error
All the inputs must be numbers!
OK

Input Error
Min arrival cannot be greater than max arrival!
OK

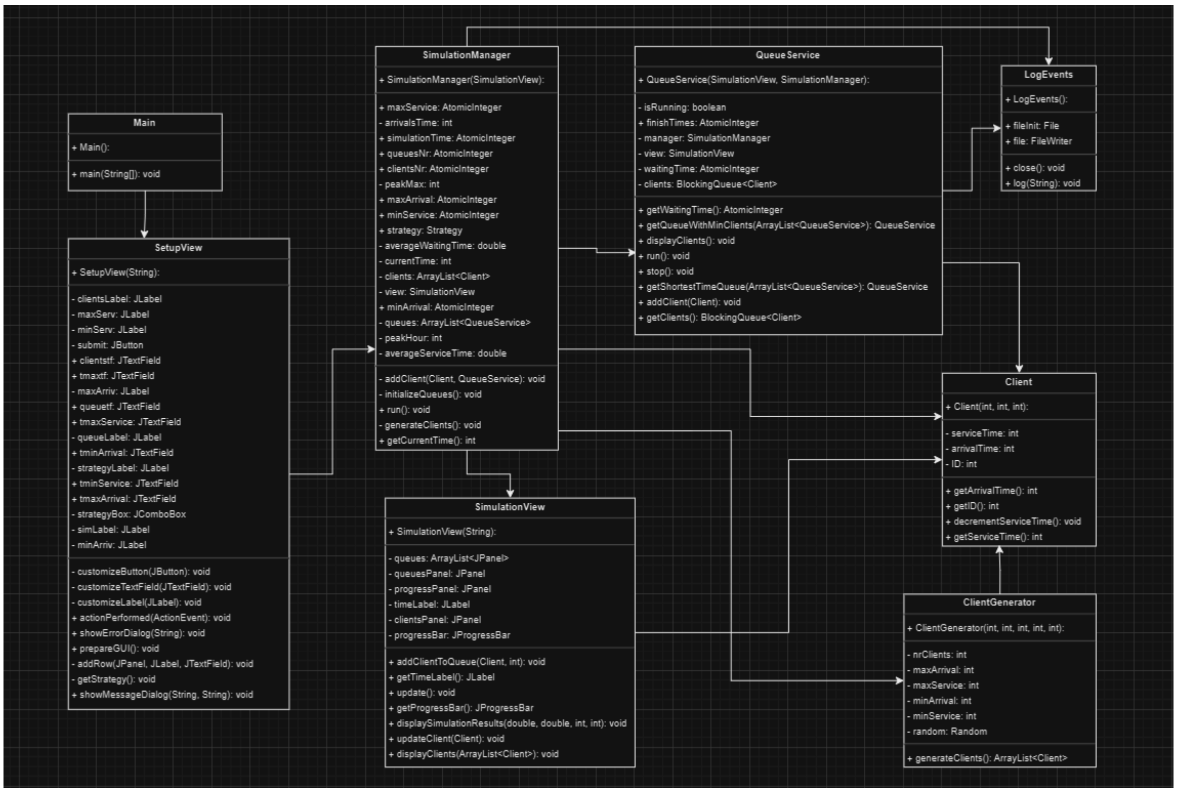
3. Design

- Package diagram:

The application is divided into several packages, each containing related classes. These packages include org.example.application, org.example.GUI, org.example.logic, and org.example.models. Each package contains classes responsible for specific functionalities, such as the main application logic (SimulationManager), GUI components (SetupView, SimulationView), and clients and queues representation (Client, QueueService).



- Class Diagram:


Main
+ Main():
+ main(String[]): void
SetupView
+ SetupView(String):
- clientsLabel: JLabel
- maxServ: JLabel
- minServ: JLabel
- submit: JButton
+ clientstf: JTextField
+ tmaxtf: JTextField
- maxArriv: JLabel
+ queuetf: JTextField
+ tmaxService: JTextField
- queueLabel: JLabel
+ tminArrival: JTextField
- strategyLabel: JLabel
+ tminService: JTextField
+ tmaxArrival: JTextField
- strategyBox: JComboBox
- simLabel: JLabel
- minArriv: JLabel
- customizeButton(JButton): void
- customizeTextField(JTextField): void
- customizeLabel(JLabel): void
+ actionPerformed(ActionEvent): void
+ showErrorDialog(String): void
+ prepareGUI(): void
- addRow(JPanel, JLabel, JTextField): void
- getStrategy(): void
+ showMessageDialog(String, String): void
SimulationManager
+ SimulationManager(SimulationView):
+ maxService: AtomicInteger
- arrivalsTime: int
+ simulationTime: AtomicInteger
+ queuesNr: AtomicInteger
+ clientsNr: AtomicInteger
- peakMax: int
+ maxArrival: AtomicInteger
+ minService: AtomicInteger
+ strategy: Strategy
- averageWaitingTime: double
- currentTime: int
- clients: ArrayList<Client>
- view: SimulationView
+ minArrival: AtomicInteger
- queues: ArrayList<QueueService>
- peakHour: int
- averageServiceTime: double
- addClient(Client, QueueService): void
- initializeQueues(): void
+ run(): void
- generateClients(): void
+ getCurrentTime(): int
SimulationView
+ SimulationView(String):
- queues: ArrayList<JPanel>
- queuesPanel: JPanel
- progressPanel: JPanel
- timeLabel: JLabel
- clientsPanel: JPanel
- progressBar: JProgressBar
+ addClientToQueue(Client, int): void
+ getTimeLabel(): JLabel
+ update(): void
+ getProgressBar(): JProgressBar
+ displaySimulationResults(double, double, int, int): void
+ updateClient(Client): void
+ displayClients(ArrayList<Client>): void
QueueService
+ QueueService(SimulationView, SimulationManager):
- isRunning: boolean
+ finishTimes: AtomicInteger
- manager: SimulationManager
- view: SimulationView
- waitingTime: AtomicInteger
- clients: BlockingQueue<Client>
+ getWaitingTime(): AtomicInteger
+ getQueueWithMinClients(ArrayList<QueueService>): QueueService
+ displayClients(): void
+ run(): void
+ stop(): void
+ getShortestTimeQueue(ArrayList<QueueService>): QueueService
+ addClient(Client): void
+ getClients(): BlockingQueue<Client>
LogEvents
+ LogEvents():
+ fileInit: File
+ file: FileWriter
+ close(): void
+ log(String): void
Client
+ Client(int, int, int):
- serviceTime: int
- arrivalTime: int
- ID: int
+ getArrivalTime(): int
+ getID(): int
+ decrementServiceTime(): void
+ getServiceTime(): int
ClientGenerator
+ ClientGenerator(int, int, int, int, int):
- nrClients: int
- maxArrival: int
- maxService: int
- minArrival: int
- minService: int
- random: Random
+ generateClients(): ArrayList<Client>

4. Classes Implementation

- Application: The Main class is used to start the application.
- Model

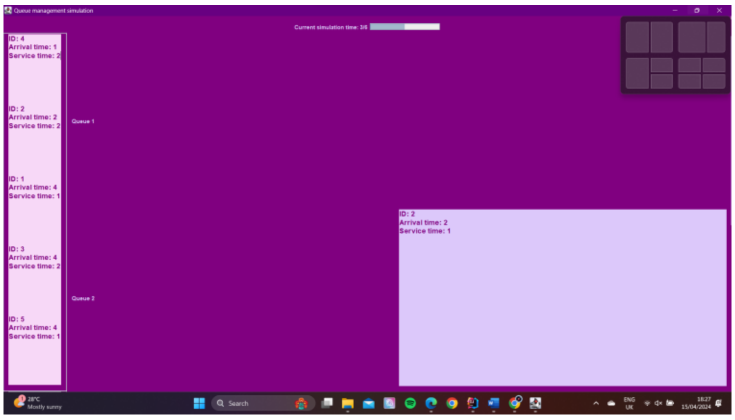
Client – contains data about each individual client and methods to obtain the private attributes of the class such as ID, arrival time and service time.

QueueService – implements the Runnable interface so it has a run() method and can be executed concurrently (because this class is a thread itself). In this method, operations are made on the first client in the queue. Clients are stored in an ArrayBlockingQueue to provide thread safety. The first client is removed if its task is completed. In the other case, it will decrease its remaining time in the queue. Besides its constructor, this class contains methods for displaying the clients from a queue, adding a client to a queue, getting the shortest queue and the queue with the shortest time.

- GUI

SetupView – this class displays the first frame of the application, the first view in other words. Here the user must introduce the data needed to start the simulation. This data consists of number of clients, number of queues, maximum simulation time, minimum arrival time, maximum arrival time, minimum service time, and maximum service time. This data is passed to the controller for validation.

SimulationView – the second frame (view) of the app. It displays the real time evolution of all the queues. The main frame has a BorderLayout and in the North side there is a so called "progress" panel that contains a label that represents the current time and is continuously updated and a progress bar that is also updated as one unit of time passes. In the Centre part another panel is placed. It has a GridLayout with 1 column and as many rows as the number of queues. Each of these rows is a panel on its own. Each panel also has a GridLayout that splits the client boxes between them.



Current simulation time: 3/6

ID: 4
Arrival time: 1
Service time: 2

ID: 2
Arrival time: 2
Service time: 2

ID: 1
Arrival time: 4
Service time: 1

ID: 3
Arrival time: 4
Service time: 2

ID: 5
Arrival time: 4
Service time: 1

ID: 2
Arrival time: 2
Service time: 1

- Logic

ClientGenerator: the class responsible for generating clients with random arrival and service times. It contains instance variables for the number of clients to generate, and the range of arrival and service times. It has a generateClients() method that generates a list of clients with random arrival and service times within the specified ranges. Each client is created with a unique ID, random arrival time, and random service time. For generating random integers for arrival and service times it uses a Random object.

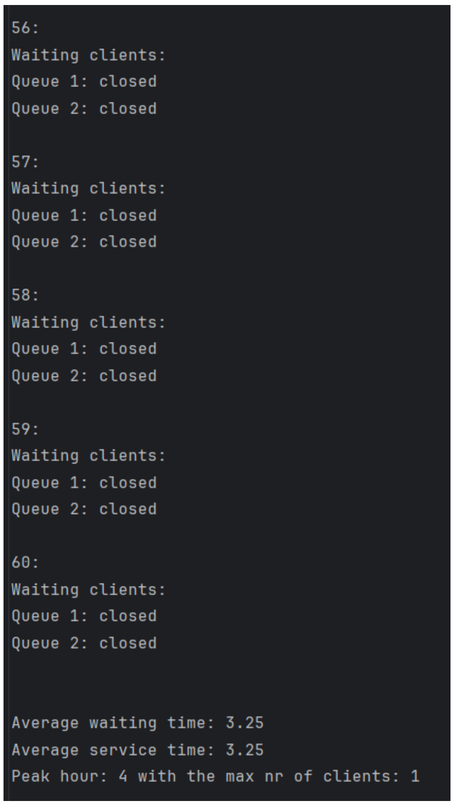
LogEvents: the class responsible for the logging of events to a file. It contains a static File object fileInit to define the log file path and a static FileWriter object file to write logs to the file. The static block initializes the FileWriter and creates the log file. This initialization is done only once for thread safety. It contains a log(String message) method which is synchronized to ensure thread safety and writes the provided message to the log file. When logging is done, the method close() closes the FileWriter.

SimulationManager: manages the clients, queues and simulation time; works as a thread. It contains some attributes that are used in more than one class (the inputs provided by the user). Here clients are initialized with random arrival time and service time but between the bounds set by the user for the arrival and service respectively. The run method for this class modifies the current time and dictates what should be performed. It sleeps for 1 second (1000 milliseconds) to make the simulation human perceptible. It also writes in the log file the current time and the values calculated for the peak hour, average waiting time and average service time. At each iteration, the current time increases by 1 until it reaches maximum simulation time. At each iteration, it goes through the clients ArrayList and checks if any client has the arrival time equal to the current time. If it does, it finds out the best queue based on the strategy chosen by the user. The client is added in that queue with the help of addClient method.

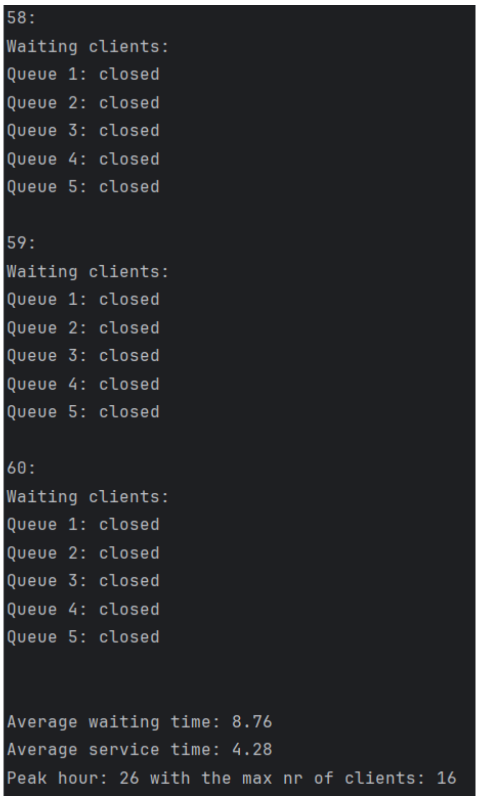
5. Results

The results of the 3 tests are stored in the text files test1.txt, test2.txt, test3.txt.

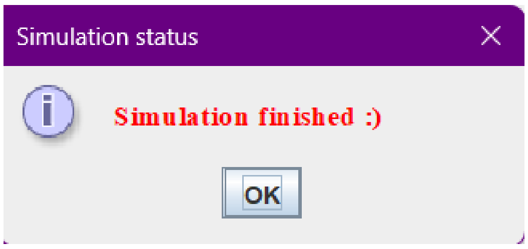
Test 1	Test 2	Test 3
N = 4 Q = 2 $t_{simulation}^{MAX}$ = 60 seconds $[t_{arrival}^{MIN}, t_{arrival}^{MAX}]$ = [2, 30] $[t_{service}^{MIN}, t_{service}^{MAX}]$= [2, 4]	N = 50 Q = 5 $t_{simulation}^{MAX}$ = 60 seconds $[t_{arrival}^{MIN}, t_{arrival}^{MAX}]$ = [2, 40] $[t_{service}^{MIN}, t_{service}^{MAX}]$= [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]



```
56:
Waiting clients:
Queue 1: closed
Queue 2: closed

57:
Waiting clients:
Queue 1: closed
Queue 2: closed

58:
Waiting clients:
Queue 1: closed
Queue 2: closed

59:
Waiting clients:
Queue 1: closed
Queue 2: closed

60:
Waiting clients:
Queue 1: closed
Queue 2: closed


Average waiting time: 3.25
Average service time: 3.25
Peak hour: 4 with the max nr of clients: 1
```

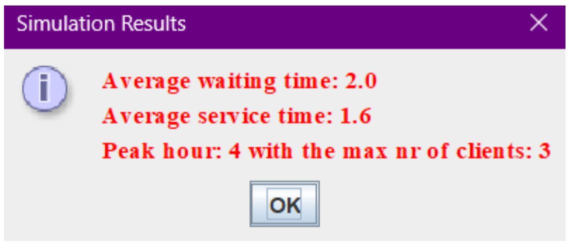


```
58:
Waiting clients:
Queue 1: closed
Queue 2: closed
Queue 3: closed
Queue 4: closed
Queue 5: closed

59:
Waiting clients:
Queue 1: closed
Queue 2: closed
Queue 3: closed
Queue 4: closed
Queue 5: closed

60:
Waiting clients:
Queue 1: closed
Queue 2: closed
Queue 3: closed
Queue 4: closed
Queue 5: closed


Average waiting time: 8.76
Average service time: 4.28
Peak hour: 26 with the max nr of clients: 16
```



Simulation status

Simulation finished :)



Simulation Results

Average waiting time: 2.0
Average service time: 1.6
Peak hour: 4 with the max nr of clients: 3

6. Conclusions

This project helped me discover new functionalities in Java programming language, for instance, how to read from a file and to use threads, which could be extremely useful for performing complicated tasks in the background without interrupting the main program and are fundamental in improving the performance of the application. All the main and secondary requirements have been accomplished. At the end of the assignment, I consider that my Java programming skills evolved since the last project.

7. Bibliography

1. *www.tutorialspoint.com*
2. *Object-Oriented Programming - Lecture Slides of prof. Marius Joldoș*
3. *Programming Techniques - Lectures of prof. Cristina Pop*
4. *app.smartdraw.com – for UML package diagram*
5. *app.creately.com – for the use case diagram*
6. *www.stackoverflow.com*
7. *www.geeksforgeeks.com*
8. *app.diagrams.net – for UML class diagram*