DOCUMENTATION

ASSIGNMENT 2

STUDENT NAME: Crăciunaș Victor
GROUP: 30425

CONTENTS

1. Assignment Objective

The primary objective of this assignment is to design and implement an application aiming to analyze queuing-based systems by simulating a series of N clients arriving for service, entering Q queues, waiting, being served and finally leaving the queues, and computing the average waiting time, average service time and peak hour .

The following table lists the sub-objectives, each describing necessary steps to achieve the main objective, along with the sections where these will be addressed:

Sub-Objective	Description	Section where it will be addressed
Analyze the problem and identify requirements	First, we'll look closely at what issues current queue systems have and figure out what our new system needs to fix these problems. This step helps us understand what features and functions our system should have to make it better and more user-friendly.	Problem addressed at section 2: Problem Analysis, Modeling, Scenarios ,Use cases
Design the simulation application	Next, we plan out how the application will work. This includes drawing up plans for how the app's parts fit together, choosing the right tools and methods for building it, and making sure it can be adjusted and expanded easily in the future.	Problem addressed at section 3: Design
Implement the simulation application	Then, we build the application according to our plans. This involves writing code, using the right programming tools, and making sure everything works together as intended. The goal here is to turn our designs into a working app.	Problem addressed at section 4: Implementation

Test the simulation application	Lastly, we check that the app works correctly by testing it. We'll look for any mistakes and make sure it does what it's supposed to do under different conditions. This step ensures the app is ready to be used without problems.	Problem addressed at section 5: Results

2. Problem Analysis, Modeling, Scenarios, Use Cases

In the problem analysis phase, I carefully examined the requirements and identified the functional and non-functional ones for the simulation application, aiming to give the most efficient way to distribute a client to a queue in order to minimize the amount of time they wait in queues before they are served.

For functional requirements, the following functions describe what my simulation application can do:

- Setup Simulation: The application will present a user-friendly interface for inputting simulation parameters. This includes specifying the number of clients, the number of queues, and timing parameters like the simulation interval, minimum and maximum arrival times, and minimum and maximum service times. It will validate these inputs to ensure they are within acceptable ranges before the simulation starts.

- Start the Simulation: Upon successful setup, the user can begin the simulation with a single action, such as clicking a 'Start' button. The application will then simulate the queuing process based on the parameters provided, managing clients as they arrive and are serviced by the system.

- Display Real-time Queues Evolution: As the simulation starts, the application will graphically display the queues in real-time, showing the addition and removal of clients as they are processed. This live feedback allows users to visually monitor and understand the flow of clients through the system.

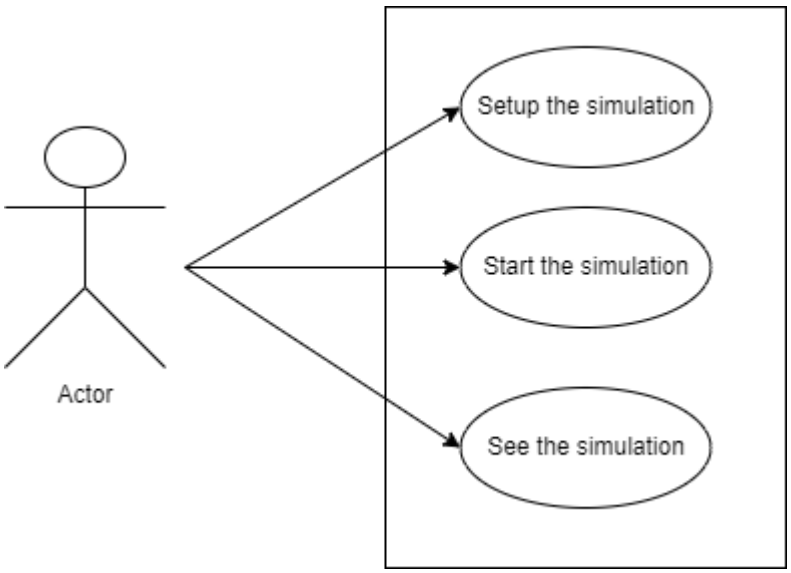
- Reporting and Analytics: After the simulation is complete, or upon user request, the application will offer a summary report. This report would include analytics on performance, such as average waiting time per client, queue length over time, and other relevant statistics.

- Stop the simulation: As an additional functional requirement, I've added a stop option where the user can stop the simulation and clear the inputs in order to start a new one.

And as for the non-functional functions:

- Intuitive User Interface: The system is user-friendly, with a well-organized interface that's easy to navigate. Users will be able to perform all functions intuitively, without the need for in-depth tutorials or guidance.
- System Performance: The application will be responsive, with minimal lag between user actions and system responses, even when handling a large number of simultaneous clients in the simulation.

For the use cases, I developed clear descriptions to illustrate how users would interact with my app:



Use Case: setup simulation
Primary Actor: User
Main Success Scenario:

1. The user inserts the values for the: number of clients, number of queues, simulation interval, minimum and maximum arrival time, and minimum and maximum service time.

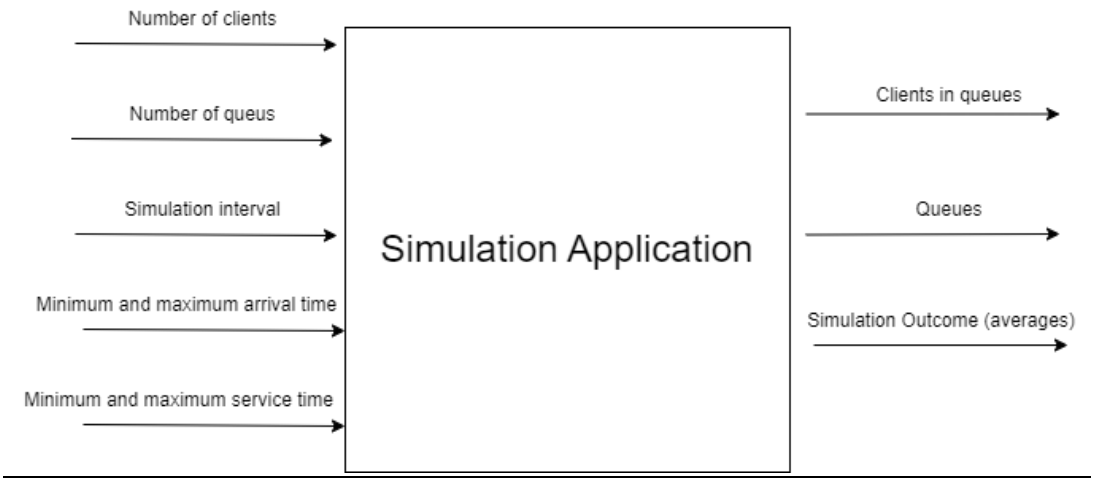
2. The user clicks on the validate input data button
3. The application validates the data and displays a message informing the user to start the simulation

Alternative Sequence: Invalid values for the setup parameters

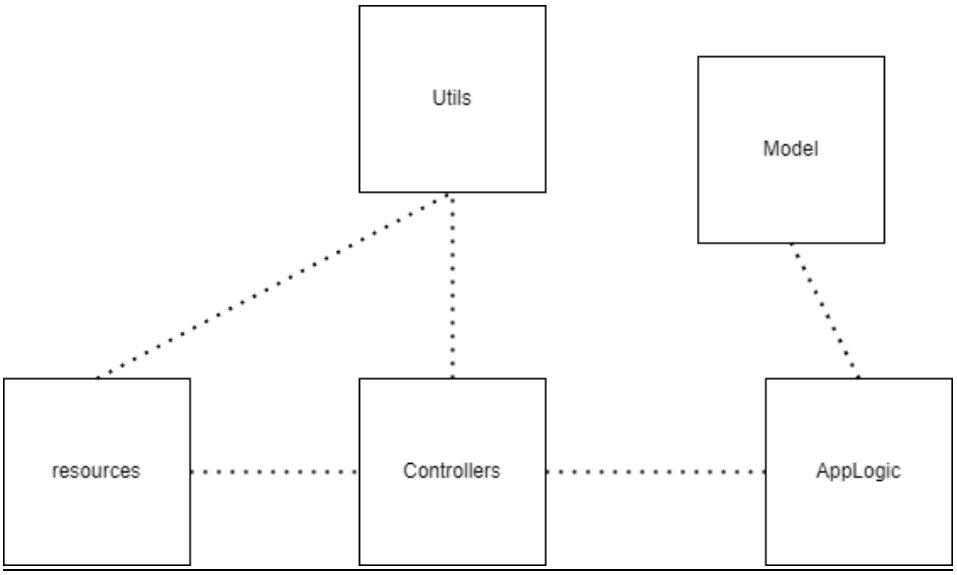
-The user inserts invalid values for the application's setup parameters
-The application displays an error message and requests the user to insert valid values
-The scenario returns to step 1

3. Design

Overall design

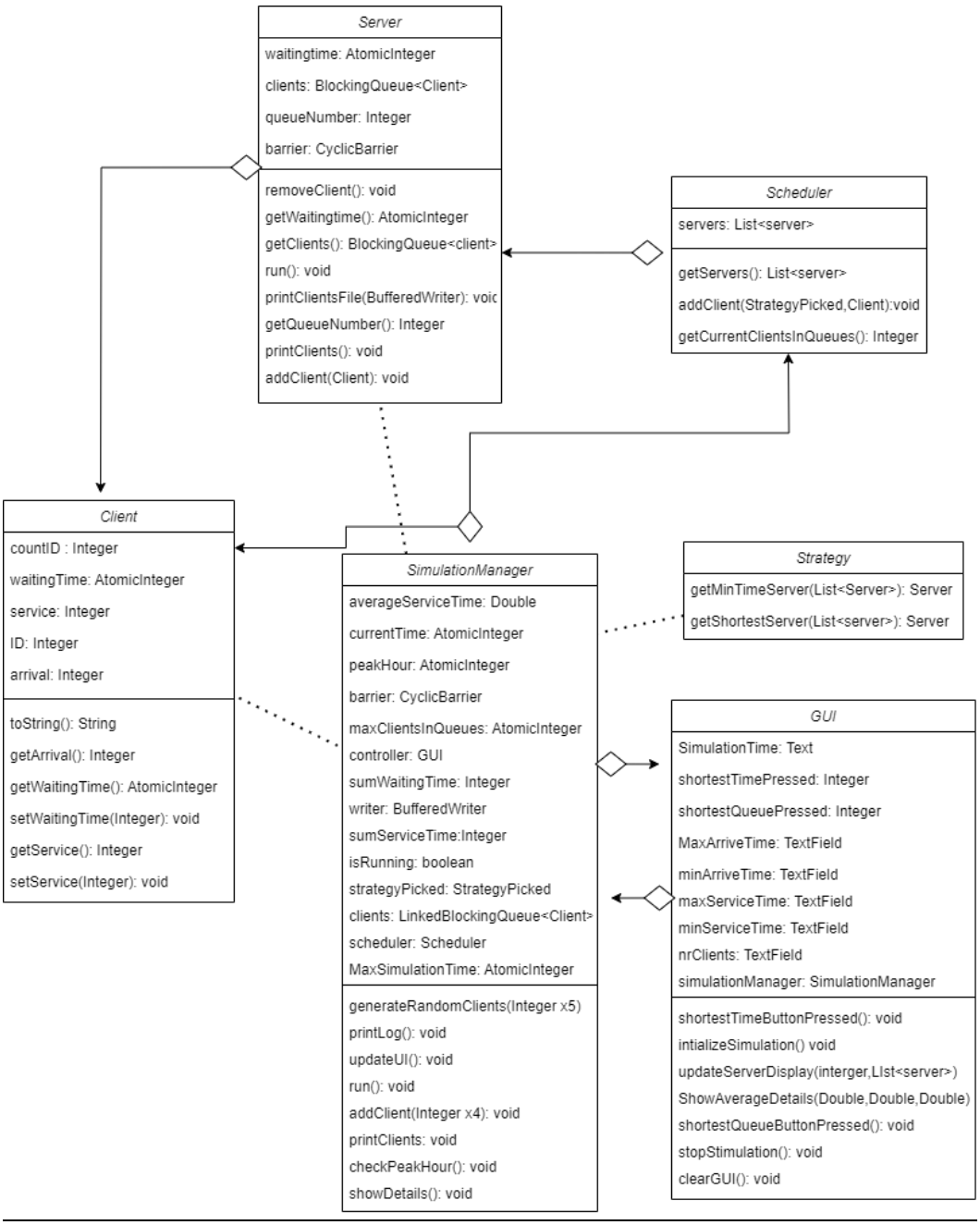


Package Diagram



- Utils: The package where the classes that launch the app are;
- Resources: The package where we make the design of the app;
- Controllers: The package where the controllers of the views are(the place where GUI classes are);
- AppLogic: The package where the classes that contains the app's logic are;
- Model: Contains the Server and Client classes.

Class Diagram



Data structures used:

- BlockingQueue: I use BlockingQueue to create the queues where the clients are going to wait until they get serverd. It also is a thread safe option which is necessary to have an expected behavior;
- ArrayList: I use ArrayList in order to save the queues and the clients.

4. Implementation

- Client

The Client class represents a client or a customer who will be managed within the simulation application.

Properties:

-countID (private static Integer): A static counter used to assign unique IDs to each client instance. It starts at 1 and increments with the creation of each new client.

-ID (public Integer): The unique identifier for the client, assigned upon creation.

-arrival (public Integer): The time at which the client arrives in the system.

-service (public Integer): The time required to service the client.

-waitingTime (public AtomicInteger): An atomic integer representing the client's waiting time in the queue plus the service time. Using AtomicInteger ensures thread-safe manipulation of the waiting time in a multi-threaded environment.

Methods:

-getWaitingTime(): Retrieves the current waiting time of the client.

-setWaitingTime(Integer waitingTime): Sets the client's waiting time. This is a thread-safe operation due to the use of AtomicInteger.

-Client(Integer arrival, Integer service): A constructor that initializes the client with arrival and service times and assigns a unique ID using the incremented countID.

-getArrival(): Returns the arrival time of the client.

-getService():Returns the service time for the client.

-toString():method gives a textual representation of the client's ID, arrival, and service times, simplifying the process of tracking client information during the simulation.

- Server

The Server class acts as a queue manager in the queue simulation system. This class is designed to handle the operations related to managing clients within a particular queue, using thread-safe mechanisms to ensure reliable operation in a multi-threaded environment.

Properties:

-clients (private BlockingQueue<Client>): A thread-safe queue that holds clients.

-waitingtime (private AtomicInteger): A counter that keeps track of the cumulative waiting time of all clients in the queue.

-queueNumber (public Integer): An identifier for the server's queue, allowing differentiation between multiple queue instances.

-barrier (private CyclicBarrier): A synchronization aid that allows the server to wait until all threads have reached a common barrier point, useful for synchronizing the start or end of a simulation step with other servers.

Constructor:

-Server(Integer MaxClients, Integer queueNumber, CyclicBarrier barrier): Initializes the server with a maximum client capacity, an identifier for the queue, and a barrier for synchronization with other threads.

Methods:

-addClient(Client client): Adds a client to the queue and updates the client's waiting time based on the current waiting time of the queue plus the client's service time.

-removeClient(): Removes a client from the queue

-getQueueNumber(): Retrieves the identifier for this queue.

-run(): The main thread execution method, which continually processes clients, updating waiting and service times, and synchronizes at a barrier point.

-getWaitingtime(): Provides access to the current waiting time of the queue.

-printClients(): Outputs the details of the clients in the queue to the console.

-getClients(): Returns the queue of clients.

-printClientsFile(BufferedWriter writer): Writes the details of the clients in the queue to a file using a BufferedWriter.

- Simulation Manager

This class is going to be the brain of the application and controls the flow of the simulation. It initialize the servers and generates the clients. This is the place where it put each client to a queue and updates the system's state at each time step. . The run method is going to run until it reaches the maximum simulation time or until or clients are served.

Key Components:

-clients: A LinkedBlockingQueue<Client> which stores the clients that are yet to be scheduled into servers. It ensures thread-safe operations when adding or removing clients.

-currentTime: An AtomicInteger tracking the current time step of the simulation.

-MaxSimulationTime: An AtomicInteger specifying the total duration of the simulation.

-scheduler: A Scheduler object responsible for distributing clients among available servers based on the specified strategy.

-controller: A GUI component linked to the graphical interface to update and display simulation results.

-barrier: A CyclicBarrier used to synchronize the operation of multiple server threads and the simulation manager itself.

-strategyPicked: An enumeration (assumed) that specifies the queuing strategy (like shortest queue first, round robin, etc.).

-writer: A BufferedWriter used for logging simulation results to a file.

- Scheduler

  The Scheduler class plays a key role in managing the distribution of clients to various servers in a queue simulation system. This class is specifically designed to allocate clients based on different strategies, ensuring efficient management of queues according to specified criteria.

- Strategy

  This is the class which two strategies for a more efficient distribution:

  -SHORTEST_TIME: Directs the client to the server with the shortest expected processing time, aiming to minimize overall wait times.

  -SHORTEST_QUEUE: Allocates the client to the server with the fewest number of waiting clients, potentially reducing the immediate wait time for the client being added.

- GUI

  -This is the class that creates the friendly-interface. At first it provides the TextFields where the user can insert his inputs for the simulation, then when the simulation starts, the interface is updated in real-time, showing how the clients are being processed.

5. Results

As for testing, the application has been tested with the following inputs:

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 $[t_{arrival}^{MIN}, t_{arrival}^{MAX}] = [10, 100]$ $[t_{service}^{MIN}, t_{service}^{MAX}] = [3, 9]$

Test 1:

```
Clients:  Client{ID=1, arrival=27, service=2}
Queue 1 closed
Queue 2 closed

TIME SIMULATION ------------ 27
Clients: empty
Queue: 1 |  Client{ID=1, arrival=27, service=2}
Queue 2 closed

TIME SIMULATION ------------ 28
Clients: empty
Queue: 1 |  Client{ID=1, arrival=27, service=1}
Queue 2 closed

TIME SIMULATION ------------ 29
Clients: empty
Queue 1 closed
Queue 2 closed
Average service Time: 2.25
Average waiting Time: 2.25
Peak hour: 2 with 1 clients
```

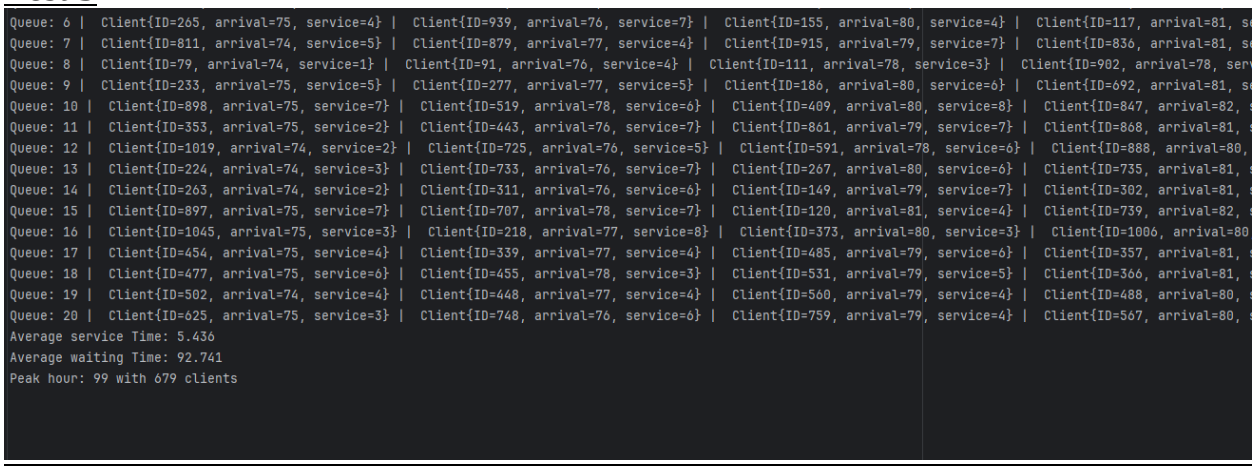
Test 2

```
Queue 4 closed
Queue: 5 |  Client{ID=53, arrival=39, service=1}

TIME SIMULATION ------------ 44
Clients: empty
Queue: 1 |  Client{ID=31, arrival=39, service=1}
Queue 2 closed
Queue 3 closed
Queue 4 closed
Queue 5 closed

TIME SIMULATION ------------ 45
Clients: empty
Queue 1 closed
Queue 2 closed
Queue 3 closed
Queue 4 closed
Queue 5 closed
Average service Time: 3.34
Average waiting Time: 4.02
Peak hour: 14 with 9 clients
```

Test 3



6. Conclusions

This project has significantly deepened my understanding of queue management systems through the practical application of Java and JavaFX, demonstrating the impact of different queuing strategies on efficiency. Key learnings include the importance of effective concurrency handling, the role of algorithm choice in system performance, and the need for thoughtful system design for scalability. Future developments could explore more complex algorithms and expand the system's capabilities to handle varying types of client interactions, potentially integrating real-time data inputs to further enhance the simulation's applicability to real-world scenarios.

7. Bibliography

1. https://dsrl.eu/courses/pt/materials/PT_2024_A2_S1.pdf
2. https://dsrl.eu/courses/pt/materials/PT_2024_A2_S2.pdf