DOCUMENTATION

ASSIGNMENT *2*

QUEUES MANAGEMENT APPLICATION

STUDENT NAME: Mihoc Daniel-Alexandru
GROUP: 30425

CONTENTS

1. Assignment Objective

Main Objective: Develop a JavaFX queues management application.

Sub-Objectives:

a. Design classes representing clients and queues (Client.java, ServiceQueue.java).

b. Design a management class that handles the queue assignment of clients and runs the simulation (QueueManager.java, SimulationTask.java)

c. Implement different strategies for client assignment in queues (QueueAssignmentStrategy.java interface) which is implemented by the ShortestQueueStrategy.java and ShortestTimeStrategy.java

d. Create a separate config module that can handle all the client creation configs (ClientConfig.java) and all the simulation configs (SimulationConfig.java).

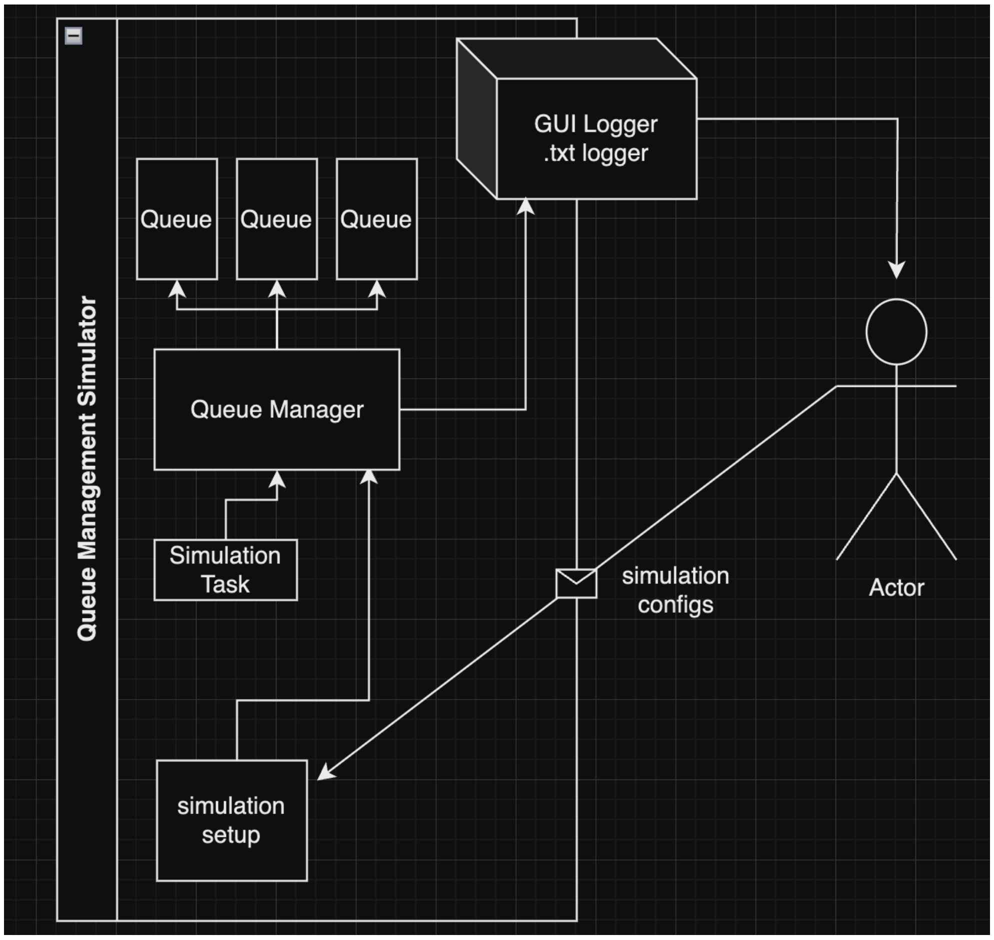
e. Create a log handler that can log to a text file in various other places for ease of use (LogHandler.java, SimulationLogger.java).

f. Create a JavaFX application with a graphical user interface (GUI) to enter the simulation and client creation configs and show the logs of the simulation (QueueManagerController.java).

2. Problem Analysis, Modeling Scenarios, Use Cases

The **Queue Management Simulator** is designed to model and analyze the process of queuing clients and servicing them using various strategies. This simulation will aid in understanding the dynamics of queue management and the efficiency of different queue-handling methods.

Use Case Diagram: The user flow begins with the actor (end-user) initiating the simulation by providing the configuration parameters for the simulation. These parameters include the number of queues, the total number of clients, and the simulation time. The actor can select a strategy for queue management, such as the shortest queue or shortest time strategy, and then start the simulation.



– Use Case Diagram (representing the User Flow when interacting with the simulation) –

The user flow diagram illustrates the steps an actor takes to interact with the simulation:

1. The actor inputs simulation configurations.
2. The actor selects a strategy for client assignment.
3. The actor starts the simulation.
4. The actor observes the simulation progress and results.

3. Design

OOP Design Principles Used:

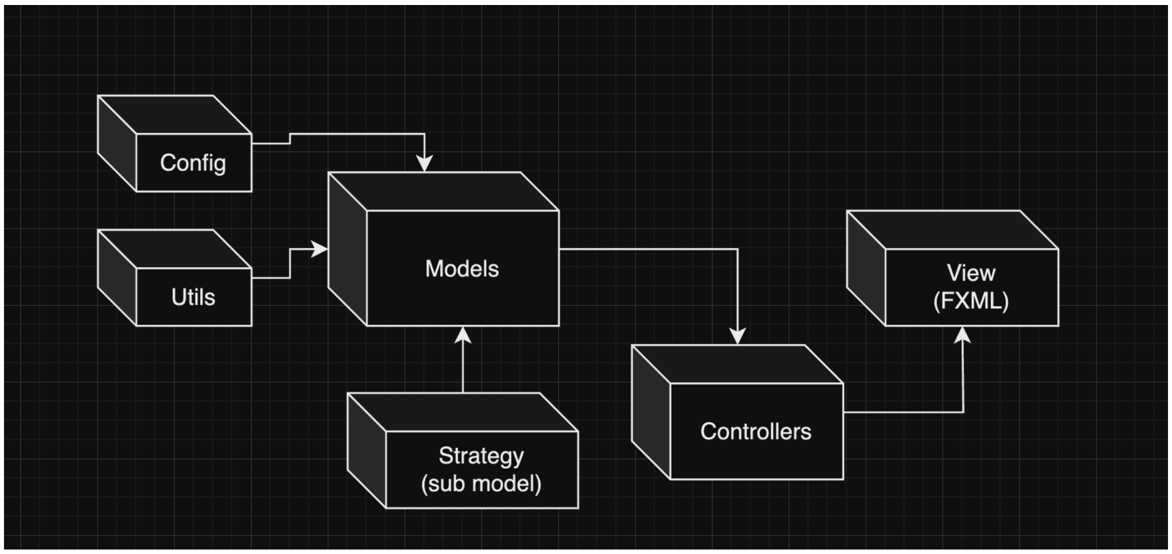
- **Encapsulation**: The application encapsulates the details of each class, exposing only necessary information through methods like run(), addClient(), and assignClientToQueue().
- **Abstraction**: The QueueAssignmentStrategy interface abstractly defines how clients are assigned to queues, allowing for different strategies without changing the manager or queues.
- **Inheritance**: Strategies like ShortestTimeStrategy and ShortestQueueStrategy inherit from the QueueAssignmentStrategy interface, leveraging polymorphism.
- **Modularity**: The application is divided into modules like Config, Utils, Models, Controllers, and View, promoting separation of concerns.

Package Structure:

The package structure follows a modular approach, separating the concerns into clear categories:

- Models: Contains classes like Client, ServiceQueue, and SimulationTask that define the data model and business logic.
- Strategy: Encapsulates the algorithms for queue assignment, serving as a sub-model within the application.

- Config: Holds configuration classes like ClientGenerationConfig and SimulationConfig.
- Controllers: Manages the interaction between the UI and the application logic, containing classes like QueueSimulationController



Data Structures & Algorithms:

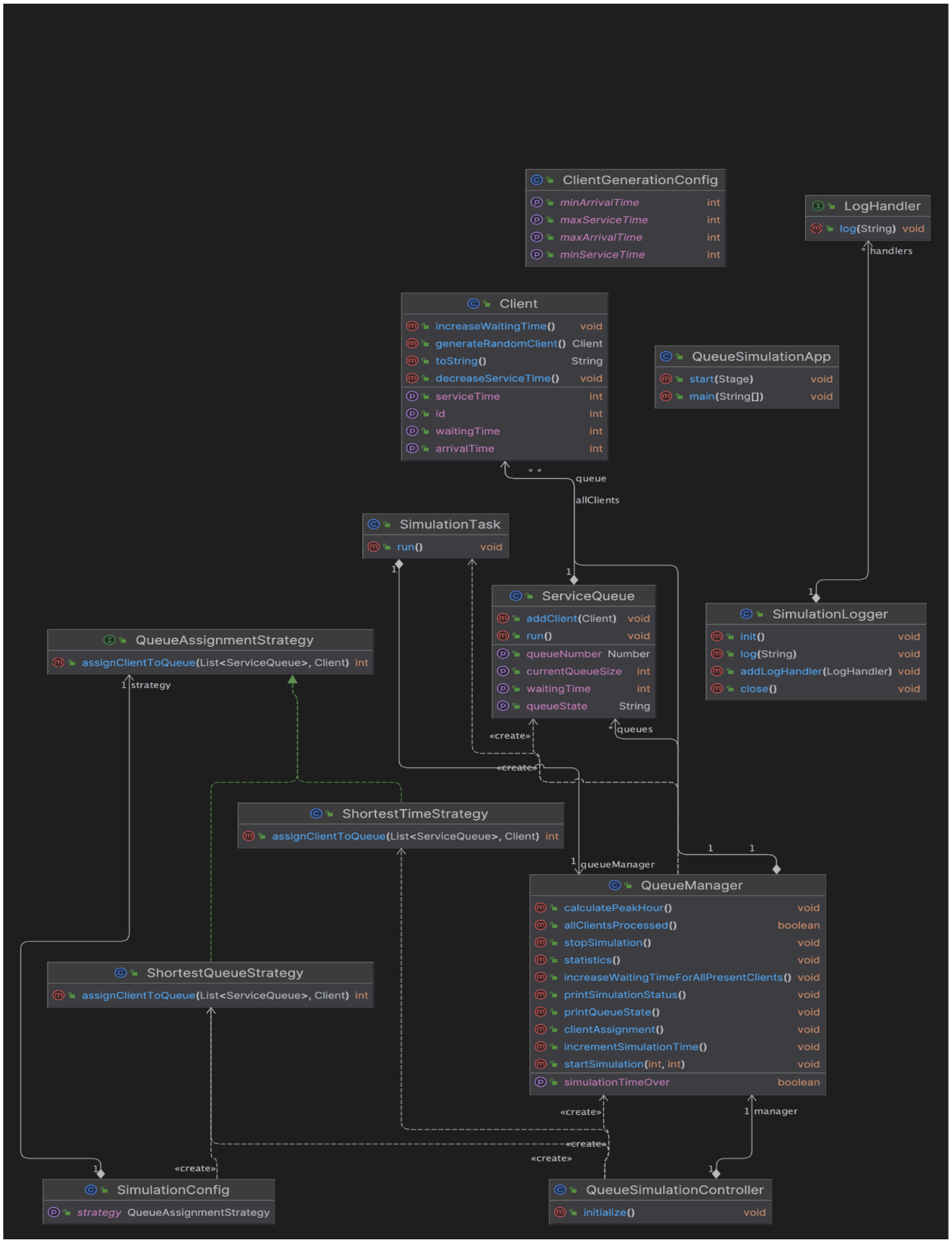
The application leverages a variety of data structures to manage the queueing system effectively. At the core, each ServiceQueue acts as a FIFO (First-In-First-Out) structure, ideal for ensuring that clients are served in the order they arrive.

For managing the overall set of queues and clients, the QueueManager maintains a list or array of ServiceQueue objects and a collection of Client objects, in an ArrayList, which provides the flexibility to grow dynamically as clients are generated.

In the case of the ShortestQueueStrategy, a very simple algorithm, where each queue is sorted in the stream and we choose the shortest one based on its length.

The ShortestTimeStrategy involves a similar not too complex algorithm, sorting the queues using streams based on the expected service time.

Class Diagram



4. Implementation

The implementation of the Queue Management Simulator involves multithreading and synchronization as each queue operates on a separate thread. This concurrency allows for the simultaneous processing of clients in different queues.

Challenges Encountered and Solved:

- **Concurrency Control**: Managing access to shared resources without causing data inconsistency was crucial. Utilizing Java synchronization primitives ensured that thread interference and memory consistency errors were avoided.
- **Performance**: Balancing the simulation's realism with performance. Efficient data structures and thread management were used to optimize the simulation run-time.
- **Usability**: Ensuring that the user interface remains responsive while the simulation is running. This was achieved through JavaFX's threading model, which separates UI updates from the simulation logic.

Threads and Synchronization:

Given that each ServiceQueue runs in a separate thread, synchronization is crucial to prevent concurrent modifications. Java's synchronized keyword and locking mechanisms (ReentrantLock) are used to manage access to shared resources like client lists. This ensures thread safety when multiple threads try to update shared data.

Additionally, wait() mechanisms are employed to efficiently handle the scenario where queues are empty or when a new client arrives. These methods help in reducing CPU usage by avoiding busy-waiting.

Implementing concurrency introduces several challenges, such as deadlocks and thread starvation, which are mitigated through careful design and by adhering to well-established concurrency patterns.

Asynchronous (Async) Structure:

To simulate the real-world scenario where clients are being serviced concurrently, the application adopts an asynchronous structure. Each `ServiceQueue` runs within its thread, allowing for parallel processing of clients. The `QueueManager` oversees these threads, ensuring they are appropriately synchronized to prevent race conditions and data corruption.

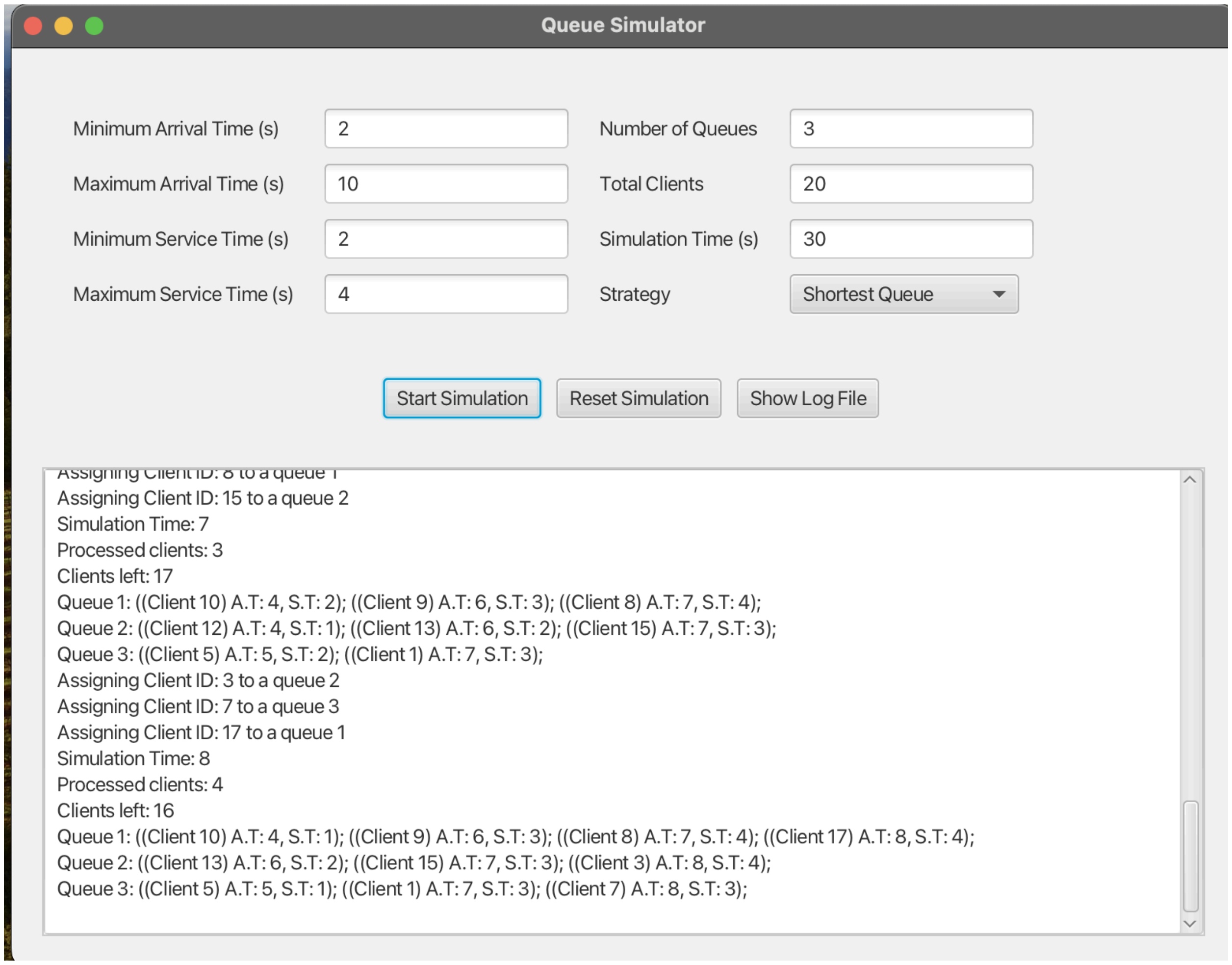
The `SimulationTask`, which utilizes the Singleton design pattern to ensure only one instance of the simulation runs at a time, coordinates the execution of each `ServiceQueue` thread. **The Singleton**

pattern is essential here to prevent multiple instances of the simulation from being instantiated, which would lead to inconsistent states within the simulation.

The async structure allows the application to mimic a real-world queue system where multiple service points operate independently, yet under a coordinated management system.

GUI Implementation:

The QueueManagementController utilizes JavaFX components like text fields, buttons, text labels, and log boxes to create the user interface for entering the simulation configs, selecting the strategy, and displaying the simulation results.



Minimum Arrival Time (s)	2	Number of Queues	3
Maximum Arrival Time (s)	10	Total Clients	20
Minimum Service Time (s)	2	Simulation Time (s)	30
Maximum Service Time (s)	4	Strategy	Shortest Queue

```
Assigning Client ID: 8 to a queue 1
Assigning Client ID: 15 to a queue 2
Simulation Time: 7
Processed clients: 3
Clients left: 17
Queue 1: ((Client 10) A.T: 4, S.T: 2); ((Client 9) A.T: 6, S.T: 3); ((Client 8) A.T: 7, S.T: 4);
Queue 2: ((Client 12) A.T: 4, S.T: 1); ((Client 13) A.T: 6, S.T: 2); ((Client 15) A.T: 7, S.T: 3);
Queue 3: ((Client 5) A.T: 5, S.T: 2); ((Client 1) A.T: 7, S.T: 3);
Assigning Client ID: 3 to a queue 2
Assigning Client ID: 7 to a queue 3
Assigning Client ID: 17 to a queue 1
Simulation Time: 8
Processed clients: 4
Clients left: 16
Queue 1: ((Client 10) A.T: 4, S.T: 1); ((Client 9) A.T: 6, S.T: 3); ((Client 8) A.T: 7, S.T: 4); ((Client 17) A.T: 8, S.T: 4);
Queue 2: ((Client 13) A.T: 6, S.T: 2); ((Client 15) A.T: 7, S.T: 3); ((Client 3) A.T: 8, S.T: 4);
Queue 3: ((Client 5) A.T: 5, S.T: 1); ((Client 1) A.T: 7, S.T: 3); ((Client 7) A.T: 8, S.T: 3);
```

5. Results

The simulation generates logs (`test1_log.txt`, `test2_log.txt`, and `test3_log.txt`) which capture the sequence of events, client assignments, and service completion. These logs represent different scenarios and strategies, providing insights into the efficiency of the queue management strategies employed.

Test Log Interpretation:

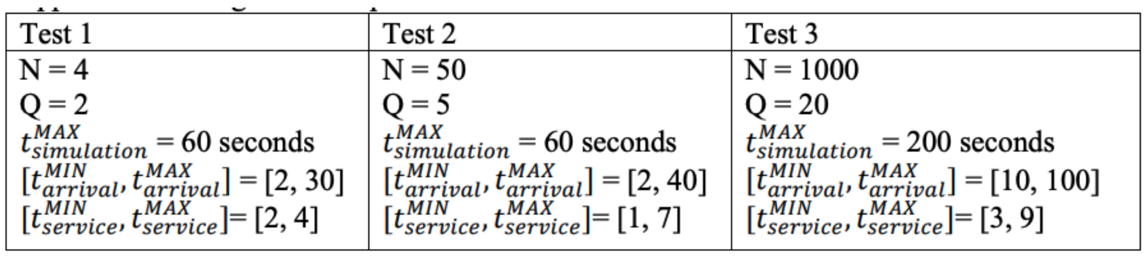
- `test1_log.txt`: Represents a simulation with a 'Shortest Time' strategy, showing the time each client is processed and the system's overall performance.
- `test2_log.txt`: Captures a more complex simulation with a larger number of clients and queues, detailing each client's assignment, service time, and waiting time.
- `test3_log.txt`: This log could represent another unique scenario, potentially using a different strategy or varying the number of clients and queues to test under different load conditions.

The application provides a robust logging system that captures detailed information about the simulation process. Through the analysis of the logs from `test1_log.txt`, `test2_log.txt`, and `test3_log.txt`, valuable data metrics are extracted such as average waiting time, service time, and identification of peak hours.

The logs also serve as a testament to the application's concurrency model, showcasing real-time task handling and client processing across different queues. For instance, in `test2_log.txt`, we observe simultaneous client assignments to various queues, an essential feature of the simulation's multithreaded environment.

Furthermore, by examining the average waiting and service times and comparing them across different strategies, stakeholders can make informed decisions about which strategy might be best suited for particular scenarios, thereby optimizing queue management.

The logs are present in the log directory from the root of the project on GitLab.
The tests were generated to match the table below:



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]

Future Improvements

- **Dynamic Strategy Switching**: Allowing the simulation to change strategies mid-way based on certain triggers or performance metrics.
- **User Experience Enhancements**: Implementation of real-time analytics dashboards within the GUI to provide a visual representation of the simulation data.
- **Distributed Simulation**: Expanding the application to support distributed simulations across multiple machines to simulate large-scale queuing systems like those used by airlines or banking systems.

6. Conclusions

The Queue Management Simulator project demonstrates the application of Java and object-oriented principles in creating a functional simulation. It showcases the integration of JavaFX for UI, Java threading for processes, and data structures for efficient queue management. Through this project, we learned practical approaches to solving concurrency challenges and the importance of responsive UI design. Future enhancements focus on adaptability and scalability. This project has been a valuable learning tool, laying the groundwork for more complex systems and reflecting the diligent efforts of its development.

7. Bibliography

For understanding of the concepts utilized within the application, the following resources were consulted during the development process:

a. Oracle Java Documentation: For detailed insights into Java synchronization and concurrency control mechanisms, Oracle's official Java documentation is an authoritative resource. URL: https://docs.oracle.com/javase/tutorial/essential/concurrency/

b. Baeldung on Java Queues: An extensive guide on Java Queue implementations and management. URL: https://www.baeldung.com/java-queue

c. Load Balancing Architecture: To learn about the architecture of load balancers, which is somewhat analogous to queue management in software systems, AWS provides comprehensive resources. URL: https://aws.amazon.com/elasticloadbalancing/

These resources offer foundational knowledge as well as advanced concepts that were applied to enhance the Queue Management Simulator and similar applications.