DOCUMENTATION

ASSIGNMENT *ASSIGNMENT_NUMBER_2*

STUDENT NAME: Feier Catalin Vasile
GROUP: 30424-1

CONTENTS

1. Assignment Objective 3
2. Problem Analysis, Modeling, Scenarios, Use Cases 3
3. Design 5
4. Implementation 7
5. Results 12
6. Conclusions 12
7. Bibliography 13

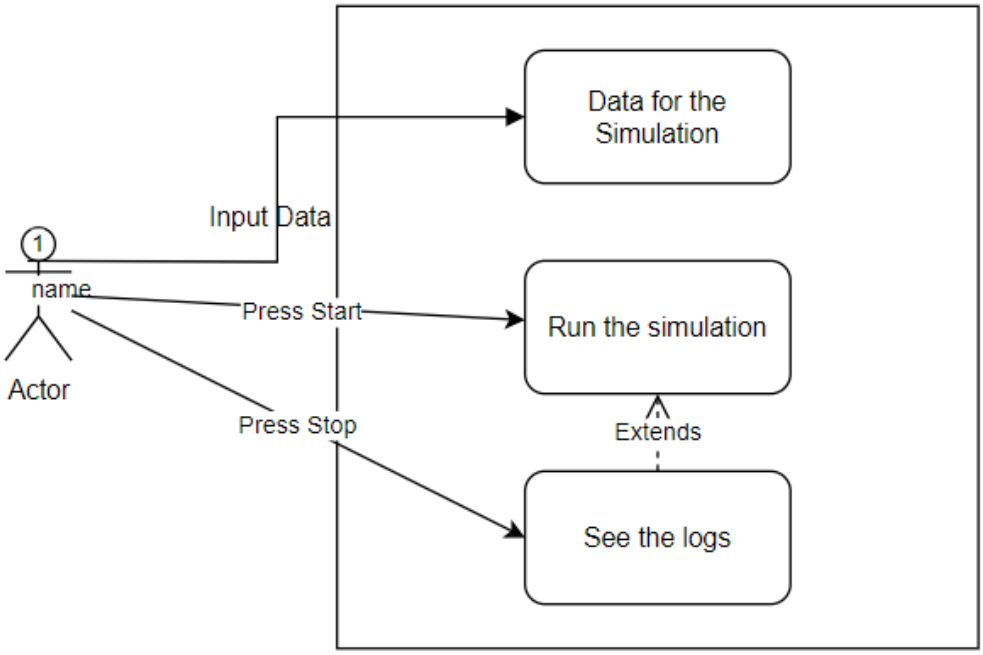
1. Assignment Objective

This project's overarching aim is to develop a sophisticated queue management application capable of emulating client service within a multi-queue framework. Through the implementation of diverse scheduling strategies, the goal is to enhance queue efficiency, whether by minimizing queue congestion or streamlining client waiting times. The comprehensive objectives encompass thorough analysis of the problem domain, identification of essential requirements, meticulous simulator design, robust implementation, and rigorous testing protocols to ensure reliability and effectiveness.

2. Problem Analysis, Modeling, Scenarios, Use Cases

The functional requirements should be presented together with the use cases (use case diagrams and use case description). The use cases' descriptions can be done as a flow-chart or as a list containing the execution steps of each use case.

USE CASE-> Diagram



USE CASE -> Description:

Use Case: Data For The Simulation/Run The Simulation
Primary Actor: User
Success Scenario Steps:

I. The users inputs the following data:

a. Name of the file where to save the logs.
b. Number of clients,
c. Number of queues,
d. Simulation interval,
e. Minimum and Maximum arrival time,
f. Minimum and Maximum service time,
g. Selects between the 2 policies: SHORTEST_QUEUE or SHORTEST_TIME

II. The user presses start
III. The simulation runs
IV. The simulation is finished
V. The user presses Stop
VI. The user can see the results on the screen/in the logs file.

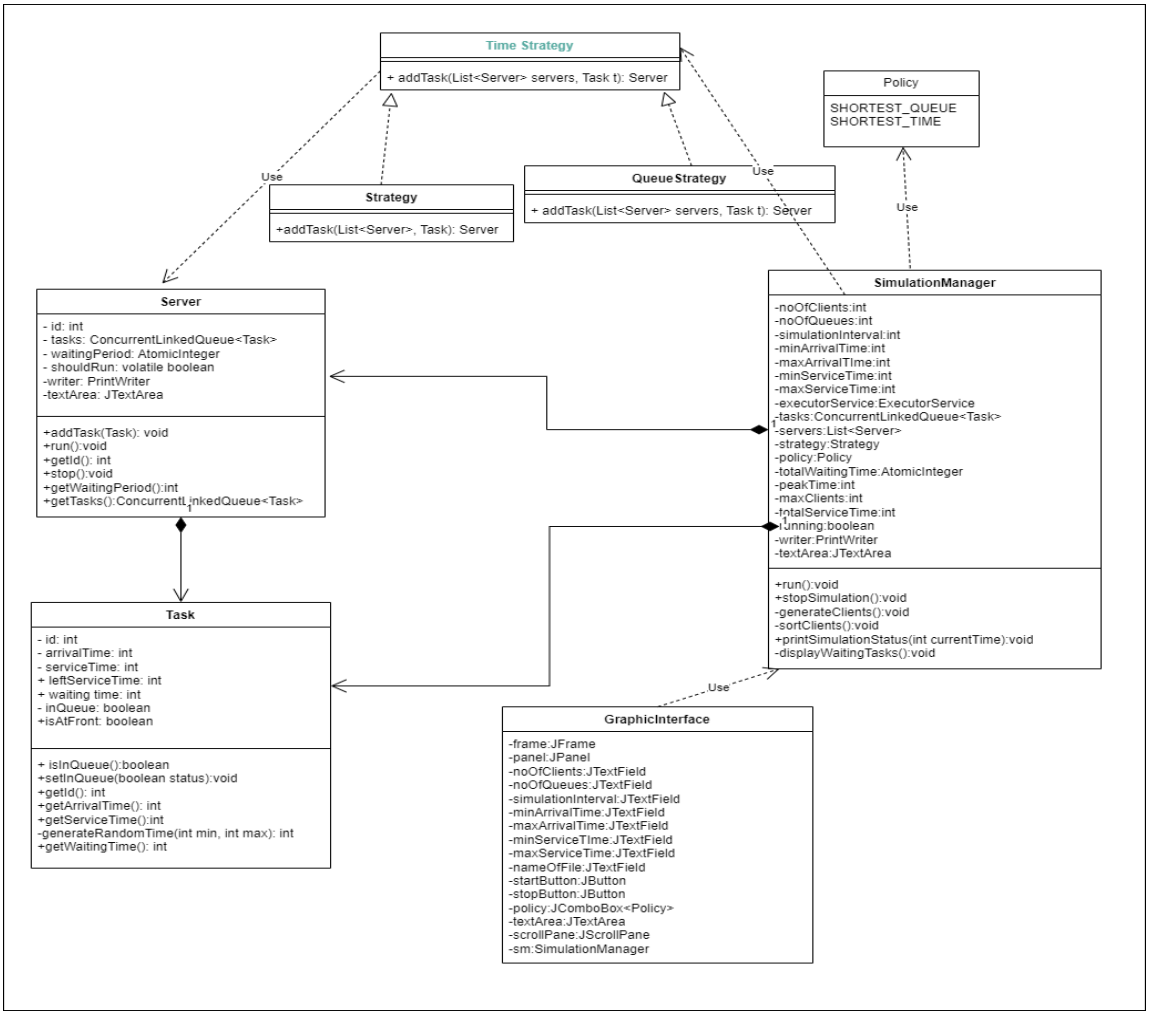
Alternative Sequences:

a) Invalid input data:
- The App displays the message “Invalid Input”
- Go to step I.

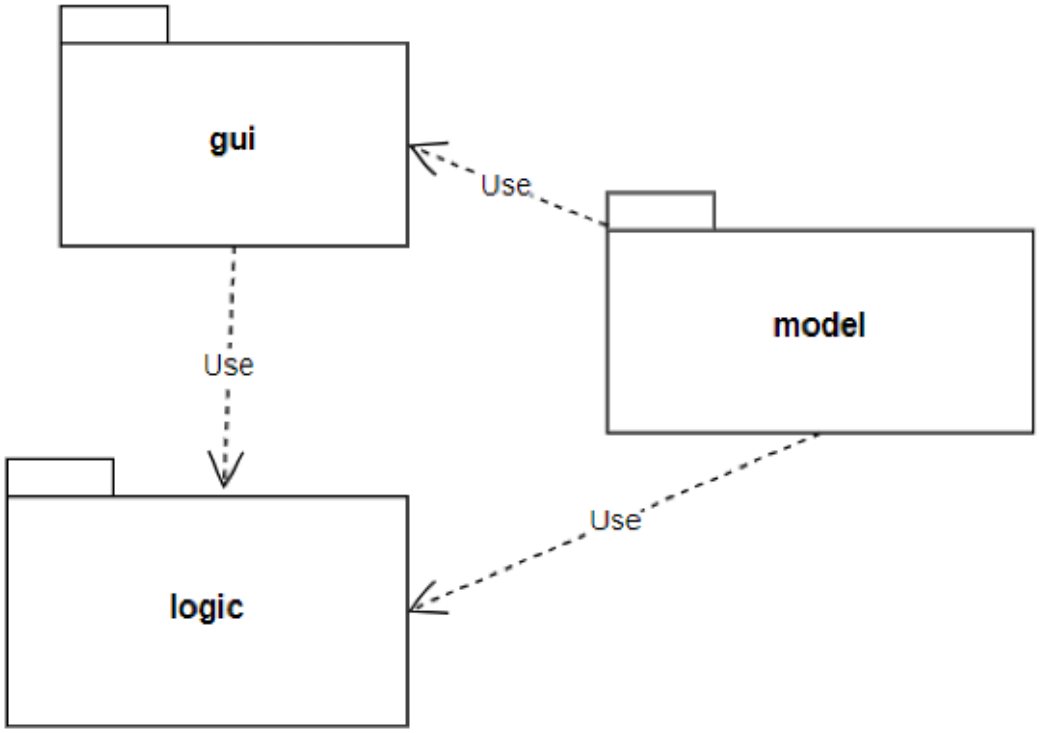
3. Design

The object-oriented design of the application follows fundamental principles to create a robust and modular system for simulating multi-queue management. Abstraction is employed to model real-world entities, with classes like Task encapsulating attributes such as ID, arrival time, and service time. Encapsulation ensures that the internal state of objects is protected and accessed only through well-defined interfaces, exemplified by classes like Server, which encapsulates the task queue and waiting period. Composition is utilized to create complex objects by combining simpler components, as seen in the SimulationManager class, which contains and manages instances of other classes like Server and Task.
Inheritance, while not explicitly demonstrated, could be leveraged to promote code reuse and hierarchy if common characteristics exist among different types of tasks or servers. Additionally, polymorphism enables different scheduling strategies, represented by the Strategy interface, to be used interchangeably within the SimulationManager. Overall, the OOP design of the application adheres to these principles to create a maintainable and extensible system, providing a solid foundation for simulating multi-queue scenarios efficiently and effectively.

CLASS DIAGRAM:



PACKAGE DIAGRAM:



USED DATA STRUCTURES:

ConcurrentLinkedQueue: This data structure is employed to store tasks in the tasks queue within the SimulationManager class. The use of ConcurrentLinkedQueue ensures thread-safety, making it suitable for concurrent access by multiple servers during the simulation. It provides high throughput for adding and removing elements, which is essential for handling tasks arriving and departing from the system in real-time.
ArrayList: An ArrayList is used to store server instances within the SimulationManager class. Unlike linked lists, ArrayLists provide constant-time access to elements, making them suitable for quickly retrieving servers during the simulation. Since the number of servers is fixed and known in advance, ArrayList's dynamic resizing capability is not necessary, and its simplicity and fast random access make it a suitable choice.
AtomicInteger: AtomicInteger is employed in SimulationManager to track the total waiting time of tasks in the system. It ensures atomicity in operations such as incrementing and decrementing, making it safe for use in concurrent environments. AtomicInteger avoids the need for explicit synchronization mechanisms, which simplifies the code and reduces the risk of deadlocks or race conditions.

4. Implementation

Each class will be described (fields, important methods). Also, the implementation of the graphical user interface will be described.

I. The Strategy interface defines a contract for classes that implement different strategies for adding tasks to servers in the queue management system. It contains a single method:
   - ➔ addTask(List<Server> servers, Task t): This method takes a list of servers and a task as parameters and returns a Server object.

II. The TimeStrategy class implements the Strategy interface, providing a strategy for adding tasks to servers based on the server's waiting time.
   - ➔ addTask(List<Server> servers, Task t): This method overrides the addTask method defined in the Strategy interface. It takes a list of servers (servers) and a task (t) as parameters. The method iterates through the list of servers to find the server with the minimum waiting period (waitingPeriod). It then adds the task to this server's queue using the addTask method of the Server class. Finally, it returns the server to which the task was added.

III. The QueueStrategy class is another implementation of the Strategy interface, providing a different strategy for adding tasks to servers based on the size of their queues.
   - ➔ addTask(List<Server> servers, Task t): This method overrides the addTask method defined in the Strategy interface. It takes a list of servers (servers) and a task (t) as parameters. The method initializes minQueueServer to the first server in the list. It then iterates through the list of servers to find the server with the smallest queue size (tasks). Once found, it adds the task to this server's queue using the addTask method of the Server class. Finally, it returns the server to which the task was added.

IV. The Policy enum defines two policy options for task assignment strategies within the queue management system:
   - ➔ SHORTEST_QUEUE: represents the policy for selecting the server with the shortest queue to add a task.
   - ➔ SHORTEST_TIME: represents the policy for selecting the server with the shortest waiting time to add a task.

V. The Task class represents a task or job in a queue management system. Each instance of this class encapsulates information about a specific task, such as its unique identifier (id), the time at which it arrives (arrivalTime), and the duration of service required (serviceTime). These attributes are essential for managing and processing tasks efficiently within the system.
   a. Fields:
      i. id: An integer representing the unique identifier of the task.

      ii. arrivalTime: An integer indicating the time at which the task arrives in the system.

iii. serviceTime: An integer representing the time required to service (process) the task.

iv. leftServiceTime: An integer indicating the remaining service time for the task.

v. waitingTime: An integer representing the amount of time the task has spent waiting in the queue.

vi. inQueue: A boolean flag indicating whether the task is currently in the queue.

vii. isAtFront: A boolean flag indicating whether the task is at the front of the queue.

b. Methods:

i. isInQueue(): Returns a boolean indicating whether the task is in the queue (true if in queue, false otherwise).

ii. setInQueue(boolean inQueue): Sets the inQueue field based on the provided boolean value.

iii. getId(): Returns the unique identifier of the task.

iv. generateRandomTime(int min, int max): A helper method that generates a random integer within the specified range [min, max].

v. getArrivalTime(): Returns the arrival time of the task.

vi. getServiceTime(): Returns the service time required for the task.

vii. getWaitingTime(): Returns the waiting time of the task.

VI. The Server class represents a server or processing unit in the queue management system. The class provides methods to add tasks to the server's queue (addTask(Task t)), process tasks asynchronously in its run() method, and print the current state of the queue (printTasks()) to the console and a designated output file.

a. Fields:

i. private int id;: Represents the unique identifier of the server.

ii. private final ConcurrentLinkedQueue<Task> tasks;: Represents the queue of tasks that need to be processed by the server. It's a concurrent linked queue, ensuring thread safety.

iii. private AtomicInteger waitingPeriod;: Tracks the cumulative waiting time of tasks in the server's queue. It's an AtomicInteger, providing atomic operations for thread safety.

iv. private volatile boolean shouldRun;: Controls the execution of the server's processing loop. Marked as volatile for thread safety.

v. private PrintWriter writer;: Used for writing output to a file.

vi. private JTextArea textArea;: Used for displaying real-time information in a GUI.

b. Constructor: public Server(int id, PrintWriter writer, JTextArea textArea): Initializes a new server instance with the provided ID, writer, and text area.

c. Methods:

i. public void addTask(Task t): Adds a task to the server's queue, updating the waiting time accordingly.

ii. public void run(): Implements the server's processing logic, where tasks are dequeued and processed in a loop.

iii. public void printTasks(): Prints the tasks in the server's queue to the console and the associated text area in the GUI.

iv. public int getId(): Returns the ID of the server.

v. public void stop(): Stops the server by setting the shouldRun flag to false and interrupting the current thread.

vi. public int getWaitingPeriod(): Returns the current waiting period of the server.

vii. public ConcurrentLinkedQueue<Task> getTasks(): Returns the queue of tasks handled by the server.

VII. The SimulationManager class orchestrates the simulation of a queueing system. It manages the generation and processing of clients, the assignment of tasks to server queues based on different policies, and tracks various metrics such as waiting time, peak time, and service time. Additionally, it provides methods to start, stop, and display the status of the simulation. Overall, it serves as the core component responsible for simulating and analyzing the behavior of the queueing system.

a. Fields:

i. private int noOfClients;: Represents the total number of clients in the simulation.

ii. private int noOfQueues;: Represents the total number of server queues in the simulation.

iii. private int simulationInterval;: Represents the duration of the simulation.

iv. private int minArrivalTime;: Represents the minimum arrival time for clients.

v. private int maxArrivalTime;: Represents the maximum arrival time for clients.

vi. private int minServiceTime;: Represents the minimum service time for clients.

vii. private int maxServiceTime;: Represents the maximum service time for clients.

viii. private ExecutorService executorService;: Manages the threads for server processing.

ix. private ConcurrentLinkedQueue<Task> tasks;: Represents the queue of tasks to be processed.

x. private List<Server> servers;: Stores the server instances.

xi. private Strategy strategy;: Represents the strategy used to assign tasks to servers.

xii. private Policy policy;: Represents the policy used for task assignment.

xiii. private AtomicInteger totalWaitingTime;: Tracks the total waiting time of clients.

xiv. private int peakTime;: Stores the peak time during the simulation.

xv. private int maxClients;: Stores the maximum number of clients during the simulation.

xvi. private int totalServiceTime;: Tracks the total service time of clients.

xvii. private volatile boolean running;: Controls the execution of the simulation loop.

xviii. private PrintWriter writer;: Used for writing output to a file.

xix. private JTextArea textArea;: Used for displaying real-time information in a GUI.

b. Constructor: public SimulationManager(...): Initializes a new simulation manager instance with the specified parameters.
c. Methods:
    i. public void run(): Implements the main logic of the simulation.

ii. public void stopSimulation(): Stops the simulation and cleans up resources.

iii. public void closeWriter(): Closes the writer used for file output.

iv. private void generateClients(): Generates clients with random arrival and service times.

v. private void sortClients(): Sorts the clients based on arrival time.

vi. public void printSimulationStatus(int currentTime): Prints the status of the simulation at the current time.

vii. private void displayWaitingTasks(): Displays the waiting tasks in the simulation.

VIII. The GraphicInterface class provides a graphical user interface for configuring and controlling the queue simulation. It allows users to input parameters such as the number of clients, number of queues, simulation interval, arrival time range, service time range, and policy. Users can start and stop the simulation, and the GUI displays real-time information about the simulation progress. Additionally, it enables users to specify an output file for saving simulation results. Overall, the class serves as an interface between the user and the simulation manager, facilitating interaction and visualization of simulation data.

a. Fields:
   i. private JFrame frame;: Represents the main frame of the GUI.

   ii. private JPanel panel;: Represents the panel containing input fields and buttons.

   iii. private JTextField, private JButton, private JComboBox, private JTextArea: Represent various input components such as text fields, buttons, and combo boxes.

   iv. private JScrollPane scrollPane;: Represents a scroll pane for displaying text area content.

   v. private SimulationManager sm;: Manages the simulation logic.

   vi. private AtomicReference<PrintWriter> writer;: Atomic reference to a PrintWriter for writing output to a file.

   vii. private AtomicReference<Boolean> started, stopped;: Atomic references for tracking simulation state.
b. Constructor: public GraphicInterface(): Initializes the GUI components and event listeners.
c. Methods:

i. startButton.addActionListener(e -> { ... });: Defines action when the start button is clicked. Parses input values, validates them, starts the simulation, and updates GUI components accordingly.

ii. stopButton.addActionListener(e -> { ... });: Defines action when the stop button is clicked. Stops the simulation and enables input fields for modification.

5. Results

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

Average waiting time: 0.0
Peak time: 24 with 2 clients
Average service time: 2.5

Test 2
$N = 50$ $Q = 5$ $t_{simulation}^{MAX} = 60$ seconds $[t_{arrival}^{MIN}, t_{arrival}^{MAX}] = [2, 40]$ $[t_{service}^{MIN}, t_{service}^{MAX}] = [1, 7]$

Average waiting time: 1.98

Peak time: 20 with 8 clients

Average service time: 3.8

Test 3
$N = 1000$ $Q = 20$ $t_{simulation}^{MAX} = 200$ seconds $[t_{arrival}^{MIN}, t_{arrival}^{MAX}] = [10, 100]$ $[t_{service}^{MIN}, t_{service}^{MAX}] = [3, 9]$

To be noted that in this case where the simulation time runs out before the tasks can be finished.

Average waiting time: 103.681

Peak time: 100 with 700 clients

Average service time: 6.032

6. Conclusions

This assignment provided valuable insights into developing a queue simulation application using Java. Through the implementation process, several key concepts and techniques were reinforced, including object-oriented design, multithreading, graphical user interface (GUI) development, and file handling. By simulating a queue system with various policies and parameters, I gained a deeper understanding of how different factors impact system performance and efficiency.

7. Bibliography

https://www.baeldung.com/java-concurrency
https://www.baeldung.com/java-wait-and-sleep
https://www.baeldung.com/java-util-concurrent
https://www.baeldung.com/java-threadpooltaskexecutor-core-vs-max-poolsize
https://www.baeldung.com/java-thread-stop
https://www.baeldung.com/java-future