DOCUMENTATION

ASSIGNMENT *2*

STUDENT NAME: Duică Sebastian
GROUP: 30424

CONTENT

1. Assignment Objective

Main Objective: To design and implement a queues management application in Java that minimizes client waiting time.

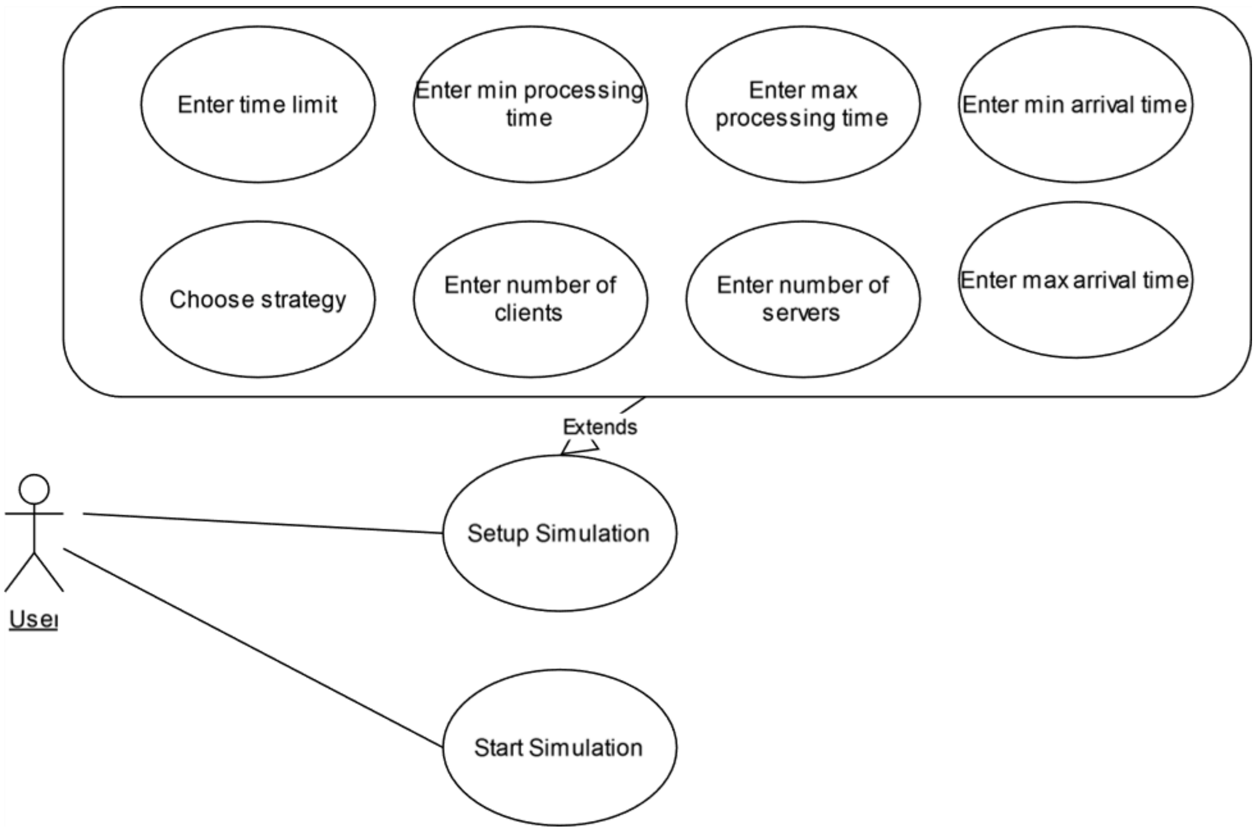
Sub-Objectives:

Sub-Objective	Description	Section
Design classes to represent clients, queues, and the main application logic	Define the structure of the application in an object-oriented manner	Requirements
Implement multithreading to handle multiple clients and queues concurrently	Ensure efficient processing of clients using concurrent threads	Requirements
Implement synchronization mechanisms to ensure thread safety when accessing shared data structures	Prevent race conditions and maintain data integrity in a concurrent environment	Requirements
Simulate client arrival, queueing, service, and departure processes	Create a simulation engine to model the behavior of clients and queues	Requirements
Track total time spent by each client in queues and compute average waiting time	Calculate statistics to evaluate the performance of the queues management system	Requirements
Generate log of events displaying the status of clients and queues over time	Record the progression of the simulation for analysis and debugging	Requirements
Implement strategy pattern for allocating clients to queues based on shortest time or shortest queue	Allow for flexible queue assignment strategies to optimize performance	Evaluation
Develop a graphical user interface for simulation setup and real-time queue evolution display	Provide a user-friendly interface for configuring and monitoring the simulation	Evaluation
Display simulation results including average waiting time and peak hour	Present key performance metrics to evaluate the effectiveness of the queues management system	Evaluation
Test application on various input data sets and include generated logs in documentation	Verify functionality and performance under different scenarios	Evaluation

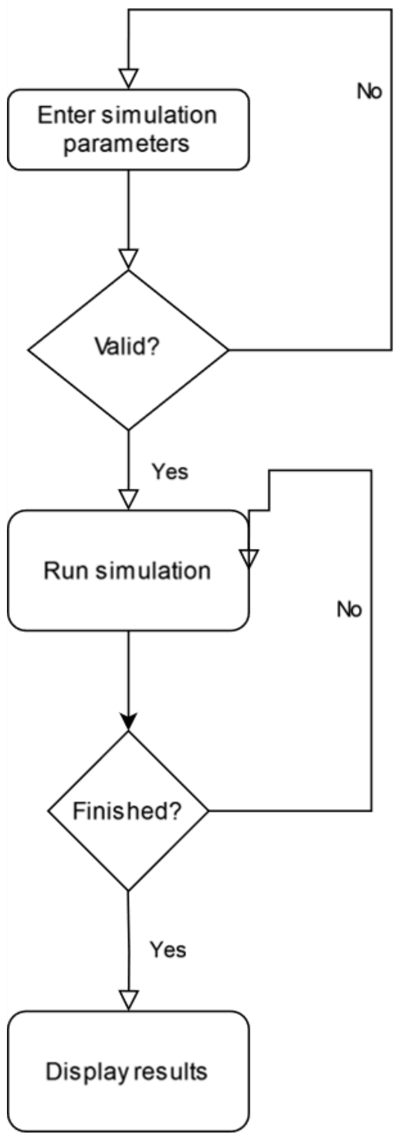
2. These sub-objectives outline the steps necessary to achieve the main objective of designing and implementing a queues management application. Each sub-objective is described along with the section in which it will be addressed.

3. Problem Analysis, Modeling, Scenarios, Use Cases

Use Case Diagram



Flow Chart



Functional Requirements:

- **Client Generation:**
    - Users can input the number of clients, arrival time bounds, and service time bounds.
    - The system generates clients with unique IDs, random arrival times, and random service times within specified bounds.
- **Queue Management:**
    - Users can input the number of queues.
    - The system creates and manages the specified number of queues.
    - Clients are assigned to queues based on their arrival time and the current queue status.
- **Simulation Execution:**
    - Users can input the simulation interval.
    - The system simulates the arrival, queueing, service, and departure processes for the specified interval.
- **Data Tracking:**
    - The system tracks the total time spent by each client in queues.
    - The system computes the average waiting time for all clients.
- **Logging Events:**
    - The system logs events such as client arrivals, queue openings/closings, and client departures.
    - The log is updated periodically to reflect the simulation progress.
- **Queue Assignment Strategies:**
    - Users can choose between two queue assignment strategies: shortest time and shortest queue.
    - The system implements the selected strategy to assign clients to queues.

Use Cases:

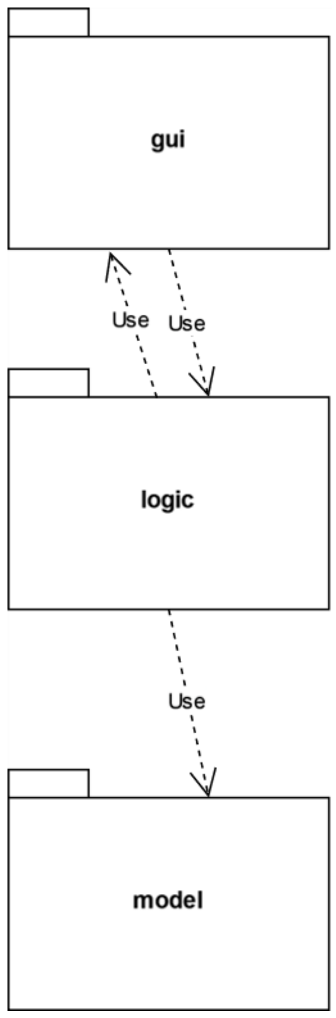
- **Generate Clients:**
    - **Description:** Users input parameters for client generation.
    - **Steps:**
        - User provides the number of clients, arrival time bounds, and service time bounds.
        - System generates clients with random arrival times and service times within specified bounds.
- **Manage Queues:**
    - **Description:** Users input the number of queues.
    - **Steps:**
        - User provides the number of queues.
        - System creates and manages the specified number of queues.
- **Execute Simulation:**
    - **Description:** Users input the simulation interval.
    - **Steps:**
        - User provides the simulation interval.
        - System simulates client arrival, queueing, service, and departure processes for the specified interval.

- **Track Data:**
    - **Description:** System tracks client wait times and computes average waiting time.
    - **Steps:**
        - System tracks the total time spent by each client in queues.
        - System computes the average waiting time for all clients.
- **Log Events:**
    - **Description:** System logs events during simulation.
    - **Steps:**
        - System records client arrivals, queue openings/closings, and client departures.
        - Log is periodically updated to reflect simulation progress.
- **Choose Queue Assignment Strategy:**
    - **Description:** Users select a queue assignment strategy.
    - **Steps:**
        - User chooses between shortest time and shortest queue strategies.
        - System implements the selected strategy to assign clients to queues.

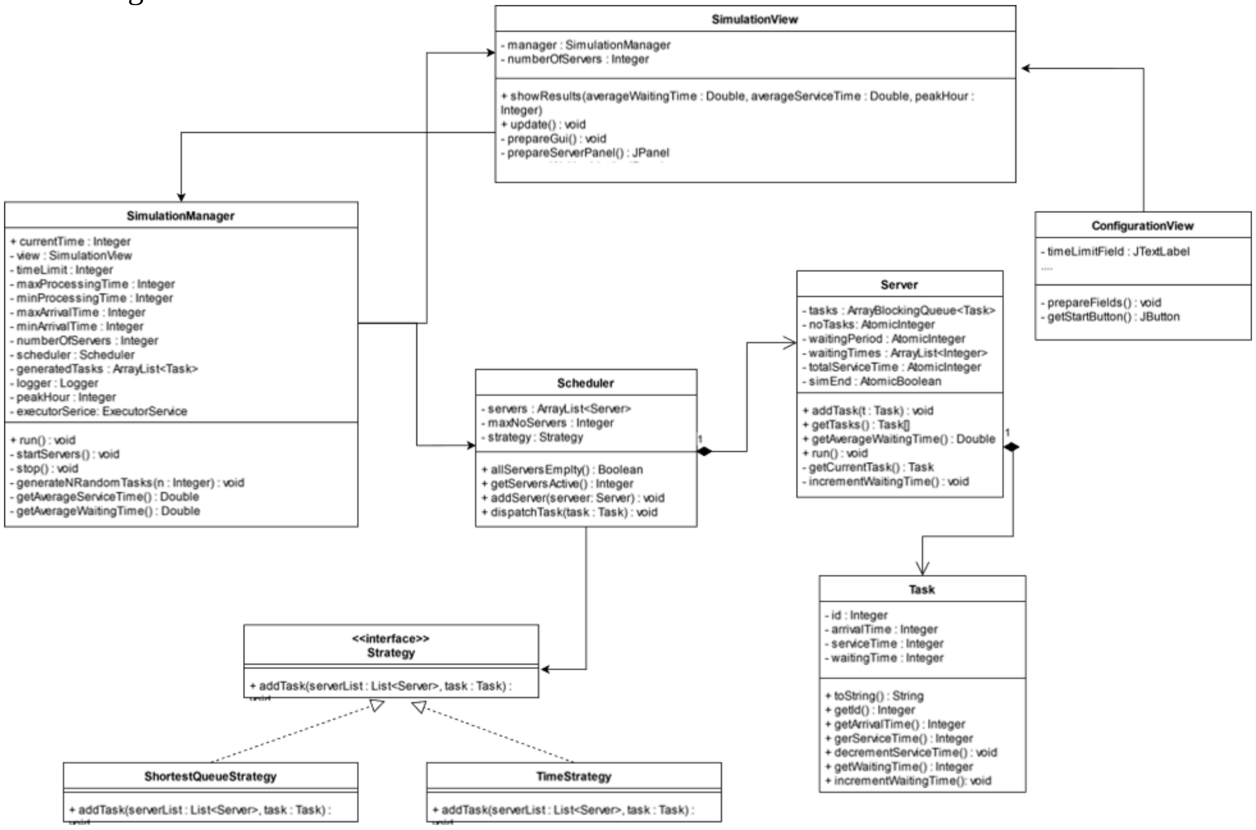
These functional requirements and use cases outline the key features and interactions expected in the queues management application. Use cases are described as execution steps or flowcharts to provide clarity on how users interact with the system.

4. Design

Package Diagram



Class Diagram


SimulationView
- manager : SimulationManager
- numberOfServers : Integer
+ showResults(averageWaitingTime : Double, averageServiceTime : Double, peakHour : Integer)
+ update() : void
- prepareGui() : void
- prepareServerPanel() : JPanel
SimulationManager
+ currentTime : Integer
- view : SimulationView
- timeLimit : Integer
- maxProcessingTime : Integer
- minProcessingTime : Integer
- maxArrivalTime : Integer
- minArrivalTime : Integer
- numberOfServers : Integer
- scheduler : Scheduler
- generatedTasks : ArrayList<Task>
- logger : Logger
- peakHour : Integer
- executorSerice: ExecutorService
+ run() : void
- startServers() : void
- stop() : void
- generateNRandomTasks(n : Integer) : void
- getAverageServiceTime() : Double
- getAverageWaitingTime() : Double
ConfigurationView
- timeLimitField : JTextLabel
....
- prepareFields() : void
- getStartButton() : JButton
Server
- tasks : ArrayBlockingQueue<Task>
- noTasks: AtomicInteger
- waitingPeriod : AtomicInteger
- waitingTimes : ArrayList<Integer>
- totalServiceTime : AtomicInteger
- simEnd : AtomicBoolean
+ addTask(t : Task) : void
+ getTasks() : Task[]
+ getAverageWaitingTime() : Double
+ run() : void
- getCurrentTask() : Task
- incrementWaitingTime() : void
Scheduler
- servers : ArrayList<Server>
- maxNoServers : Integer
- strategy : Strategy
+ allServersEmplty() : Boolean
+ getServersActive() : Integer
+ addServer(serveer: Server) : void
+ dispatchTask(task : Task) : void
1
1
Task
- id : Integer
- arrivalTime : Integer
- serviceTime : Integer
- waitingTime : Integer
+ toString() : String
+ getId() : Integer
+ getArrivalTime() : Integer
+ gerServiceTime() : Integer
+ decrementServiceTime() : void
+ getWaitingTime() : Integer
+ incrementWaitingTime(): void
<<interface>>
Strategy
+ addTask(serverList : List<Server>, task : Task) :
ShortestQueueStrategy
+ addTask(serverList : List<Server>, task : Task) :
TimeStrategy
+ addTask(serverList : List<Server>, task : Task) :

5. **Implementation**

Task Class:

- **Fields:**
  - Id: Represents the unique identifier of the task.
  - arrivalTime: Indicates the arrival time of the task.
  - serviceTime: Represents the service time required for the task.
  - waitingTime: Tracks the waiting time of the task.
- **Important Methods:**
  - Task(int Id, int arrivalTime, int serviceTime): Constructor method for initializing a task with the provided ID, arrival time, and service time.
  - toString(): Overrides the toString() method to provide a string representation of the task.
  - getId(): Retrieves the ID of the task.
  - getArrivalTime(): Retrieves the arrival time of the task.
  - getServiceTime(): Retrieves the service time required for the task.
  - decrementServiceTime(): Decrements the service time of the task.
  - getWaitingTime(): Retrieves the waiting time of the task.
  - incrementWaitingTime(): Increments the waiting time of the task.

Server Class:

- **Fields:**
  - tasks: Represents the queue of tasks assigned to the server.
  - noTasks: Tracks the number of tasks currently assigned to the server.
  - waitingPeriod: Indicates the total waiting period for the tasks in the server.
  - waitingTimes: Stores the waiting times of tasks in the server.
  - totalServiceTime: Tracks the total service time provided by the server.
  - simEnd: Indicates whether the simulation has ended.
- **Important Methods:**
  - Server(int maxNoTasks): Constructor method for initializing a server with the maximum number of tasks it can handle.
  - addTask(Task t): Adds a task to the server's queue and updates relevant metrics.
  - getTasks(): Retrieves an array of tasks currently assigned to the server.
  - getWaitingPeriod(): Retrieves the total waiting period for tasks in the server.
  - getNoTasks(): Retrieves the number of tasks currently assigned to the server.
  - getAverageWaitingTime(): Calculates and returns the average waiting time of tasks in the server.
  - getServiceTime(): Retrieves the total service time provided by the server.
  - run(): Implements the server's behavior during the simulation.
  - getCurrentTask(): Retrieves the current task being processed by the server.
  - incrementWaitingTime(): Increments the waiting time of tasks in the server.
  - setSimEnd(boolean simEnd): Sets the status of the simulation end.

SimulationManager Class:

- **Fields:**
  - currentTime: Tracks the current time in the simulation.
  - view: Reference to the simulation view for updating the GUI.
  - timeLimit: Indicates the time limit for the simulation.
  - maxProcessingTime, minProcessingTime: Define the range of processing time for tasks.
  - maxArrivalTime, minArrivalTime: Define the range of arrival time for tasks.
  - numberOfServers: Indicates the number of servers in the simulation.
  - scheduler: Manages the scheduling of tasks among servers.
  - generatedTasks: Stores the list of generated tasks for the simulation.
  - logger: Handles logging simulation events and metrics.
  - peakHour: Tracks the peak hour during the simulation.
  - executorService: Manages the execution of server tasks concurrently.
- **Important Methods:**
  - SimulationManager(SimulationView view, int timeLimit, int maxProcessingTime, int minProcessingTime, int maxArrivalTime, int minArrivalTime, int numberOfServers, int numberOfClients, StrategyPolicy strategyPolicy): Constructor method for initializing the simulation manager with parameters for simulation setup.
  - getGeneratedTasks(): Retrieves the list of generated tasks for the simulation.
  - getServers(): Retrieves the list of servers used in the simulation.
  - run(): Implements the main logic of the simulation.
  - showResults(double averageWaitingTime, double averageServiceTime, int peakHour): Displays the simulation results in the GUI.
  - update(): Updates the simulation view to reflect the current state.
  - Other helper methods for managing the simulation flow and tasks.

ConfigurationView Class:

- **Fields:**
  - Fields for various configuration parameters such as time limit, processing time, arrival time, number of servers, number of clients, and strategy selection.
- **Important Methods:**
  - ConfigurationView(): Constructor method for initializing the configuration view.
  - prepareFields(): Sets up the GUI components for configuring simulation parameters.
  - getStartButton(): Creates and configures the "Start" button for initiating the simulation.
  - Other methods for handling user input and interaction with the GUI.

The graphical user interface (GUI) for the simulation is implemented using Swing components such as JFrame, JPanel, JLabel, JTextField, JButton, and JList. The GUI elements are organized to provide a user-friendly interface for configuring simulation parameters and visualizing the simulation process, including real-time updates of server states and task queues.

6. Conclusions

- The assignment provided valuable insights into the design and implementation of a queue management system using object-oriented programming principles.
- Through the development process, several key concepts such as task scheduling strategies, server management, and GUI design were explored and applied.
- The completion of the assignment highlights the importance of systematic problem-solving and the effective utilization of software development methodologies.

What I Have Learned from the Assignment:

- **Object-Oriented Design:** I gained a deeper understanding of object-oriented design principles, including encapsulation, inheritance, and polymorphism, by applying them to real-world problem-solving.
- **Concurrency and Multithreading:** The assignment allowed me to enhance my knowledge of concurrency and multithreading concepts through the implementation of concurrent server tasks and simulation management.
- **GUI Development:** I acquired practical experience in GUI development using Java Swing, learning how to create interactive and user-friendly interfaces for software applications.
- **Algorithmic Thinking:** Developing task scheduling algorithms and strategies challenged my algorithmic thinking skills, fostering a mindset for optimizing system performance and resource utilization.

Future Developments:

- **Enhanced Functionality:** Future developments could focus on enhancing the functionality of the queue management system by incorporating additional features such as task prioritization, dynamic server allocation, and real-time analytics.
- **User Experience Improvements:** Further improvements to the GUI could be made to enhance the user experience, including implementing more intuitive controls, visualizations, and customization options.
- **Performance Optimization:** Continual optimization of the system's performance and efficiency through algorithm refinement, code optimization, and utilization of advanced data structures and algorithms.
- **Integration and Scalability:** Exploring opportunities for integrating the queue management system with other systems or platforms and ensuring scalability to accommodate growing user demands and system requirements.

- **Testing and Validation:** Rigorous testing and validation procedures should be implemented to ensure the reliability, robustness, and correctness of the system under various scenarios and edge cases

7. Bibliography

https://dsrl.eu/courses/pt/