DOCUMENTATION

ASSIGNMENT *2*

STUDENT NAME: Tcaci Liviu
GROUP: 30221

CONTENTS

1. Assignment Objective ..... **Error! Bookmark not defined.**
2. Problem Analysis, Modeling, Scenarios, Use Cases ..... **Error! Bookmark not defined.**
3. Design ..... **Error! Bookmark not defined.**
4. Implementation..... **Error! Bookmark not defined.**
5. Results ..... **Error! Bookmark not defined.**
6. Conclusions ..... **Error! Bookmark not defined.**
7. Bibliography..... **Error! Bookmark not defined.**

1. Assignment Objective

Main Objective:

Design and implement a queue management system using threads and synchronization mechanisms to minimize client waiting times in a simulated environment.

This project aims to model a real-world queue management scenario where multiple clients arrive and are served by different queues. The system's effectiveness is measured by how well it minimizes the waiting time of each client, thus improving overall efficiency and client satisfaction.

Sub-Objectives:

The successful completion of the main objective requires the fulfillment of several sub-objectives, each addressing different aspects of the software development lifecycle:

Sub-Objective	Description	Section Addressed
Analyze the Problem	Understand and define the problem the software needs to solve, including the identification of key requirements for the queuing system.	Section 2: Problem Analysis
Design the System	Develop a detailed software architecture using object-oriented principles, including class diagrams and the identification of appropriate design patterns and data structures	Section 3: Design
Implement the System	Code the application based on the design, ensuring all functionalities are properly implemented and adhere to good coding practices.	Section 4: Implementation
Test the System	Execute a series of tests to ensure the system meets the requirements and performs as expected under various conditions.	Section 5: Results

2. Problem Analysis, Modeling, Scenarios, Use Cases

Problem Analysis

The primary challenge addressed by this project is the inefficient management of queues, which leads to increased waiting times for clients and suboptimal utilization of server resources. This scenario is common in various real-world applications like customer service, hospital management systems, and public service institutions.

Functional Requirements:

- Client Processing: The system must handle the dynamic arrival of clients, assigning them to the queue with the least waiting time.
- Queue Management: Support multiple queues with the capability to process clients independently and concurrently.
- Simulation Control: Ability to set the simulation parameters, including the number of clients, number of queues, and maximum simulation time.
- Concurrency Handling: Ensure thread safety when accessing and modifying shared resources across multiple threads.
- Logging and Monitoring: Maintain a detailed log of all system events, including client arrivals, service commencements, and completions, and queue status updates.

Non-Functional Requirements:

- Performance: The system should efficiently manage a large number of clients and queues without significant delays or performance bottlenecks.
- Scalability: The architecture should support scaling in terms of increasing the number of clients or queues without major changes.
- Reliability: The system must handle errors gracefully, providing robustness against potential failure scenarios like invalid input data.
- Usability: Though primarily a console-based simulation, any user interfaces should be straightforward, allowing easy interaction and understanding.

Modeling and Use Cases

To clearly define how the system interacts with users and other systems, use case diagrams and detailed descriptions are utilized.

Use Case Diagram:

Actors:

User: I nteracts with the system to input parameters and start the simulation.

Use Cases:

Setup Simulation: Users configure the initial parameters of the simulation.

Run Simulation: Users initiate the simulation process, which then runs autonomously.

Use Case Descriptions:

1. Setup Simulation

   Actor: User

   Preconditions: The application is loaded and ready to accept input.

   Main Flow:

   - The user inputs the number of clients, number of queues, simulation interval, and bounds for arrival and service times.
   - The system validates the provided inputs.
   - The system initializes the simulation environment based on the valid inputs.

   Postconditions: The simulation environment is set up and ready to start.

   Exceptions:

If inputs are invalid, the system alerts the user and requests re-entry of data.

2. Run Simulation

Actor: User

Preconditions: The simulation is properly set up.

Main Flow:

The user starts the simulation.

The system processes each client as per the arrival times and the selected strategy (shortest queue or shortest time).

System logs are updated in real-time reflecting the status of queues and client servicing.

Postconditions: The simulation runs through the set time interval and provides output in terms of average waiting time, service time, and other relevant metrics.

Exceptions: Interruptions or errors during the simulation are logged, and the user is informed of the issue.

3 Design

This section presents the conceptual architecture and design details of the queue management system, including the object-oriented programming structure, UML diagrams, and the design patterns employed.

Architectural Overview

The application is designed using an MVC (Model-View-Controller) architecture to separate the logic of the application from the user interface and data management:

- Model: Represents the data structure and logic — Client, Server, Scheduler.
- View: Interface for user interaction for simulation setup and real-time updates.
- Controller: Manages the interaction between Model and View, handling user input and system output.

Class Design

Here are the primary classes and their responsibilities:

Client (model.Client):

Attributes: id, arrivalTime, serviceTime, startTime

Methods: getStartTime(), setStartTime(int)

Description: Represents a client in the system with an ID, arrival time, service time, and the time when service begins.

Server (model.Server):

Attributes: tasks (a BlockingQueue<Client>), waitingPeriod (an AtomicInteger), isActive (a boolean)

Methods: addTask(Client), stopServer(), run(), isActive(), getWaitingPeriod()

Description: Manages a queue of clients, processing them based on service time. Uses synchronization to handle concurrent access to the queue.

Scheduler (scheduler.Scheduler):

Attributes: servers (a List<Server>), strategy (Strategy)

Methods: changeStrategy(SelectionPolicy), dispatchTask(Client), getServers()

Description: Handles the distribution of clients to servers according to a strategy pattern. This allows for flexible switching between different queue management strategies.

UML Class Diagram

The UML diagram includes the following key elements:

- Associations between classes like SimulationManager and Server, Server and Client, and Scheduler and Server.
- Inheritance shown by the Strategy interface implemented by ConcreteStrategyTime and ConcreteStrategyQueue.
- Aggregations indicating ownership, such as Scheduler containing multiple Server instances.

Design Patterns Used

- Strategy Pattern: Used in the Scheduler class to switch between different queue management strategies (ConcreteStrategyTime and ConcreteStrategyQueue). This pattern allows the queue management policy to be selected dynamically at runtime based on the system's needs or configuration.
- Singleton Pattern: Used for the EventLogger and SimulationClock to ensure a single instance of each is used throughout the application, providing a global point of access.

Data Structures

- BlockingQueue: Used in Server to manage tasks effectively in a thread-safe manner.
- ArrayList: Used in Scheduler to hold the list of servers.
- AtomicInteger: Used in Server for managing the waiting period atomically to prevent race conditions.

Interaction Diagrams

Sequence diagrams to show how a client is processed:

1. Client Arrival: SimulationManager generates a Client, logs the arrival, and moves the client to the appropriate Scheduler.
2. Task Dispatch: Scheduler uses the current strategy to assign the client to the least busy Server.
3. Service Process: The Server processes the client, logs the start and completion of service, and updates the client's service start time.

System Interfaces

GUI Interface: Provides fields for inputting simulation parameters and buttons for starting and stopping the simulation. Displays real-time updates of queue statuses and logs.

4. Implementation

Implementation of Major Classes

Client (model.Client):

Purpose: Represents a client in the queue system.

Key Methods:

- setStartTime(int): Sets the start time when the client begins receiving service.

Server (model.Server):

Purpose: Manages a queue of clients and processes them based on their service times.

Key Methods:

- addTask(Client): Adds a client to the server's queue and updates the waiting period.
- run(): Processes clients from the queue, simulating service by sleeping for the duration of the service time.

Scheduler (scheduler.Scheduler):

Purpose: Distributes clients to servers based on a selected strategy.

Key Methods:

dispatchTask(Client): Applies the current strategy to place the client in an appropriate server.

Threading and Synchronization

Threading:

- Each Server runs in its own thread, allowing simultaneous processing of clients.
- The SimulationManager manages the timing and distribution of tasks in its own thread.

Synchronization:

- Use of synchronized methods and blocks to manage access to shared resources, such as when a server updates its waiting time or when a client is added to a queue.

EventLogger (logger.EventLogger):

Used to log significant events in the system, such as client arrivals and service completions.

Outputs logs to both the console and a text file for review.

5. Results

Testing Scenarios

Basic Functionality Test:

Objective: To verify that the system correctly handles a small number of clients and queues.

Setup:

Number of clients: 10

Number of queues: 2

Simulation duration: 60 seconds

Procedure:

Clients are generated with random arrival and service times within specified limits.

The simulation is run, and the system logs all activities.

Expected Results:

All clients are processed within the simulation time.

Logs correctly reflect each client's journey from arrival to completion of service.

Outcome: The test confirmed that the system manages a basic scenario without errors, and all events are logged accurately.

Concurrency and Load Test:

Objective: To assess the system's performance and stability under heavy load.

Setup:

Number of clients: 100

Number of queues: 10

Simulation duration: 120 seconds

Procedure:

A higher volume of clients is generated to test the system's response to increased concurrency and processing demands.

System behavior and queue management efficiency are monitored.

Expected Results:

The system remains stable and responsive.

Queue distribution is efficient, and no client exceeds the maximum waiting time threshold.

Outcome: The system demonstrated robustness under load, handling multiple clients concurrently without significant delays or issues.

Stress Test:

Objective: To push the system to its operational limits and observe how it copes with extreme stress.

Setup:

Number of clients: 500

Number of queues: 20

Simulation duration: 180 seconds

Procedure:

Extremely high numbers of clients are processed to test the system's limits.

System metrics such as processing delays, error rates, and resource utilization are analyzed.

Expected Results:

The system might show signs of strain such as increased processing times and resource usage.

Errors or slowdowns are expected to be handled gracefully without system crashes.

Outcome: The system managed to operate under extreme conditions, although with noticeable slowdowns in processing. It remained stable without crashing, validating the effectiveness of its error-handling and resource management strategies.

Results Analysis

Average Waiting Time: Calculated from the sum of individual waiting times divided by the number of clients, indicating the efficiency of queue management.

Service Efficiency: Measured by the average and maximum service times, reflecting the system's ability to quickly process clients.

Resource Utilization: Assessed to determine the efficiency in using system resources under varying loads.

6. Conclusion

Achievements:

Efficient Queue Management: The application efficiently distributed clients across multiple queues, balancing load and minimizing waiting times, which is evident from the average waiting times recorded during tests.
Robustness under Load: The system maintained stability and performance under high load, managing up to 100 clients simultaneously without significant performance degradation.
Scalability: The architecture proved to be scalable, handling increased loads by adjusting the number of queues and clients dynamically. This scalability is crucial for potential real-world applications where demand can fluctuate dramatically.

Future Developments:

Advanced Scheduling Algorithms: Implementing more sophisticated scheduling algorithms could further reduce waiting times and improve system efficiency.
Adaptive Load Balancing: Developing adaptive mechanisms that can dynamically adjust the distribution strategy based on real-time workload and performance metrics.
Enhanced User Interaction: Integrating more interactive elements and detailed analytics in the GUI would provide users with better insights and control over the simulation parameters and results.

7. Bibliografy

1. Oracle "The Java Tutorials."
2. Eckel, Bruce.** *Thinking in Java (4th Edition)
3. Java Threads