DOCUMENTATION

PROJECT NR. 2

STUDENT NAME: Pop Maria Alexandra
GROUP: 30425

CONTENTS

1. Assignment Objective

The objective of this assignment is to design and implement a queues management application that optimally assigns clients to queues to minimize their waiting time. The application simulates a real-world queue management system, commonly used in various domains to model scenarios where entities (clients) wait for services.

Key Goals:

- Minimize Client Waiting Time:
  The primary goal is to ensure that the clients spend the least amount of time waiting in queues before being served.
- Efficient Queue Management:
  The system should manage multiple queues efficiently, using strategies to distribute clients based on the queue's current state.
- Simulation of Real-World Scenarios:
  The application will simulate clients arriving, waiting, being served, and leaving the queues within a defined simulation period.
- Random Client Generation:
  Clients are generated randomly within specified parameters for arrival and service times.
- Multithreading:
  Each queue operates in its own thread to simulate concurrent processing of clients.
- Thread Safety:
  Use appropriate synchronized data structures to ensure thread safety in a concurrent environment.
- Logging Events:
  Log the events of the simulation, including client arrivals, queue states, and statistics, to a text file.

2. Problem Analysis

The primary problem of this project is to efficiently manage a system of queues in order to minimize client waiting times and operational costs for the service supplier. The application achieves this by simulating a real-world scenario where a series of clients arrive for service, wait in queues, are served, and eventually leave the queues. Each client is characterized by three essential parameters: ID, arrival time, and service time. The goal is to ensure that the total waiting time for all clients is minimized while managing the resource costs effectively.

Detailed Analysis
Client Characteristics:

- ID: A unique identifier for each client, ranging from 1 to N (the total number of clients).
- Arrival Time: The specific time at which a client arrives and is ready to enter the queue. This is a crucial parameter as it determines when the client will be added to a queue.

- Service Time: The duration required to serve the client once they reach the front of the queue. This parameter affects the total waiting time for other clients in the queue.

System Components:

- Queues (Q): Multiple queues are used to manage clients. Each queue has its own processing thread to simulate concurrent servicing of clients.
- Simulation Interval (t_simulationMAX): The total duration for which the simulation runs. During this interval, clients arrive, wait, are served, and then leave the system.
- Arrival Time Range ($t\_arrivalMIN \leq t\_arrival \leq t\_arrivalMAX$): Defines the window during which clients can arrive. This range helps in generating clients with varied arrival times to simulate real-world randomness.
- Service Time Range ($t\_serviceMIN \leq t\_service \leq t\_serviceMAX$): Defines the possible durations for which clients need to be served. This range adds variability to the service times, making the simulation more realistic.

System Requirements:
User Inputs:

- Number of clients (N).
- Number of queues (Q).
- Simulation interval (t_simulationMAX).
- Minimum and maximum arrival times (t_arrivalMIN, t_arrivalMAX).
- Minimum and maximum service times (t_serviceMIN, t_serviceMAX).

Client Distribution Strategies:

- Shortest Queue Strategy: Assigns a client to the queue with the least number of clients.
- Shortest Time Strategy: Assigns a client to the queue with the shortest total waiting time.

System Behavior:

- Client Generation: Clients are generated randomly within the defined arrival and service time ranges when the simulation starts. This ensures diversity in client parameters, mimicking a real-world scenario.
- Client Assignment: As clients arrive (based on their arrival times), they are assigned to queues using the selected strategy (shortest queue or shortest time).
- Queue Processing: Each queue processes its clients concurrently in separate threads. The queue removes a client after serving them for the specified service time.
- Synchronization: To ensure thread safety, synchronized data structures and mechanisms are used. This prevents race conditions and ensures consistent updates to shared variables like total waiting time.
- Event Logging: The application logs various events, including client arrivals, queue assignments, and service completions. This log is maintained in a text file for review and analysis.

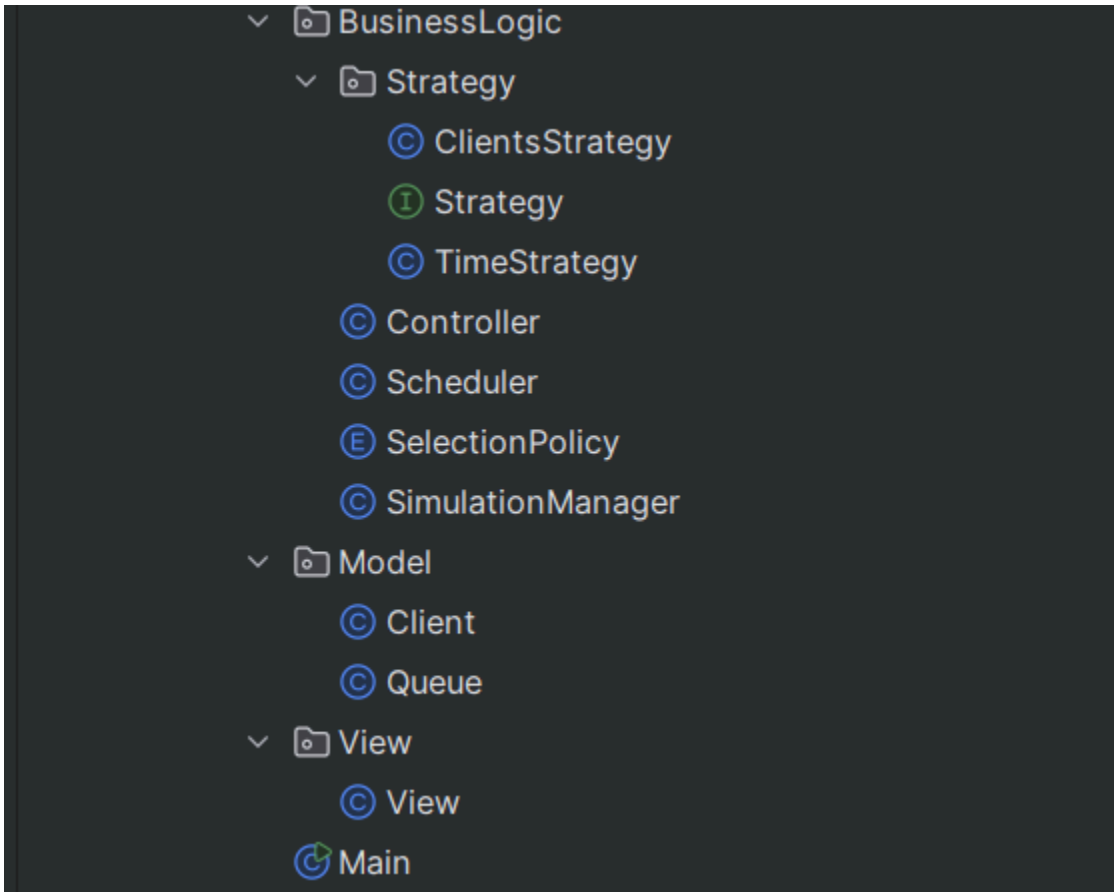
Performance Metrics:

- Total Time Spent by Clients: The application tracks the total time each client spends in the system, from arrival to service completion.
- Average Waiting Time: Calculated as the mean of all clients' waiting times. This metric helps in evaluating the efficiency of the queue management system.
- Peak Hour: The time during the simulation when the system experiences the maximum load (i.e., highest total waiting time).

Challenges and Solutions:

- Concurrency Management: Managing multiple queues in parallel threads requires careful synchronization to prevent data inconsistencies. The use of BlockingQueue and synchronized blocks ensures thread-safe operations.
- Random Client Generation: Generating clients with random but controlled arrival and service times to ensure the simulation remains realistic and manageable.
- Efficient Queue Assignment: Implementing strategies that efficiently distribute clients across queues to minimize overall waiting time while avoiding overloading any single queue.

2.2 Modeling: Packages and Classes



Packages

- Model:
  Core entities like clients and queues.
- View:
  Manages the graphical user interface and user interactions.
- BusinessLogic:
  Core logic, client distribution strategies, scheduling, and simulation management.
- Strategy:
  Defines strategies for assigning clients to queues.

Classes

Model Package

- Client
  Represents a client with an ID, arrival time, and service time.
- Queue:
  Manages clients in a queue, processes them in a separate thread, tracks waiting time.

View Package

- View:
  Provides and manages the user interface for input and simulation results.

BusinessLogic Package

- Controller:
  Connects the GUI with the business logic, manages user inputs, and controls the simulation.
- Scheduler:
  Manages the assignment of clients to queues based on selected strategies.
- SimulationManager:
  Handles the simulation process, including client generation and queue assignment.
- SelectionPolicy:
  Enum representing different strategies for client distribution.

Strategy Package

- Strategy:
  Interface for different client distribution strategies.
- ClientsStrategy:
  Assigns clients to the queue with the fewest clients.
- TimeStrategy:
  Assigns clients to the queue with the shortest waiting time.

2.3 Scenarios

Scenario 1: Normal Operation

- Description: The user inputs valid parameters and starts the simulation. Clients are generated, assigned to queues, processed, and results are displayed.
- Possible Complications: None.
- Error Handling: Not required as inputs are valid.

Scenario 2: Invalid Input Data

- Description: The user inputs non-numeric values for the number of clients, queues, or time intervals.
- Possible Complications: Application crashes or behaves unexpectedly due to invalid input types.
- Error Handling:
- Validation: Check inputs and show error messages for invalid data.
- Example: If a non-numeric value is entered, display a dialog box: "Please enter a valid number."

Scenario 3: Extreme Input Values

- Description: The user inputs extremely high values for the number of clients or queues.
- Possible Complications: Application may slow down or run out of memory.
- Error Handling:
- Validation: Set maximum limits for the number of clients and queues.
- Example: If the input exceeds the maximum limit, display a dialog box: "The number of clients/queues exceeds the allowed limit."

3. DESIGN

The Queues Management Application is structured using object-oriented programming (OOP) principles. The application is organized into several packages, each focusing on a different aspect of the application's functionality, such as user interface, client management, and queue management. The class diagram illustrates the relationships between various classes, such as the Client class and Queue class.

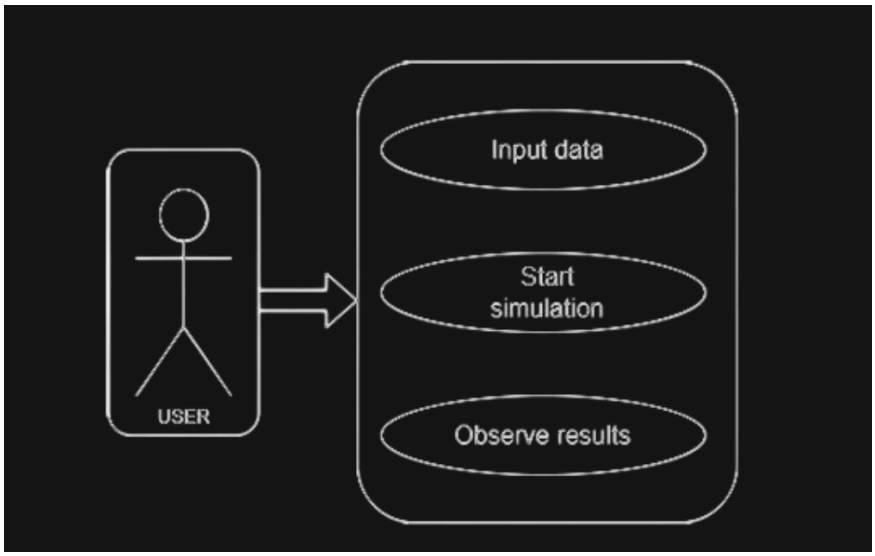
The application's user interface is crafted to be intuitive and user-friendly. Users can input essential data, including the number of clients, number of queues, simulation interval, minimum and maximum arrival times, and minimum and maximum service times. The application then runs the queue management simulation and displays results, such as the average waiting time, providing users with clear insights into the system's performance.

USER DIAGRAM

The user diagram illustrates the interaction between the user and the queue management application in three main steps:

- Input Data: Users input necessary simulation parameters such as the number of clients, queues, simulation interval, and arrival/service times.
- Start Simulation: Users initiate the simulation, which runs based on the provided parameters, managing client generation and queue assignments.
- Observe Results: Users review the simulation results, including metrics like average waiting time and total time spent in queues, to analyze system performance.

This diagram succinctly outlines the user's journey from data entry to result analysis.



PACKAGE DIAGRAM

The purpose of Java packages is to group related classes together for better organization and management. In Object-Oriented Programming (OOP), the Model-View-Controller (MVC) pattern is a prevalent approach for designing projects to achieve optimal efficiency. This pattern is widely adopted across various programming languages. In this project, a modified version of the MVC pattern is employed. The model and view components are kept as standard, while the controller section is restructured into a package named BusinessLogic. This package contains a sub-package named Strategy for enhanced organization and clarity.

Class diagram

The UML (Unified Modeling Language) diagram provides a visual representation of the software system, illustrating the classes used in the project and their relationships and dependencies. It serves as a blueprint for the application's design, highlighting the structure and interactions between different components.

4. IMPLEMENTATION

Class 'Client':

- Attributes: ID (unique identifier), arrival time, service time.
- Constructor: Initializes client attributes.
- Getters and Setters: Accessor and mutator methods for attributes.

- toString(): Returns a string representation for logs/interface.

Class 'Queue':

- Represents: A queue handling clients.
- Implements: Runnable interface for multithreading.
- Data Structure: Uses BlockingQueue for thread safety.
- Attribute: waitingTime tracks total waiting time.
- run(): Decrements service time per second, removes clients.
- addClient(Client c): Adds client to queue, updates waiting time.
- Getters and Setters: Accessor and mutator methods.
- toString(): Displays "(I)" for each client, for GUI aesthetics.

Class 'Scheduler':

- Purpose: Assigns clients to queues based on policy.
- Attributes: List of queues.
- Constructor: Initializes queues, starts threads, sets strategy.
- changeSelectionPolicy(): Changes client assignment strategy.
- calculateAWT(): Calculates average wait time of queues.
- addClient(): Assigns clients to queues based on strategy.
- toString(): Formats queue information for event logs.

Package 'Strategy':

- Interface 'Strategy': Method addClient for queue assignment.
- Class 'ClientsStrategy': Assigns clients to queue with fewest clients.
- Class 'TimeStrategy': Assigns clients to queue with shortest wait time.

Class 'SimulationManager':

- Attributes: Scheduler, client list, controller, view.
- Client Generation: Randomly generates clients using Math.random().
- run(): Loops from 0 to time limit, assigns clients, updates interface, logs events, calculates metrics.

Class 'View':

- Components: Buttons, labels, text fields for GUI.
- Layout: Arranges components in a JPanel, adds to JFrame.
- Aesthetics: Different font sizes for buttons/labels.
- ActionListeners: Methods to add listeners for buttons.
- updateView(): Displays current simulation information in GUI.
- Client Representation: "(I)" symbol for clients in queues.
- Client List: Updates as clients arrive.

Class 'Controller':

- Purpose: Links GUI buttons with functionalities.
- ActionListener Classes: Handles button actions.
- Data Retrieval: Gets simulation parameters from GUI.
- Class 'Main':
- Functionality Test: Tests queue functionality.
- Objects: Initializes view and controller.

Random Client Generator

The "Random Client Generator" functionality is implemented in the generateClients method of the SimulationManager class. This method is responsible for creating clients with random arrival and service times. Here's the relevant part of the code from SimulationManager class:

```
public ArrayList<Client> generateClients() {
   ArrayList<Client> temp = new ArrayList<Client>();
   for(int i = 0; i < noClients; i++) {
      Client c = new Client(i, randomNumber(arrivalTimeInterval),
randomNumber(processingTimeInterval));
      temp.add(c);
   }
   Collections.sort(temp, new SortClients());
   return temp;
}
```

supporting method:

```
public int randomNumber(int[] interval) {
   Random random = new Random();
   return (int) ((Math.random() * (interval[1] - interval[0])) + interval[0]);
}
```

Appropriate synchronized data structures to assure thread safety

To ensure thread safety in a multithreaded application, it is crucial to use appropriate synchronized data structures. Java provides several thread-safe collections and classes that can be used for this purpose.

BlockingQueue in Queue Class

Use: To store and manage clients in a thread-safe manner, ensuring that client addition and removal operations are thread-safe and handle concurrency appropriately.
Structure: ArrayBlockingQueue

```
package Model;

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

public class Queue implements Runnable {

    private BlockingQueue<Client> queue; // stores the clients in queue
    private int waitingTime; // total waiting time in the queue

    public Queue() {
        queue = new ArrayBlockingQueue<>(100); // capacity of the queue
        this.waitingTime = 0;
    }
```

Explanation:
BlockingQueue: ArrayBlockingQueue is used to ensure that client operations like addition and removal are thread-safe. The blocking behavior ensures that threads attempting to take from an empty queue or put into a full queue will wait appropriately, preventing concurrency issues.

Multithreading: one thread per queue

The code snippet that demonstrates the creation of a separate thread for each queue. This is found in the Scheduler class where each Queue instance is started in its own thread.

```
public class Scheduler {

    private List<Queue> queueList; // List of all queues managed
    private Strategy strategy; // Used strategy

    public Scheduler(int maxQueues, SelectionPolicy policy) {
        queueList = new ArrayList<>();
        for (int i = 0; i < maxQueues; i++) {
            Queue q = new Queue();
            Thread t = new Thread(q); // Create a new thread for each queue
            t.start(); // Start the thread
            queueList.add(q); // Add the queue to the list
        }
        changeSelectionPolicy(policy);
    }

}
```

Log of events displayed in a .txt file

1. Creating the Log File

In the Controller class, the createLog method creates a new log file or opens an existing one.

```
public void createLog() {
    try {
        log = new File("EventLog.txt");
```

```
        if (log.createNewFile()) {
            System.out.println("File created: " + log.getName());
        } else {
            System.out.println("File already exists.");
        }
    } catch (IOException e) {
        System.out.println("An error occurred.");
        e.printStackTrace();
    }
}
```

2. Writing Events to the Log

The writeInLog methods in the Controller class handle logging different types of events.

```
public void writeInLog(int currentTime, String clientList, String queues) {
    try (FileWriter myWriter = new FileWriter(log, true)) {
        myWriter.append("Time: ").append(String.valueOf(currentTime)).append("\n");
        myWriter.append("Clients: ").append(clientList).append("\n");
        myWriter.append(queues).append("\n");
    } catch (IOException e) {
        System.out.println("An error occurred.");
        e.printStackTrace();
    }
}

public void writeInLog(double averageWaitingTime, double averageServiceTime, int peakHour) {
    try (FileWriter myWriter = new FileWriter(log, true)) {
        myWriter.append("Average waiting time:
").append(String.valueOf(averageWaitingTime)).append("\n");
        myWriter.append("Average service time:
").append(String.valueOf(averageServiceTime)).append("\n");
        myWriter.append("Peak hour: ").append(String.valueOf(peakHour)).append("\n");
    } catch (IOException e) {
        System.out.println("An error occurred.");
        e.printStackTrace();
    }
}
```

3. Logging During Simulation

In the SimulationManager class, the run method calls the logging methods at appropriate times to log current state and summary statistics.

5. RESULTS:

I successfully completed the task of calculating the average waiting time, average service time, and peak hour for each simulation.

Average Waiting Time:

- This is calculated in the scheduler by iterating through the list of queues and summing their waiting times.
- The sum of the waiting times is then divided by the total number of queues.

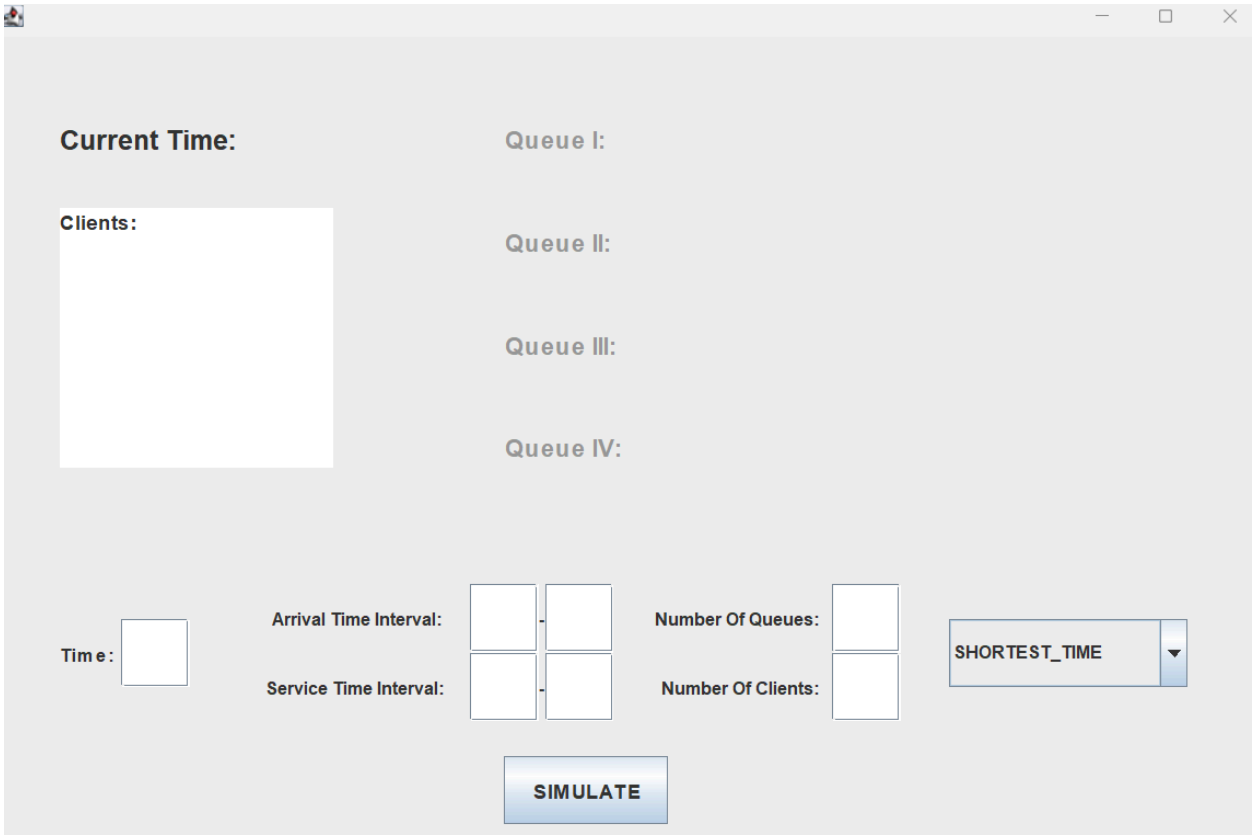
- The average waiting time is recalculated at each time step, making it essential to print the current result from the scheduler.

Average Service Time:

- This is determined by iterating through the sorted list of clients and computing the arithmetic mean of their service times.

Peak Hour:

- The peak hour, or the time with the highest load during the simulation, is tracked at time step.
- We monitor the maximum number of clients in the queues or the maximum waiting time.
- At each step, we compare the current state with the maximum recorded state. If the current state exceeds the previous maximum, 'peakHour' is updated to the current time.



6. CONCLUSIONS

This project successfully demonstrates efficient queue management using multithreading and synchronized data structures in Java. By implementing one thread per queue, the application ensures concurrent processing of clients, significantly improving performance.

Overall, the project effectively balances complexity and usability, showcasing key concepts of concurrent programming and user interface design in Java.

7. BIBLIOGRAPHY

https://stackoverflow.com/

https://www.youtube.com/

https://www.geeksforgeeks.org/

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