DOCUMENTATION

ASSIGNMENT *2*

STUDENT NAME: Borzan Călina-Annemary
GROUP:30424

CONTENTS

1. Assignment Objective

Main Objective

The assignment's main objective is to develop and execute an application that examines the queuing system by simulating the arrival of N clients, their distribution around the Q queues, their service and arrival time, and calculating the average waiting time, the average serving time and identifying peak hour.

Sub-Objectives

Sub-Objective	Description	Section
Analyze Requirements and Create a Working Plan	This phase involves a comprehensive analysis of the requirements outlined for the queues management application. The goal is to dissect and understand the intricate details of the problem statement, focusing on key aspects such as client queuing0, service duration, and system optimization. Through this analysis, we aim to identify the core functionalities and constraints of the application, including the simulation parameters, client generation rules, and queue management strategies.	Problem Analysis, Modeling, Scenarios, Use Cases
Creating the architecture of the system	Outline the way the architecture of the system will look like using object-oriented design principles: UML package, class diagrams, data structure, interfaces and algorithms.	Design
Implement Functionality	In this phase, the focus is on implementing the core functionality of the queuing simulation application. We will decide on the approach for handling various aspects such as random client generation, queue management, service processing, the average time processing.	Implement
Develop the graphical interface	Create the GUI with which the user will interact.	Implement

Reflect and Developments	Analyze the outcome of the project and find certain future developments that could be implemented.	Conclusion

2. Problem Analysis, Modeling, Scenarios, Use Cases

2.1. Functional Requirements:

- The system should manage the arrival, service, and departure of clients in simulated queueing environment.
- Servers must accept clients based on two types of strategies: Shortest Queue and Shortest Waiting Time.
- The system should be able to be initialized with parameters like N numbers of clients, Q numbers of serves, maximum simulation time, and arrival/service time ranges.
- Execute the simulation over a specified time, handle client arrival and their processing.
- Calculate and record important information such as average waiting time, average serving time and peak hour.
- Update the queues at regular time, every second, update the log accordingly and update at the end the calculated metrics.
- Provide a graphical representation of the simulation, allowing users to visualize the queueing system’s state, including client queues and server utilization.
- Update the visualization in real-time to reflect changes during the simulation execution.
- Enable users to enter the inputs and to start the simulation and to give them the possibility of choosing the strategy wanted.

2.2. Use Case

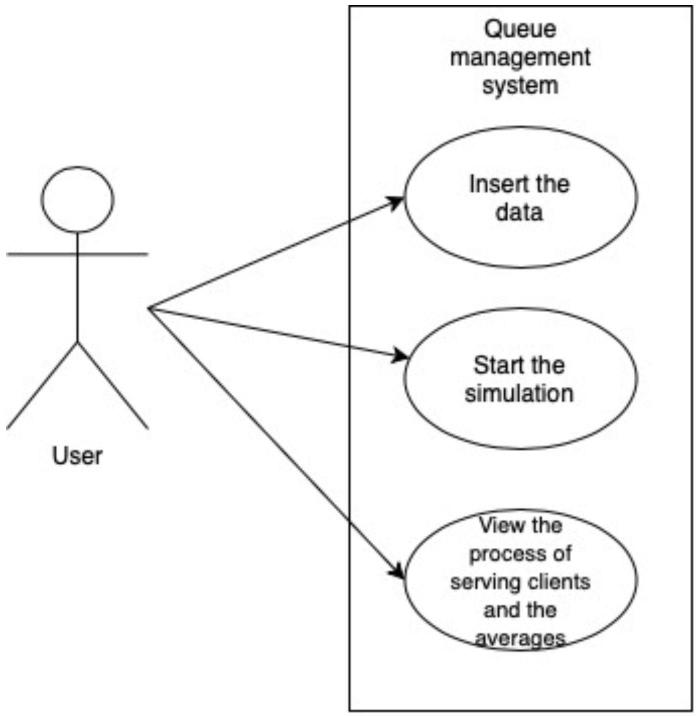


Figure 1:Use Case Diagram

Use Case Description:

- **Insert the data**: The user interacts with the Queue Management system to input data required for simulation. The system presents the user with input fields for configuring simulation parameters, such as the number of servers, maximum simulation time, number of clients, maximum arrival time, minimum arrival time, maximum processing time, and minimum processing time. The user enters values for each parameter using the keyboard.

- **Start the simulation**: Upon completing the input, the user submits the data to initiate the simulation.

- **View the process of serving client and the averages**: During the simulation process, the system updates the user interface to display the current state of the queue and server activity. The user observes the queue dynamics, including client arrivals, queue lengths, and server utilization. As the simulation progresses, the system calculates and displays average metrics such as average waiting time and average serving time. The user monitors the simulation output to analyze the efficiency and performance of the queue management system.

3. Design

In the queue management application were used a bunch of OOP principles, ensuring modularity, reusability, and ease of maintenance. The application is divided into packages that segregate the model, view and controller logic used in Model-View-Controller architectural pattern.

3.1. OOP Design

- Encapsulation: is the mechanism of bundling the data and methods that operate on the data into a single class, and restricting access to some of the object's components. : Each class encapsulates related data and behavior within its scope. For example, you have classes like Client, Server, SimulationManager, ShortestQueueStrategy, and ShortestTimeStrategy. These classes encapsulate attributes and methods relevant to clients, servers, simulation management, and strategies for distributing clients. : Encapsulation boundaries are established to control access to internal members of a class. For instance, the Server class encapsulates a queue of clients and exposes methods like addClient(), getClients(), getWaitingTime(), and endSimulation(). These methods allow controlled interaction with the server's internal state while hiding implementation details.

- Polymorphism: allows objects of different classes to be treated as objects of a common superclass, enabling code to be written that operates on objects of various types without needing to know their specific class. Polymorphism is commonly achieved through interfaces and their implementations. In your code, the Strategy interface defines a contract for strategies to distribute clients among servers. The ShortestQueueStrategy and ShortestTimeStrategy classes implement this interface, providing different implementations for distributing clients based on the shortest queue or shortest waiting time.

- Composition is a design principle in object-oriented programming where a class contains objects of other classes as member variables. This allows for building complex objects by combining simpler ones. Composition promotes code reuse, modularity, and flexibility The SimulationGUI class creates an instance of the Queues class, demonstrating composition. The Queues class is responsible for visualizing the queues in the GUI. By including an instance of Queues within

SimulationGUI, the GUI component delegates the responsibility of queue visualization to the Queues class, promoting separation of concerns and modularity.

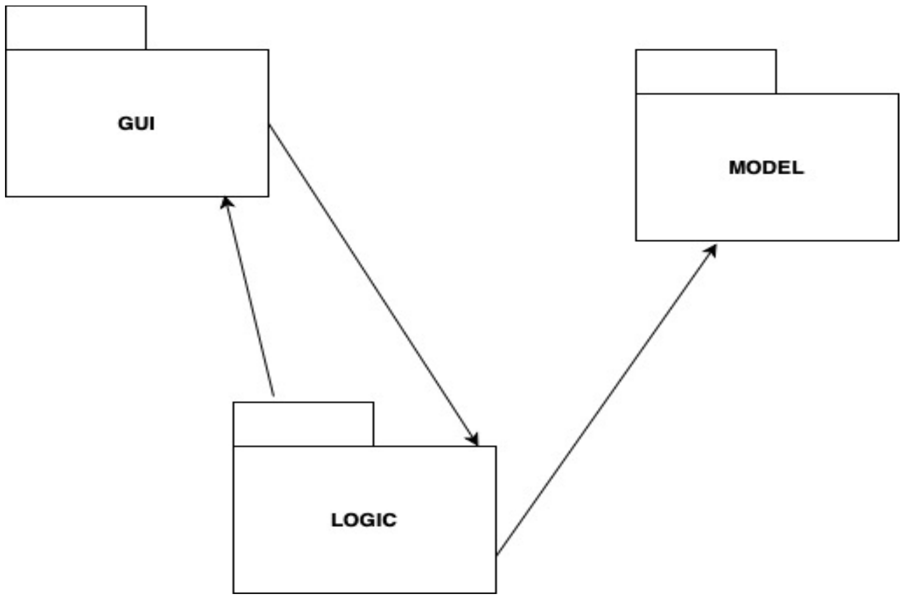
- Modularity: The fact that application is divided in multiple packages that can be maintained independently.

3.2. UML Packages

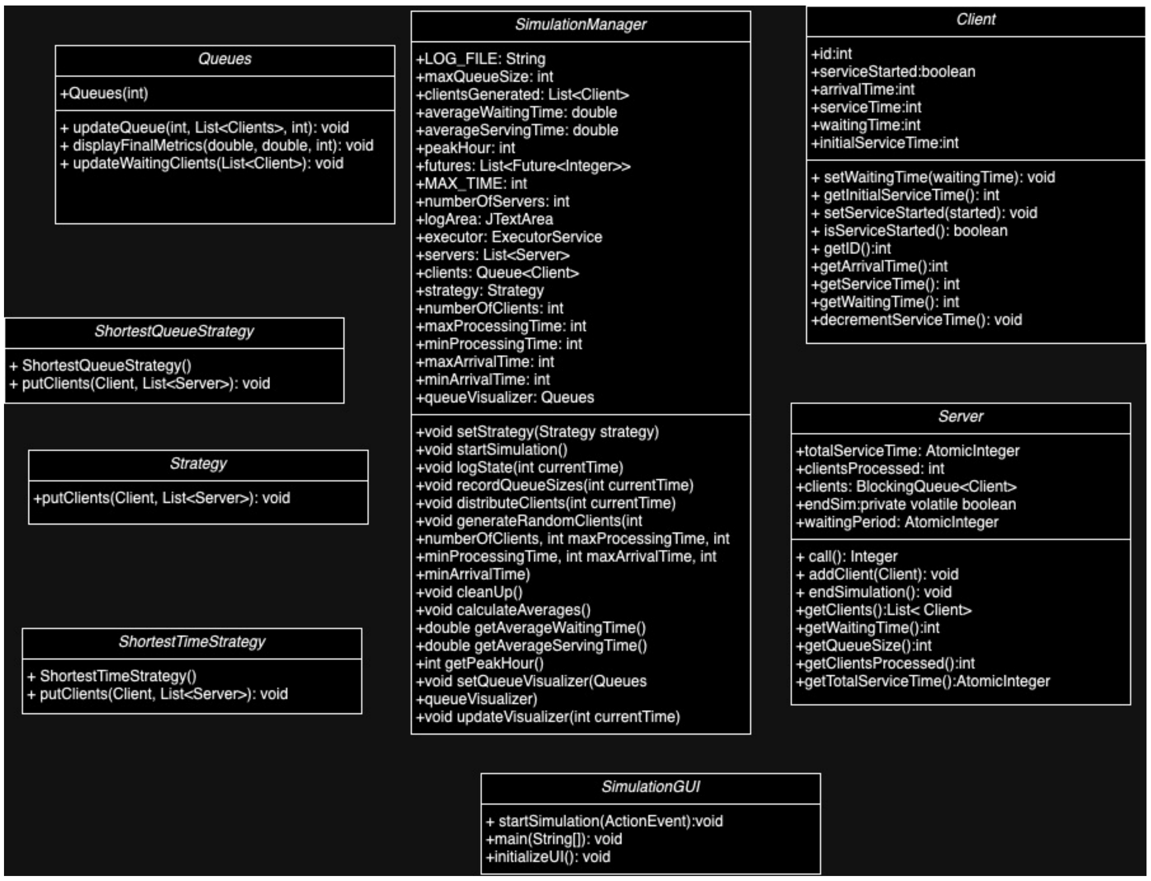
Packages play a critical role in the architectural design of a software application by allowing better organization, enhancing modularity. In UML diagrams packages are depicted with their contained elements, classes and interfaces to avoid clutter. In our program they helped organizing the program into 3 main parts:

- MODEL: This package represents the foundation of the Queue Managements System, containing the essential data structures like ‘Client’ and ‘Server’ classes. It serves as the basis for this system, containing the classes that are the core representations.
- LOGIC: This package represents the operational logic of this program. It includes classes that take care of the logical operation of the simulation ‘Simulation Manager’, and ‘Strategy’ for implementing the part where you can choose how the client to be distributed. That’s why we have here also ‘ShortestQueueStrategy’ and ‘ShortestTimeStrategy’. This package is the application’s brain.
- GUI: This package is dedicated to the user interface containing the class ‘SimulationGUI’ and ‘Queues’ that takes care of the interaction of the user with this system.

When it comes to dependency the GUI depends on the LOGIC, LOGIC depends on MODEL and GUI maintaining the ease and separation of the program.



3.3. Class diagram


Queues
+Queues(int)
+ updateQueue(int, List<Clients>, int): void
+ displayFinalMetrics(double, double, int): void
+ updateWaitingClients(List<Client>): void
SimulationManager
+LOG_FILE: String
+maxQueueSize: int
+clientsGenerated: List<Client>
+averageWaitingTime: double
+averageServingTime: double
+peakHour: int
+futures: List<Future<Integer>>
+MAX_TIME: int
+numberOfServers: int
+logArea: JTextArea
+executor: ExecutorService
+servers: List<Server>
+clients: Queue<Client>
+strategy: Strategy
+numberOfClients: int
+maxProcessingTime: int
+minProcessingTime: int
+maxArrivalTime: int
+minArrivalTime: int
+queueVisualizer: Queues
+void setStrategy(Strategy strategy)
+void startSimulation()
+void logState(int currentTime)
+void recordQueueSizes(int currentTime)
+void distributeClients(int currentTime)
+void generateRandomClients(int
+numberOfClients, int maxProcessingTime, int
+minProcessingTime, int maxArrivalTime, int
+minArrivalTime)
+void cleanUp()
+void calculateAverages()
+double getAverageWaitingTime()
+double getAverageServingTime()
+int getPeakHour()
+void setQueueVisualizer(Queues
+queueVisualizer)
+void updateVisualizer(int currentTime)
Client
+id:int
+serviceStarted:boolean
+arrivalTime:int
+serviceTime:int
+waitingTime:int
+initialServiceTime:int
+ setWaitingTime(waitingTime): void
+ getInitialServiceTime(): int
+ setServiceStarted(started): void
+ isServiceStarted(): boolean
+ getID():int
+getArrivalTime():int
+getServiceTime(): int
+getWaitingTime(): int
+decrementServiceTime(): void
ShortestQueueStrategy
+ ShortestQueueStrategy()
+ putClients(Client, List<Server>): void
Strategy
+putClients(Client, List<Server>): void
ShortestTimeStrategy
+ ShortestTimeStrategy()
+ putClients(Client, List<Server>): void
Server
+totalServiceTime: AtomicInteger
+clientsProcessed: int
+clients: BlockingQueue<Client>
+endSim:private volatile boolean
+waitingPeriod: AtomicInteger
+ call(): Integer
+ addClient(Client): void
+ endSimulation(): void
+getClients():List< Client>
+getWaitingTime():int
+getQueueSize():int
+getClientsProcessed():int
+getTotalServiceTime():AtomicInteger
SimulationGUI
+ startSimulation(ActionEvent):void
+main(String[]): void
+initializeUI(): void

3.4. Data Structures

- List<Client>: Used in the SimulationManager class to store generated clients (clientsGenerated list) and server clients (serverClients list). Represents an ordered collection of clients, allowing efficient iteration and manipulation of client data.

- Map<Integer, Integer>: Utilized in the SimulationManager class to store client arrival counts (clientArrivalCounts map). Key-value pairs where the key represents the client ID and the value represents the number of times the client arrives.

- Queue<Client>: Employed in the SimulationManager class to manage the queue of clients waiting for service (clients queue). Ensures first-in-first-out (FIFO) behavior, essential for modeling client queues in the simulation.

- BlockingQueue<Client>: Used in the Server class to represent the queue of clients waiting for service (clients queue). Implemented: LinkedBlockingDeque. Supports concurrent access by multiple threads while maintaining thread safety.

- ExecutorService: Utilized in the SimulationManager class to manage the execution of server tasks concurrently. Enables efficient utilization of multiple threads for processing clients and servers simultaneously.

- AtomicInteger: Employed in the Server class to manage waiting periods and total waiting time (waitingPeriod and totalWaitingTime). Provides atomic operations on integer values, ensuring thread-safe access in a multi-threaded environment.

3.5. Defined Interfaces

In the provided code, the Strategy interface is defined. This interface allows for the implementation of different strategies for distributing clients among servers in the simulation. Here's an overview of the interface:

4. Implementation

In this part I am going to give details about each class: the structure and the functionality, giving an attentive eye around the fields(attributes), important methods.

4.1. Client Class

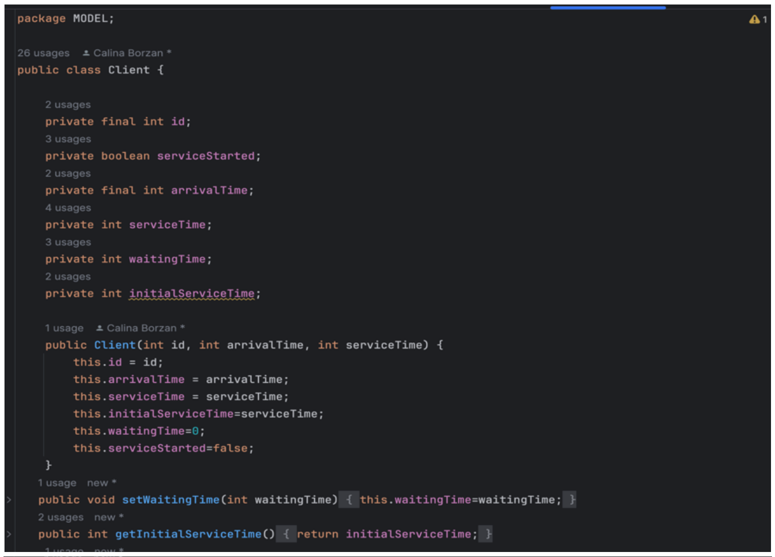
1. Package: ‘MODEL’

2. Fields:

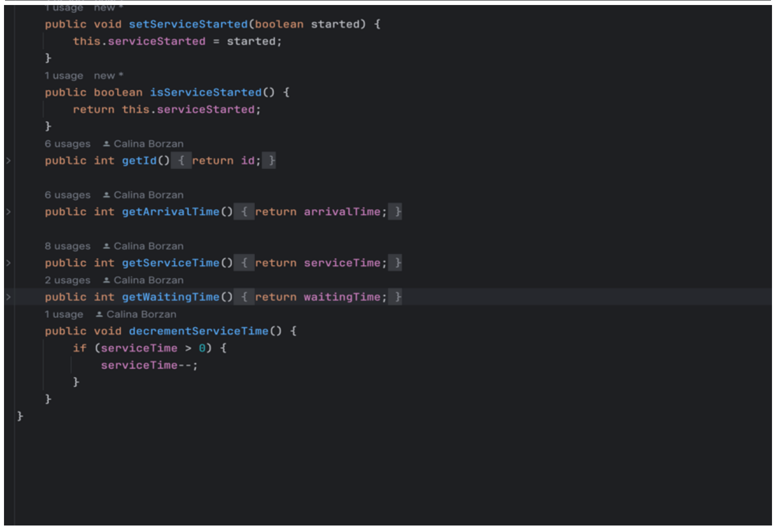
- private final int id: Unique identifier for the client.
- private boolean serviceStarted: Indicates whether the client's service has started.
- private final int arrivalTime: Time at which the client arrives.
- private int serviceTime: Time required for the client's service.
- private int waitingTime: Time the client spends waiting in the queue.
- private int initialServiceTime: Initial service time required by the client.

3. Important Methods:

- public void setWaitingTime(int waitingTime): Sets the waiting time for the client.
- public int getInitialServiceTime(): Returns the initial service time required by the client.
- public void setServiceStarted(boolean started): Sets whether the client's service has started.
- public boolean isServiceStarted(): Returns true if the client's service has started, false otherwise.
- public int getId(): Returns the ID of the client.
- public int getArrivalTime(): Returns the arrival time of the client.
- public int getServiceTime(): Returns the remaining service time required by the client.
- public int getWaitingTime(): Returns the waiting time of the client.
- public void decrementServiceTime(): Decrements the remaining service time required by the client.



```
package MODEL;

26 usages   Calina Borzan *
public class Client {

    2 usages
    private final int id;
    3 usages
    private boolean serviceStarted;
    2 usages
    private final int arrivalTime;
    4 usages
    private int serviceTime;
    3 usages
    private int waitingTime;
    2 usages
    private int initialServiceTime;

    1 usage   Calina Borzan *
    public Client(int id, int arrivalTime, int serviceTime) {
        this.id = id;
        this.arrivalTime = arrivalTime;
        this.serviceTime = serviceTime;
        this.initialServiceTime=serviceTime;
        this.waitingTime=0;
        this.serviceStarted=false;
    }
    1 usage   new *
    public void setWaitingTime(int waitingTime) { this.waitingTime=waitingTime; }
    2 usages   new *
    public int getInitialServiceTime() { return initialServiceTime; }
```



```
    1 usage   new *
    public void setServiceStarted(boolean started) {
        this.serviceStarted = started;
    }
    1 usage   new *
    public boolean isServiceStarted() {
        return this.serviceStarted;
    }
    6 usages   Calina Borzan
    public int getId() { return id; }

    6 usages   Calina Borzan
    public int getArrivalTime() { return arrivalTime; }

    8 usages   Calina Borzan
    public int getServiceTime() { return serviceTime; }
    2 usages   Calina Borzan
    public int getWaitingTime() { return waitingTime; }
    1 usage   Calina Borzan
    public void decrementServiceTime() {
        if (serviceTime > 0) {
            serviceTime--;
        }
    }
}
```

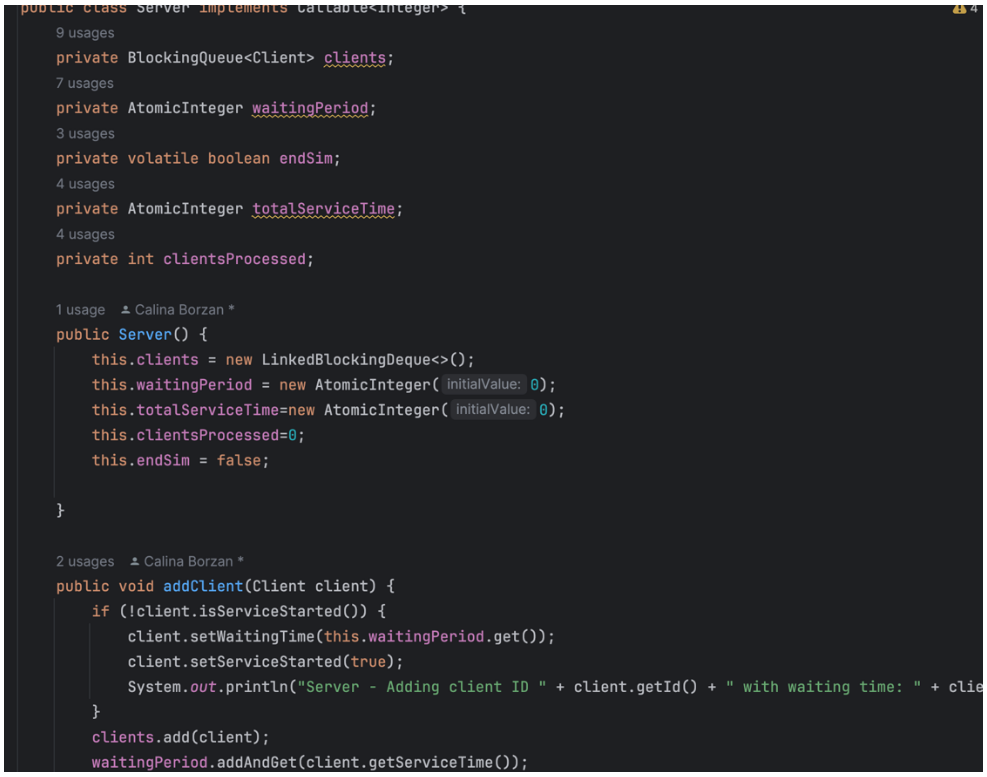
4.2. Server Class

1. **Package:** ‘MODEL’

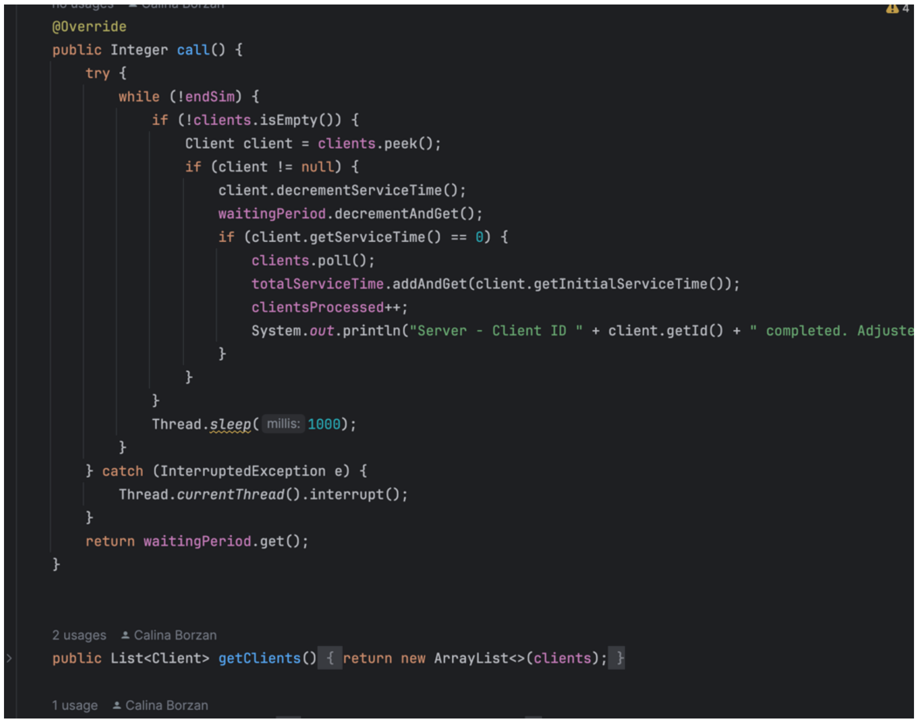
2. **Fields:**
   - private BlockingQueue<Client> clients: Queue to hold clients waiting to be served.
   - private AtomicInteger waitingPeriod: Total waiting time for all clients in the queue.
   - private volatile boolean endSim: Flag indicating whether the simulation has ended.
   - private AtomicInteger totalServiceTime: Total service time provided by the server.
   - private int clientsProcessed: Total number of clients processed by the server.

3. **Important Methods:**

- public void addClient(Client client): Adds a client to the server's queue and updates the waiting period.
- @Override public Integer call(): Overrides the call() method from the Callable interface to define server behavior. Processes clients in the queue until the simulation ends, updating waiting and service times.
- public List<Client> getClients(): Returns a copy of the list of clients in the server's queue.
- public int getWaitingTime(): Returns the total waiting time of clients in the server's queue.
- public int getQueueSize(): Returns the size of the server's queue.
- public void endSimulation(): Marks the end of the simulation, ensuring all remaining clients are processed.
- public int getClientsProcessed(): Returns the total number of clients processed by the server.
- public AtomicInteger getTotalServiceTime(): Returns the total service time provided by the server.



```java
public class Server implements Callable<Integer> {
    9 usages
    private BlockingQueue<Client> clients;
    7 usages
    private AtomicInteger waitingPeriod;
    3 usages
    private volatile boolean endSim;
    4 usages
    private AtomicInteger totalServiceTime;
    4 usages
    private int clientsProcessed;

    1 usage   Calina Borzan *
    public Server() {
        this.clients = new LinkedBlockingDeque<>();
        this.waitingPeriod = new AtomicInteger( initialValue: 0);
        this.totalServiceTime=new AtomicInteger( initialValue: 0);
        this.clientsProcessed=0;
        this.endSim = false;

    }

    2 usages   Calina Borzan *
    public void addClient(Client client) {
        if (!client.isServiceStarted()) {
            client.setWaitingTime(this.waitingPeriod.get());
            client.setServiceStarted(true);
            System.out.println("Server - Adding client ID " + client.getId() + " with waiting time: " + clie
        }
        clients.add(client);
        waitingPeriod.addAndGet(client.getServiceTime());
```

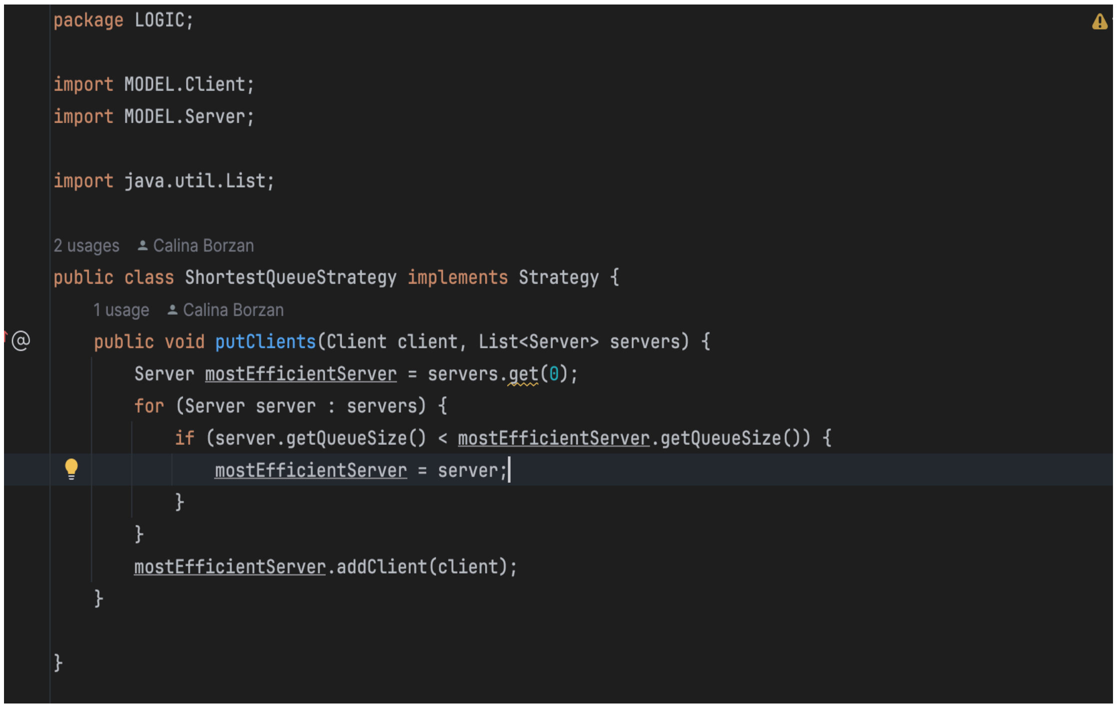


```java
    no usages   Calina Borzan
    @Override
    public Integer call() {
        try {
            while (!endSim) {
                if (!clients.isEmpty()) {
                    Client client = clients.peek();
                    if (client != null) {
                        client.decrementServiceTime();
                        waitingPeriod.decrementAndGet();
                        if (client.getServiceTime() == 0) {
                            clients.poll();
                            totalServiceTime.addAndGet(client.getInitialServiceTime());
                            clientsProcessed++;
                            System.out.println("Server - Client ID " + client.getId() + " completed. Adjuste
                        }
                    }
                }
                Thread.sleep( millis: 1000);
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
        return waitingPeriod.get();
    }


    2 usages   Calina Borzan
    public List<Client> getClients() { return new ArrayList<>(clients); }

    1 usage   Calina Borzan
```

4.3. ShortestQueueStrategy Class

1. **Package:** ‘LOGIC’

2. **Important Methods:**
   - public void putClients(Client client, List<Server> servers): Overrides the putClients method from the Strategy interface. Assigns the incoming client to the server with the shortest queue by comparing the queue sizes of all servers.

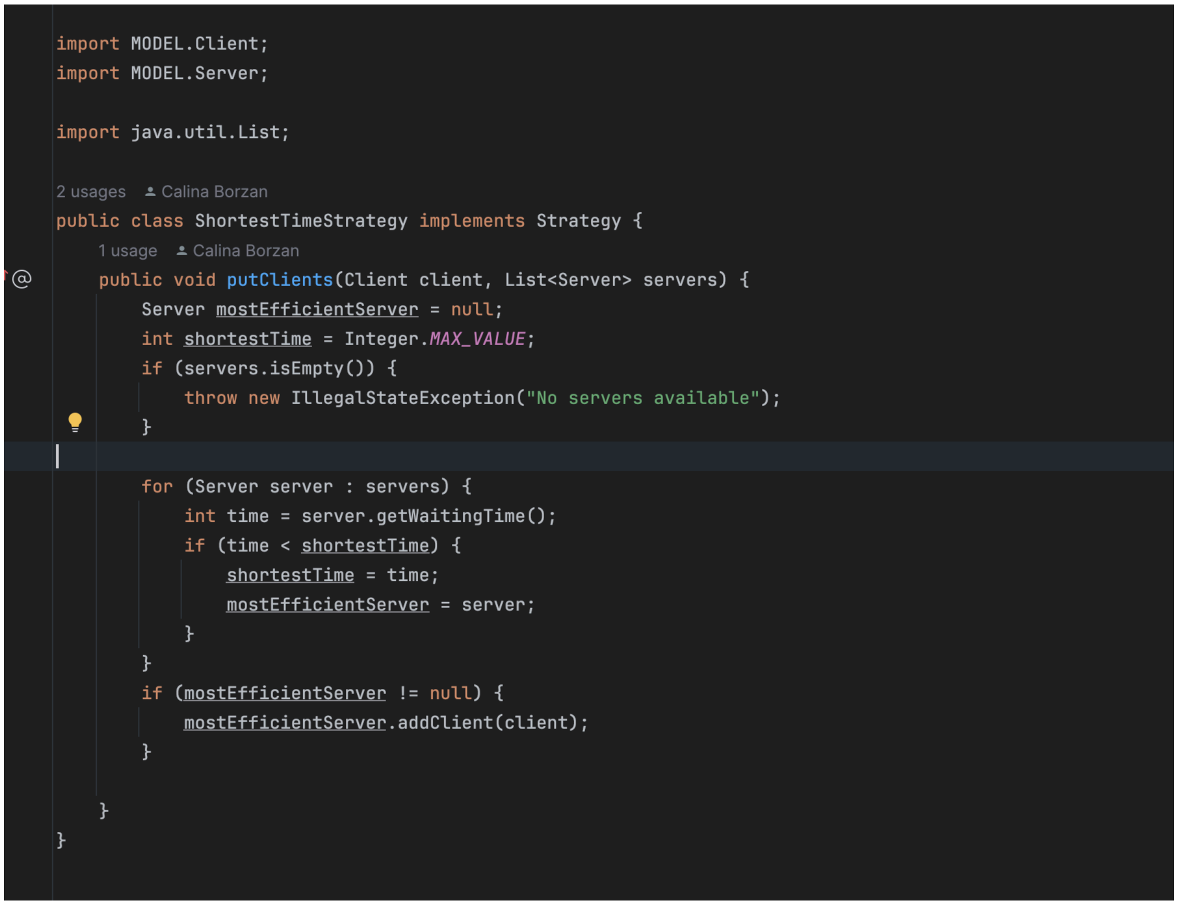


```
package LOGIC;

import MODEL.Client;
import MODEL.Server;

import java.util.List;

2 usages  Calina Borzan
public class ShortestQueueStrategy implements Strategy {
    1 usage  Calina Borzan
    public void putClients(Client client, List<Server> servers) {
        Server mostEfficientServer = servers.get(0);
        for (Server server : servers) {
            if (server.getQueueSize() < mostEfficientServer.getQueueSize()) {
                mostEfficientServer = server;
            }
        }
        mostEfficientServer.addClient(client);
    }

}
```

4.4. ShortestTimeStrategy Class

1. **Package:** ‘LOGIC’

2. **Important Methods:**
   - public void putClients(Client client, List<Server> servers): Overrides the putClients method from the Strategy interface. Assigns the incoming client to the server with the shortest waiting time by comparing the waiting times of all servers.

- 

```java
import MODEL.Client;
import MODEL.Server;

import java.util.List;

2 usages  Calina Borzan
public class ShortestTimeStrategy implements Strategy {
    1 usage  Calina Borzan
    public void putClients(Client client, List<Server> servers) {
        Server mostEfficientServer = null;
        int shortestTime = Integer.MAX_VALUE;
        if (servers.isEmpty()) {
            throw new IllegalStateException("No servers available");
        }

        for (Server server : servers) {
            int time = server.getWaitingTime();
            if (time < shortestTime) {
                shortestTime = time;
                mostEfficientServer = server;
            }
        }
        if (mostEfficientServer != null) {
            mostEfficientServer.addClient(client);
        }

    }
}
```

4.5. Strategy Class

1. **Package:** 'LOGIC'

2. **Important Methods:**
   - void putClients(Client client, List<Server> servers): Defines a method for assigning clients to servers based on a specific strategy.

4.6. SimulationManager Class

1. **Package:** 'LOGIC'

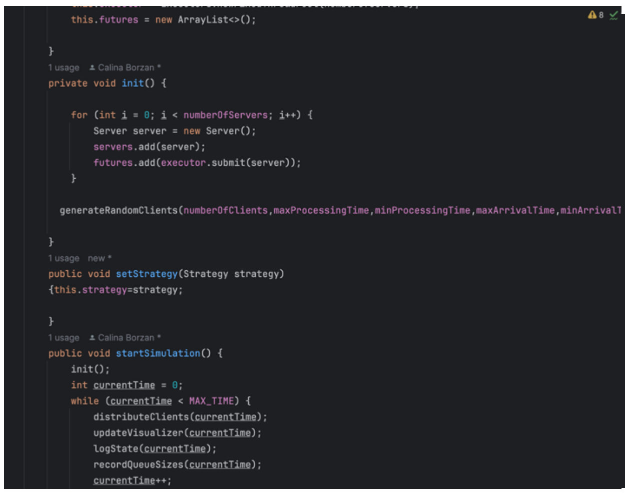
2. **Fields:**
   - public static final String LOG_FILE: Specifies the filename for the simulation log.

- private Map<Integer, Integer> clientArrivalCounts: Maps arrival times to the number of arriving clients.
- private int maxQueueSize: Tracks the maximum size of the server queues during the simulation.
- private List<Client> clientsGenerated: Stores the list of clients generated for the simulation.
- private double averageWaitingTime: Stores the average waiting time of clients in the simulation.
- private double averageServingTime: Stores the average serving time of clients in the simulation.
- private int peakHour: Stores the peak hour of activity during the simulation.
- private List<Future<Integer>> futures: Stores the futures representing server threads.
- public int MAX_TIME: Specifies the maximum simulation time.
- private final int numberOfServers: Specifies the number of servers in the simulation.
- private final JTextArea logArea: Text area for logging simulation events.
- private ExecutorService executor: Manages server threads for parallel execution.
- private List<Server> servers: Stores the list of servers in the simulation.
- private Queue<Client> clients: Stores the queue of clients waiting to be served.
- private Strategy strategy: Stores the strategy for distributing clients to servers.
- public int numberOfClients: Specifies the number of clients in the simulation.
- public int maxProcessingTime: Specifies the maximum processing time for clients.
- public int minProcessingTime: Specifies the minimum processing time for clients.
- public int maxArrivalTime: Specifies the maximum arrival time for clients.
- public int minArrivalTime: Specifies the minimum arrival time for clients.
- private Queues queueVisualizer: Stores the visualizer for server queues.

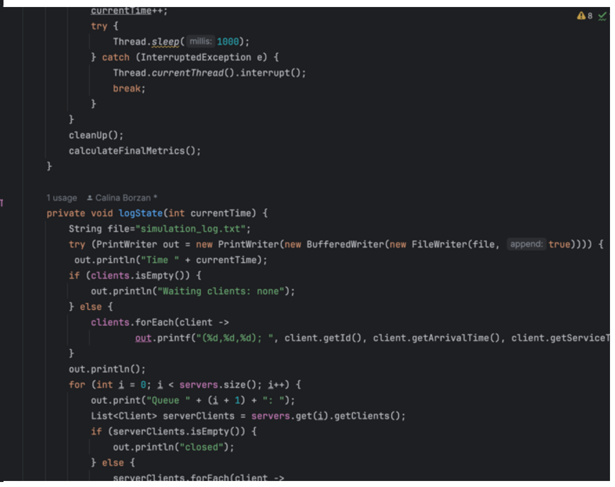
3. Important Methods:

- private void init(): Initializes servers and generates random clients for the simulation.
- public void setStrategy(Strategy strategy): Sets the strategy for client distribution.
- public void startSimulation(): Starts the simulation process.
- private void logState(int currentTime): Logs the state of the simulation at a given time.

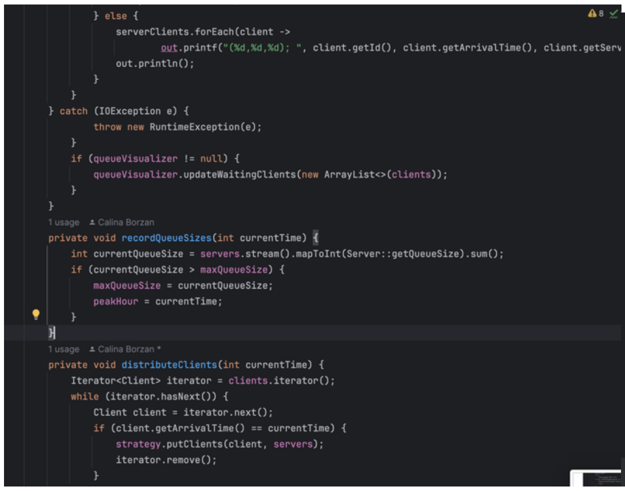
- private void recordQueueSizes(int currentTime): Records the sizes of server queues during the simulation.
- private void distributeClients(int currentTime): Distributes arriving clients to servers based on the current time.
- private void generateRandomClients(int numberOfClients, int maxProcessingTime, int minProcessingTime, int maxArrivalTime, int minArrivalTime): Generates random clients for the simulation.
- private void cleanUp(): Cleans up resources after the simulation ends.
- private void calculateFinalMetrics(): Calculates final performance metrics after the simulation ends.
- public double getAverageWaitingTime(): Returns the average waiting time of clients in the simulation.
- public double getAverageServingTime(): Returns the average serving time of clients in the simulation.
- public int getPeakHour(): Returns the peak hour of activity during the simulation.
- public void setQueueVisualizer(Queues queueVisualizer): Sets the queue visualizer for the simulation.
- private void updateVisualizer(int currentTime): Updates the queue visualizer during the simulation.



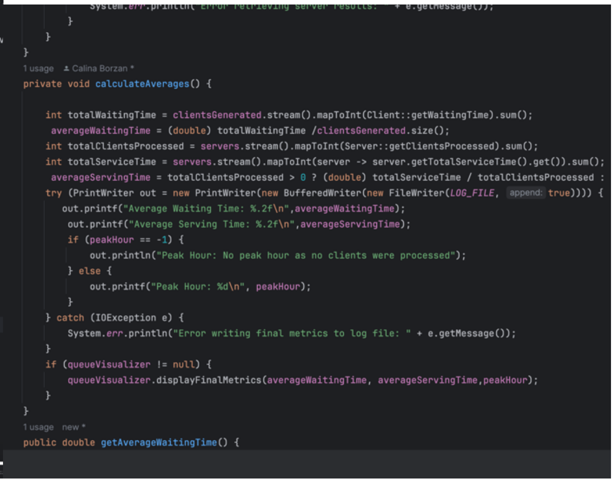
```
        this.futures = new ArrayList<>();

    }
    1 usage   Calina Borzan *
    private void init() {

        for (int i = 0; i < numberOfServers; i++) {
            Server server = new Server();
            servers.add(server);
            futures.add(executor.submit(server));
        }

      generateRandomClients(numberOfClients,maxProcessingTime,minProcessingTime,maxArrivalTime,minArrivalT

    }
    1 usage   new *
    public void setStrategy(Strategy strategy)
    {this.strategy=strategy;

    }
    1 usage   Calina Borzan *
    public void startSimulation() {
        init();
        int currentTime = 0;
        while (currentTime < MAX_TIME) {
            distributeClients(currentTime);
            updateVisualizer(currentTime);
            logState(currentTime);
            recordQueueSizes(currentTime);
            currentTime++;
```



```
            currentTime++;
            try {
                Thread.sleep( millis: 1000);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                break;
            }
        }
        cleanUp();
        calculateFinalMetrics();
    }

    1 usage   Calina Borzan *
    private void logState(int currentTime) {
        String file="simulation_log.txt";
        try (PrintWriter out = new PrintWriter(new BufferedWriter(new FileWriter(file,  append: true)))) {
         out.println("Time " + currentTime);
        if (clients.isEmpty()) {
            out.println("Waiting clients: none");
        } else {
            clients.forEach(client ->
                    out.printf("(%d,%d,%d); ", client.getId(), client.getArrivalTime(), client.getServiceT
        }
        out.println();
        for (int i = 0; i < servers.size(); i++) {
            out.print("Queue " + (i + 1) + ": ");
            List<Client> serverClients = servers.get(i).getClients();
            if (serverClients.isEmpty()) {
                out.println("closed");
            } else {
                serverClients.forEach(client ->
```



```java
            } else {
                serverClients.forEach(client ->
                        out.printf("(%d,%d,%d); ", client.getId(), client.getArrivalTime(), client.getServ
                out.println();
            }
        }
    } catch (IOException e) {
            throw new RuntimeException(e);
        }
        if (queueVisualizer != null) {
            queueVisualizer.updateWaitingClients(new ArrayList<>(clients));
        }
    }
    1 usage   Calina Borzan
    private void recordQueueSizes(int currentTime) {
        int currentQueueSize = servers.stream().mapToInt(Server::getQueueSize).sum();
        if (currentQueueSize > maxQueueSize) {
            maxQueueSize = currentQueueSize;
            peakHour = currentTime;
        }
    }
    1 usage   Calina Borzan *
    private void distributeClients(int currentTime) {
        Iterator<Client> iterator = clients.iterator();
        while (iterator.hasNext()) {
            Client client = iterator.next();
            if (client.getArrivalTime() == currentTime) {
                strategy.putClients(client, servers);
                iterator.remove();
            }
```



```java
                System.err.println("Error retrieving server results: " + e.getMessage());
            }
        }
    }
    1 usage   Calina Borzan *
    private void calculateAverages() {

        int totalWaitingTime = clientsGenerated.stream().mapToInt(Client::getWaitingTime).sum();
         averageWaitingTime = (double) totalWaitingTime /clientsGenerated.size();
        int totalClientsProcessed = servers.stream().mapToInt(Server::getClientsProcessed).sum();
        int totalServiceTime = servers.stream().mapToInt(server -> server.getTotalServiceTime().get()).sum();
         averageServingTime = totalClientsProcessed > 0 ? (double) totalServiceTime / totalClientsProcessed :
        try (PrintWriter out = new PrintWriter(new BufferedWriter(new FileWriter(LOG_FILE,  append: true)))) {
           out.printf("Average Waiting Time: %.2f\n",averageWaitingTime);
            out.printf("Average Serving Time: %.2f\n",averageServingTime);
            if (peakHour == -1) {
                out.println("Peak Hour: No peak hour as no clients were processed");
            } else {
                out.printf("Peak Hour: %d\n", peakHour);
            }
        } catch (IOException e) {
            System.err.println("Error writing final metrics to log file: " + e.getMessage());
        }
        if (queueVisualizer != null) {
            queueVisualizer.displayFinalMetrics(averageWaitingTime, averageServingTime,peakHour);
        }
    }
    1 usage   new *
    public double getAverageWaitingTime() {
```

4.7. SimulationGUI Class

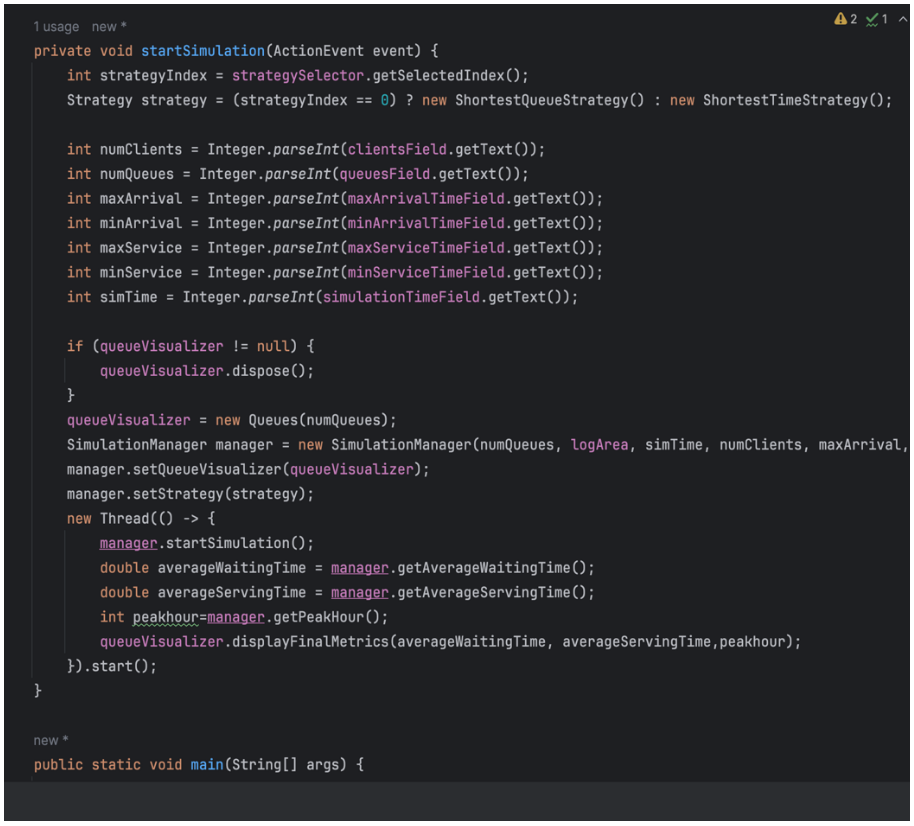
1. **Package:** ‘GUI’
2. **Fields:**

- JTextField clientsField: Represents the text field for entering the number of clients.
- JTextField queuesField: Represents the text field for entering the number of queues.
- JTextField maxArrivalTimeField: Represents the text field for entering the maximum arrival time for clients.
- JTextField minArrivalTimeField: Represents the text field for entering the minimum arrival time for clients.
- JTextField maxServiceTimeField: Represents the text field for entering the maximum service time for clients.
- JTextField minServiceTimeField: Represents the text field for entering the minimum service time for clients.
- JTextField simulationTimeField: Represents the text field for entering the maximum simulation time.
- JComboBox<String> strategySelector: Represents the combo box for selecting the strategy (Shortest Queue or Shortest Waiting Time).
- JTextArea logArea: Represents the text area for displaying simulation logs.
- JButton startButton: Represents the button for starting the simulation.
- Queues queueVisualizer: Represents the visualization panel for displaying queue information during simulation.

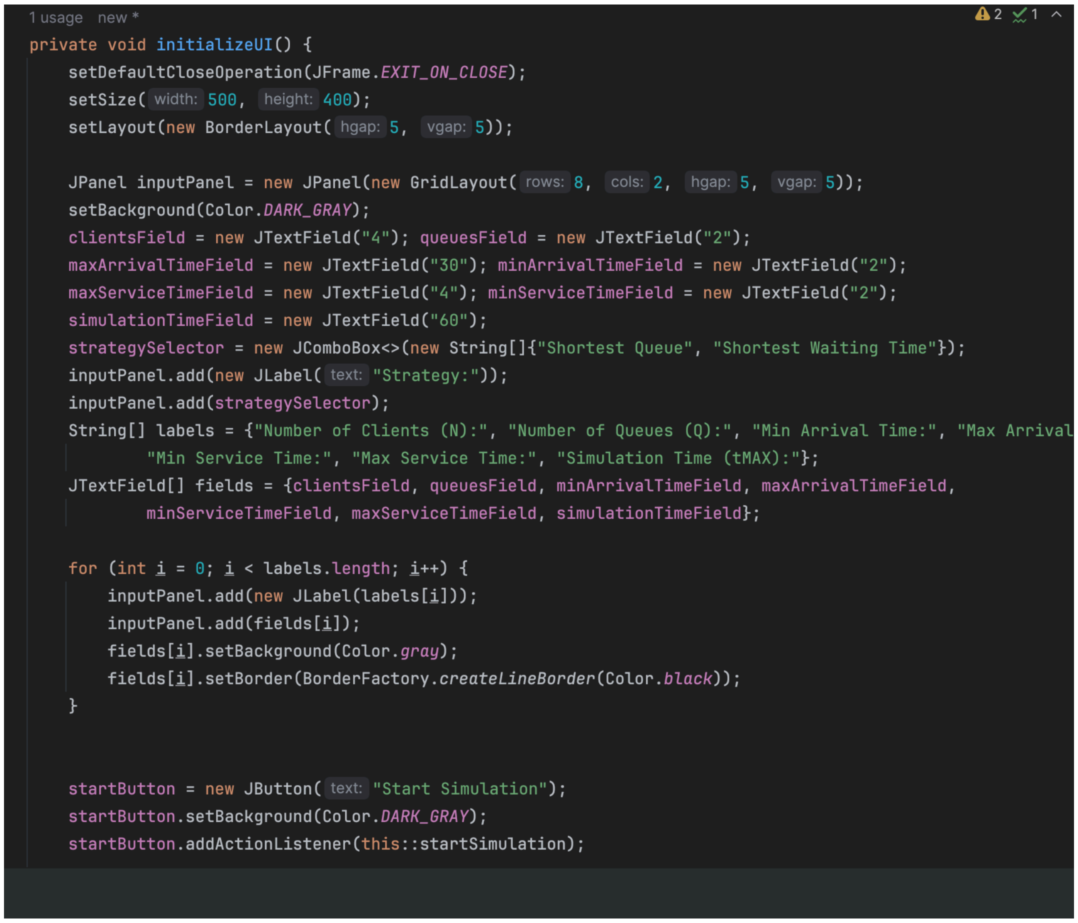
- JLabel finalMetricsLabel: Represents the label for displaying final simulation metrics.
- JLabel timerLabel: Represents the label for displaying the current simulation time.

3. Important Methods:

- initializeUI(): Initializes the user interface components of the GUI, including text fields, labels, buttons, and layout managers.
- startSimulation(ActionEvent event): Event handler for the "Start Simulation" button click event. Reads user input values, determines the selected strategy, and initiates the simulation process using a SimulationManager instance.
- main(String[] args): Entry point for the Java application. Launches the GUI using SwingUtilities.invokeLater() to ensure proper initialization on the Event Dispatch Thread (EDT).



```
1 usage  new *
private void startSimulation(ActionEvent event) {
    int strategyIndex = strategySelector.getSelectedIndex();
    Strategy strategy = (strategyIndex == 0) ? new ShortestQueueStrategy() : new ShortestTimeStrategy();

    int numClients = Integer.parseInt(clientsField.getText());
    int numQueues = Integer.parseInt(queuesField.getText());
    int maxArrival = Integer.parseInt(maxArrivalTimeField.getText());
    int minArrival = Integer.parseInt(minArrivalTimeField.getText());
    int maxService = Integer.parseInt(maxServiceTimeField.getText());
    int minService = Integer.parseInt(minServiceTimeField.getText());
    int simTime = Integer.parseInt(simulationTimeField.getText());

    if (queueVisualizer != null) {
        queueVisualizer.dispose();
    }
    queueVisualizer = new Queues(numQueues);
    SimulationManager manager = new SimulationManager(numQueues, logArea, simTime, numClients, maxArrival,
    manager.setQueueVisualizer(queueVisualizer);
    manager.setStrategy(strategy);
    new Thread(() -> {
        manager.startSimulation();
        double averageWaitingTime = manager.getAverageWaitingTime();
        double averageServingTime = manager.getAverageServingTime();
        int peakhour=manager.getPeakHour();
        queueVisualizer.displayFinalMetrics(averageWaitingTime, averageServingTime,peakhour);
    }).start();
}

new *
public static void main(String[] args) {
```



```
1 usage   new *
private void initializeUI() {
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setSize( width: 500,  height: 400);
    setLayout(new BorderLayout( hgap: 5,  vgap: 5));

    JPanel inputPanel = new JPanel(new GridLayout( rows: 8,  cols: 2,  hgap: 5,  vgap: 5));
    setBackground(Color.DARK_GRAY);
    clientsField = new JTextField("4"); queuesField = new JTextField("2");
    maxArrivalTimeField = new JTextField("30"); minArrivalTimeField = new JTextField("2");
    maxServiceTimeField = new JTextField("4"); minServiceTimeField = new JTextField("2");
    simulationTimeField = new JTextField("60");
    strategySelector = new JComboBox<>(new String[]{"Shortest Queue", "Shortest Waiting Time"});
    inputPanel.add(new JLabel( text: "Strategy:"));
    inputPanel.add(strategySelector);
    String[] labels = {"Number of Clients (N):", "Number of Queues (Q):", "Min Arrival Time:", "Max Arrival
            "Min Service Time:", "Max Service Time:", "Simulation Time (tMAX):"};
    JTextField[] fields = {clientsField, queuesField, minArrivalTimeField, maxArrivalTimeField,
            minServiceTimeField, maxServiceTimeField, simulationTimeField};

    for (int i = 0; i < labels.length; i++) {
        inputPanel.add(new JLabel(labels[i]));
        inputPanel.add(fields[i]);
        fields[i].setBackground(Color.gray);
        fields[i].setBorder(BorderFactory.createLineBorder(Color.black));
    }


    startButton = new JButton( text: "Start Simulation");
    startButton.setBackground(Color.DARK_GRAY);
    startButton.addActionListener(this::startSimulation);
```

4.8. Queues Class

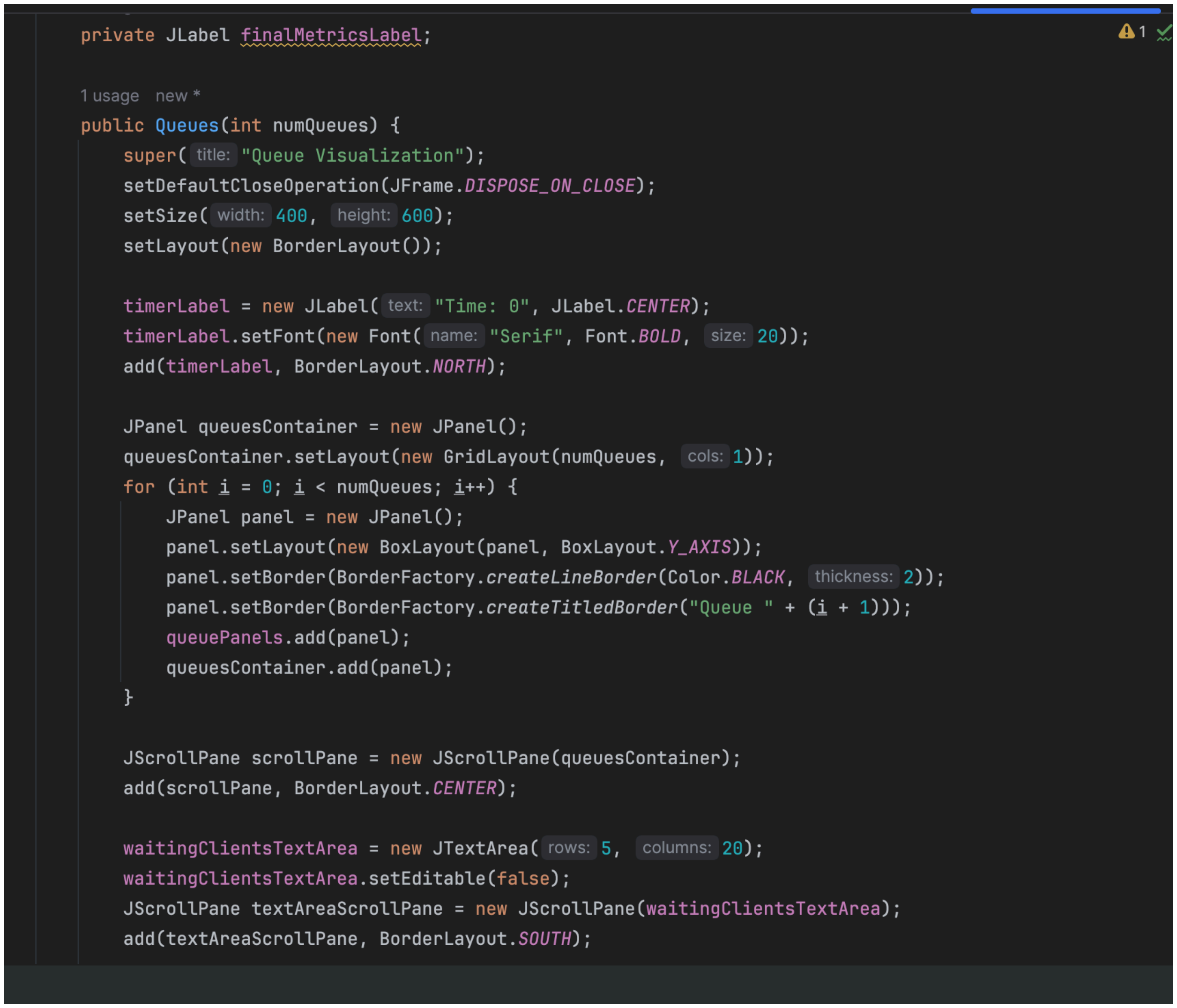
1. **Package**: ‘GUI’

2. **Fields:**

- List<JPanel> queuePanels: Represents a list of panels, each corresponding to a queue in the simulation.
- JLabel timerLabel: Represents the label for displaying the current simulation time.
- JTextArea waitingClientsTextArea: Represents the text area for displaying information about waiting clients.
- JLabel finalMetricsLabel: Represents the label for displaying final simulation metrics.

3. Important Methods:

- Queues(int numQueues): Constructor method for initializing the Queues object. Creates the visualization interface with the specified number of queue panels.
- updateQueue(int queueIndex, List<Client> clients, int currentTime): Updates the specified queue panel with information about the clients currently in the queue and the current simulation time.
- displayFinalMetrics(double averageWaitingTime, double averageServingTime, int peakHour): Displays the final simulation metrics, including average waiting time, average serving time, and peak hour.
- updateWaitingClients(List<Client> clients): Updates the text area with information about waiting clients, including their arrival time and service time.
- setDefaultCloseOperation(int operation): Overrides the setDefaultCloseOperation method from the JFrame class to specify the action to be taken when the frame is closed.
- setSize(int width, int height): Overrides the setSize method from the JFrame class to set the size of the frame.
- setLayout(LayoutManager manager): Overrides the setLayout method from the Container class to set the layout manager for the frame's content pane.
- add(Component comp, Object constraints): Overrides the add method from the Container class to add a component to the frame's content pane with the specified constraints.
- setLocationRelativeTo(Component c): Overrides the setLocationRelativeTo method from the Window class to set the location of the frame relative to the specified component.
- setVisible(boolean b): Overrides the setVisible method from the Component class to set the visibility of the frame.



```java
    private JLabel finalMetricsLabel;

    1 usage   new *
    public Queues(int numQueues) {
        super( title: "Queue Visualization");
        setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        setSize( width: 400,  height: 600);
        setLayout(new BorderLayout());

        timerLabel = new JLabel( text: "Time: 0", JLabel.CENTER);
        timerLabel.setFont(new Font( name: "Serif", Font.BOLD,  size: 20));
        add(timerLabel, BorderLayout.NORTH);

        JPanel queuesContainer = new JPanel();
        queuesContainer.setLayout(new GridLayout(numQueues,  cols: 1));
        for (int i = 0; i < numQueues; i++) {
            JPanel panel = new JPanel();
            panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
            panel.setBorder(BorderFactory.createLineBorder(Color.BLACK,  thickness: 2));
            panel.setBorder(BorderFactory.createTitledBorder("Queue " + (i + 1)));
            queuePanels.add(panel);
            queuesContainer.add(panel);
        }

        JScrollPane scrollPane = new JScrollPane(queuesContainer);
        add(scrollPane, BorderLayout.CENTER);

        waitingClientsTextArea = new JTextArea( rows: 5,  columns: 20);
        waitingClientsTextArea.setEditable(false);
        JScrollPane textAreaScrollPane = new JScrollPane(waitingClientsTextArea);
        add(textAreaScrollPane, BorderLayout.SOUTH);
```

5. Results

In this section of my documentation I will present the testing scenarios that I conducted as to be sure that my program is correct.

For the first case when I have as inputs: 4 clients, 2 queues, time max simulation of 60 seconds, time minim for arrival 2 and maxim 30, and time minim of service 2 and maxim 4 and applying the shortest queue strategy. The log of events for this case in my application is in my programm in a text file named ‘example1.txt’.

For the second case when I have as inputs: 50 clients, 5 queues, time max simulation of 60 seconds, time minim for arrival 2 and maxim 40, and time minim of service 1 and maxim 7 and applying the shortest queue strategy. The log of events for this case in my application is in my programm in a text file named ‘example2.txt’.

For the third case when I have as inputs: 1000 clients, 50 queues, time max simulation of 200 seconds, time minim for arrival 10 and maxim 100, and time minim of service 3 and maxim 9 and applying the shortest queue strategy. The log of events for this case in my application is in my programm in a text file named ‘example3.txt’.

6. Conclusions

Embarking on the journey to develop a simulation manager has been both an enriching and challenging experience, broadening my understanding of software development principles and simulation modeling. Here are the key takeaways, learnings, and potential avenues for future enhancements.

What I've Learned:

Simulation Modeling: The process of designing and implementing a simulation manager has deepened my understanding of simulation modeling principles, including event-driven simulation and queueing theory. It has also highlighted the importance of accurately modeling real-world systems to derive meaningful insights.

Concurrency and Multithreading: Developing a simulation manager involves managing multiple concurrent processes efficiently. This project has provided valuable experience in implementing multithreaded applications and handling synchronization to ensure thread safety.

User Interface Design: Integrating a graphical user interface (GUI) into the simulation manager has underscored the importance of intuitive design and user experience. Balancing functionality with simplicity is essential for user adoption and engagement.

Future Development:

Looking ahead, there are several potential directions to enhance the simulation manager:

- Advanced Simulation Features: Incorporating advanced simulation features such as priority-based scheduling, resource allocation, and dynamic event generation would make the simulation manager more versatile and applicable to a broader range of scenarios.

- Visualization Enhancements: Enhancing the visualization capabilities of the simulation manager, such as integrating real-time charts and graphs, would provide users with more insights into simulation dynamics and results.

- Parameterization and Customization: Offering options for parameterization and customization of simulation models would enable users to tailor simulations to specific scenarios and explore "what-if" scenarios more effectively.

- Performance Optimization: Continuously optimizing the simulation algorithms and data structures to improve performance, scalability, and responsiveness, especially for large-scale simulations, is essential for ensuring the simulation manager remains efficient and practical.

Overall, developing a simulation manager has been a rewarding experience, providing insights into complex system dynamics and the intricacies of simulation modeling. As the project evolves, I look forward to implementing these enhancements to create a more robust and versatile simulation tool.

7. Bibliography

The references I while implementing this program:

1. *https://docs.oracle.com/javase/tutorial/essential/concurrency/index.htmlWhat are Java classes?*

2. *http://www.tutorialspoint.com/java/util/timer_schedule_period.htm*

3. *http://www.javacodegeeks.com/2013/01/java-thread-pool-example-using-executors-and-threadpoolexecutor.html*

4. *https://www.baeldung.com/java-concurrency*