DOCUMENTATION

ASSIGNMENT 2

STUDENT NAME: Tudorache Ioan
GROUP:30226

CONTENTS

1. Assignment Objective

- **Main objective**

✓ Design and implement an application aiming to analyze queuing-based systems by simulating a series of N clients arriving for service, entering Q queues, waiting, being served and finally leaving the queues, and computing the average waiting time, average service time and peak hour.

- **Sub-objectives**

✓ **Analyze the problem and identify requirements.**

- Understand the significance of a simulated queue system and gather requirements from stakeholders. Categorize requirements into functional (what the system should do) and non-functional (qualities the system should possess).

✓ **Design the simulation application.**

- Create a blueprint for the system, including the user interface design, system architecture, and algorithms for simulating queuing-based systems. Ensure the design is scalable, modular, and user-friendly.

✓ **Implement the simulation application.**

- Translate the design into code using programming languages and development tools. Implement algorithms for client arrival, queue management, service, and departure.
- Develop modules for calculating average waiting time, average service time, and peak hour.
- Follow coding standards and best practices.

✓ **Test the simulation application.**

- Run simulations with different parameters to validate the accuracy of the results.

2. Problem Analysis, Modeling, Scenarios, Use Cases

- **Functional requirements:**

➢ The simulation application should enable users to configure parameters such as the number of queues, arrival rates, service rates, and simulation duration.

➢ The simulation application should provide options for users to choose different queuing disciplines such as Shortest Queue or Shortest Waiting Time.

- The simulation application should generate statistical reports at the end of each simulation run, including average waiting time, average service time, and peak hour analysis.

- The simulation application should support visualization of queue dynamics including the movement of clients through the queues.

- The simulation application should allow users to save simulation configurations for future use.

- The simulation application should provide error handling mechanisms to handle unexpected inputs or system failures gracefully.

- **Non-Functional requirements:**

  - The simulation application should be responsive, providing smooth interaction and feedback to users even with large-scale simulations.

  - The simulation application should be scalable, capable of handling increasing numbers of clients and queues without significant degradation in performance.

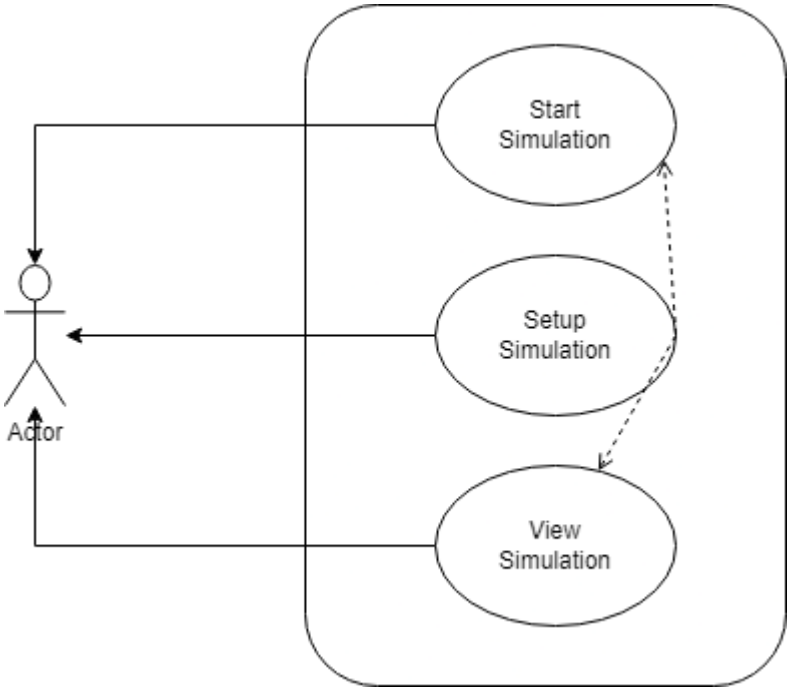
  - The simulation application should be robust, able to recover gracefully from errors and resume simulations without data loss.

  - The simulation application should be well-documented, providing comprehensive user guides and technical documentation for users and developers.

  - The simulation application should be efficient in its resource usage, minimizing memory and processing requirements to optimize performance.

  - The simulation application should be maintainable, designed with clean and modular code that facilitates future updates and enhancements.

- **Use Case Diagram:**



Use Case Description:

Use case: Setup Simulation

Primary Actor: User **Main**

Success Scenario:

1) The user inserts the values for the: number of clients, number of queues, simulation interval, minimum and maximum arrival time, and minimum and maximum service time, checks the box for Shortest Queue or Shortest Waiting Time.

2) The user clicks on the “Enter” data button.

3) The application validates the data and initiates the simulation.

Alternative Sequence: Invalid values for the setup parameters

- The user inserts invalid values for the application’s setup parameters.

- The application displays an error message and requests the user to insert valid values.

- The scenario returns to step 1.

Use case: Start Simulation

Primary Actor: User **Main**

Success Scenario:

1) Upon receiving confirmation from the user by pressing the "Enter" button in the interface, the system initializes the simulation with the specified parameters.

2) Clients begin to arrive, enter queues, get served, and depart based on the defined parameters.

3) The simulation progresses in real-time.

Alternative Sequence: User does not confirm the parameters by pressing the "Enter" button.

- The system does not initiate the simulation.

- The user remains in the setup phase until parameters are confirmed and simulation is started.

Use case: View Simulation

Primary Actor: User **Main**

Success Scenario:

1) The system automatically starts the simulation process upon receiving confirmation from the user by pressing the "Enter" button.

2) The simulation application displays real-time queue lengths and dynamics through a graphical interface.

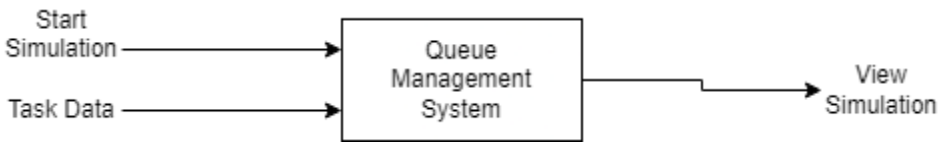
3) The user observes the movement of clients through the queues, as well as any changes in queue lengths.

Alternative Sequence: Simulation Stopped

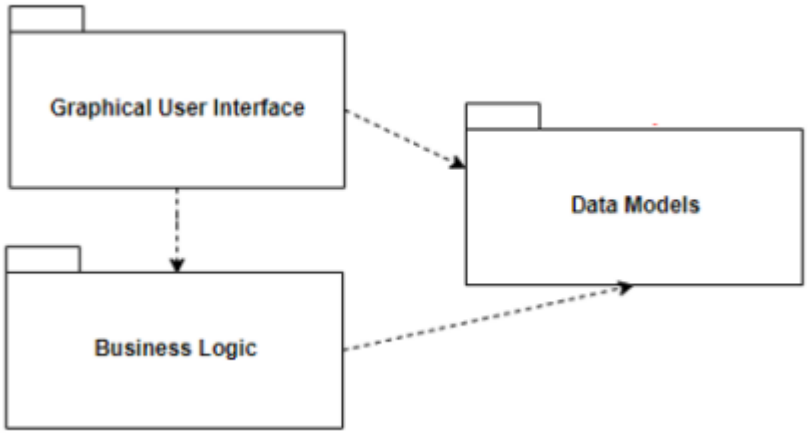
- The system stops the simulation entirely.
- The user may review simulation or initiate a new simulation if desired.

3. Design:

- **Black Box:**



- **Packages Diagram:**



- In our project's context, the three-tier architecture encapsulates the different aspects of our Queue Management System within the Graphical User Interface (GUI), Business Logic, and Data Models layers:

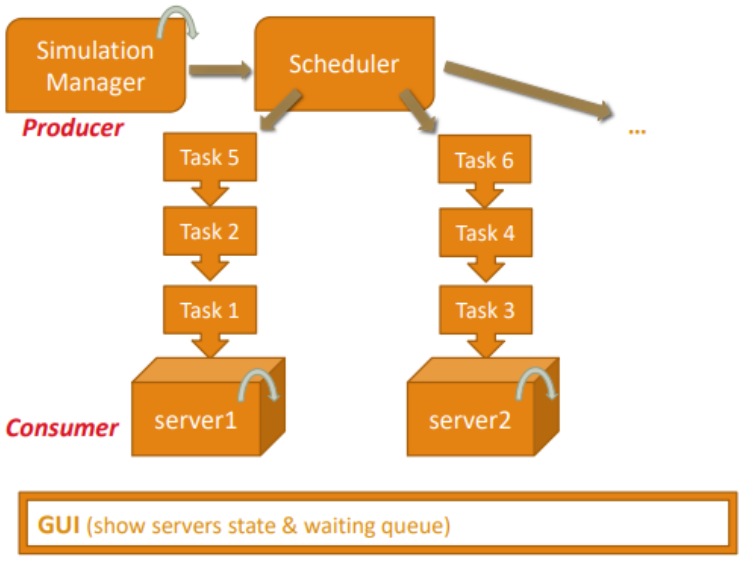
  - **Graphical User Interface (GUI):** Within our project, the 'SimulationFrame' class populates this layer. It handles all user

interactions, presenting controls and information through components such as text fields, buttons, and labels. The GUI is responsible for taking user inputs for simulation parameters and displaying the simulation's results in an understandable format.

- **Business Logic**: Our 'Scheduler', 'Server', and 'Strategy' classes, along with their derivatives like 'ShortestQueueStrategy', form the Business Logic layer. This layer implements the core scheduling algorithms, manages task distribution, and ensures that tasks are processed in an efficient manner. The 'SimulationManager' class also resides here, as it orchestrates the overall flow of the simulation, integrating user input from the GUI with the system's processing logic.

- **Data Models:** The 'Task' class represents the Data Models layer. It defines the data structure for tasks within our system, including attributes like ID, service time, and arrival time. This layer is crucial as it standardizes how tasks are represented and managed throughout the application, ensuring that data is consistent and reliable when passed between the Business Logic and the GUI.

- **Conceptual Architecture:**



➢ The diagram depicts a task scheduling system for simulations. The "Simulation Manager" creates tasks, and the "Scheduler" assigns them to servers labeled as

"server1" and "server2." These servers process the tasks in a queue. A "GUI" component displays the server statuses and the queue of pending tasks. This setup optimizes task distribution across servers for efficient processing.

- **UML Diagram:**

- Our UML diagram outlines a sophisticated Queue Management System comprising multiple classes, each with distinct responsibilities and connections. Here are the relationships between them:
- **Task ↔ Server**: The Server class has a composition relationship with the Task class. This indicates that Server instances manage the lifecycle of Task objects within their taskQueue. Tasks are not shared between Servers; each Server instance owns its Task objects.

- **Server ↔ Scheduler**: The Scheduler class aggregates Server instances. This means that Scheduler controls a collection of Server objects, which are independent and may exist outside the Scheduler's context.

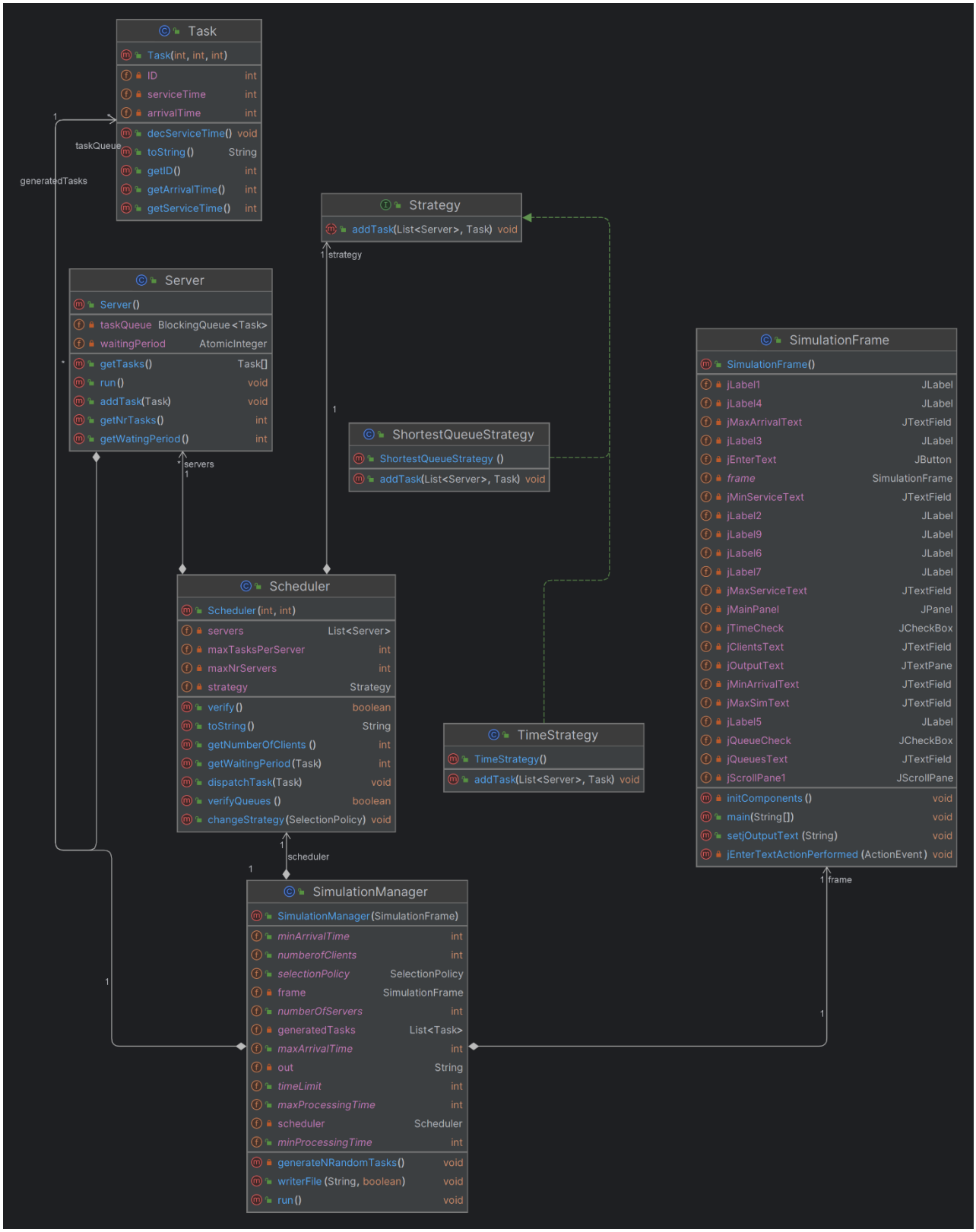
- **Strategy ↔ Server**: There is a dependency relationship from the Strategy interface to the Server class. Strategy implementations require knowledge about Server instances to add tasks appropriately.

- **ShortestQueueStrategy ↔ Strategy**: ShortestQueueStrategy implements the Strategy interface, thus forming a realization relationship. It provides a concrete strategy by overriding the addTask method to assign tasks to the server with the shortest queue.

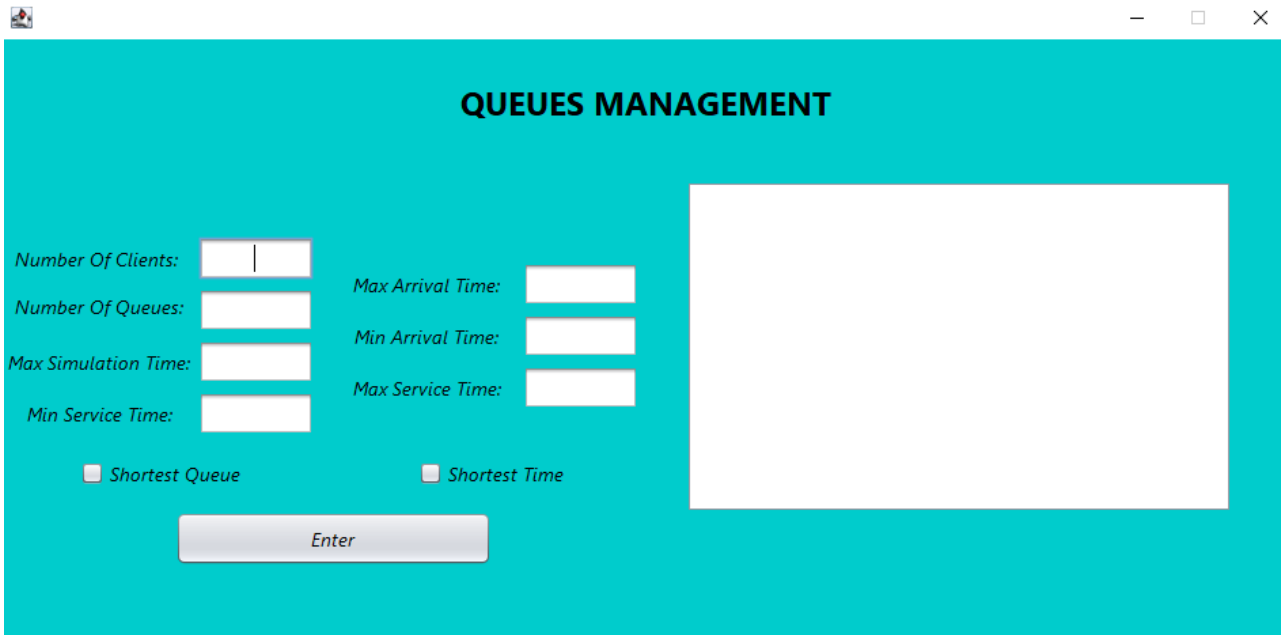
- **Scheduler ↔ Strategy**: Scheduler uses a Strategy object, demonstrating a relationship of association. The Scheduler delegates the task of adding tasks to servers to whichever Strategy instance it currently holds, allowing for flexible task distribution logic.

- **TimeStrategy ↔ Strategy**: While not directly connected on the diagram, it's presumed that TimeStrategy is another Strategy implementation, thus it would share a realization relationship with the Strategy interface similar to ShortestQueueStrategy.

- This diagram effectively utilizes UML relationships to illustrate object-oriented principles such as encapsulation, inheritance, and polymorphism, facilitating a flexible and maintainable system design.


Task
Task(int, int, int)
ID int
serviceTime int
arrivalTime int
decServiceTime() void
toString() String
getID() int
getArrivalTime() int
getServiceTime() int
taskQueue
generatedTasks
Strategy
addTask(List<Server>, Task) void
strategy
Server
Server()
taskQueue BlockingQueue<Task>
waitingPeriod AtomicInteger
getTasks() Task[]
run() void
addTask(Task) void
getNrTasks() int
getWatingPeriod() int
servers
ShortestQueueStrategy
ShortestQueueStrategy()
addTask(List<Server>, Task) void
SimulationFrame
SimulationFrame()
jLabel1 JLabel
jLabel4 JLabel
jMaxArrivalText JTextField
jLabel3 JLabel
jEnterText JButton
frame SimulationFrame
jMinServiceText JTextField
jLabel2 JLabel
jLabel9 JLabel
jLabel6 JLabel
jLabel7 JLabel
jMaxServiceText JTextField
jMainPanel JPanel
jTimeCheck JCheckBox
jClientsText JTextField
jOutputText JTextPane
jMinArrivalText JTextField
jMaxSimText JTextField
jLabel5 JLabel
jQueueCheck JCheckBox
jQueuesText JTextField
jScrollPane1 JScrollPane
initComponents() void
main(String[]) void
setjOutputText(String) void
jEnterTextActionPerformed(ActionEvent) void
Scheduler
Scheduler(int, int)
servers List<Server>
maxTasksPerServer int
maxNrServers int
strategy Strategy
verify() boolean
toString() String
getNumberOfClients() int
getWaitingPeriod(Task) int
dispatchTask(Task) void
verifyQueues() boolean
changeStrategy(SelectionPolicy) void
TimeStrategy
TimeStrategy()
addTask(List<Server>, Task) void
scheduler
frame
SimulationManager
SimulationManager(SimulationFrame)
minArrivalTime int
numberofClients int
selectionPolicy SelectionPolicy
frame SimulationFrame
numberOfServers int
generatedTasks List<Task>
maxArrivalTime int
out String
timeLimit int
maxProcessingTime int
scheduler Scheduler
minProcessingTime int
generateNRandomTasks() void
writerFile(String, boolean) void
run() void

- **Graphic Design:**
    - **Queues Management System Interface:**
        - The "Queues Management" interface is part of a simulation software designed to manage and analyze queuing systems. Users can input the number of clients, queues, and set parameters for arrival and service times. The system offers options to prioritize by the shortest queue or time, which likely helps in determining the most efficient queuing strategy. The large white space to the right is meant to display real-time simulation results of the queuing process. The interface is a practical tool for optimizing customer service operations.



- **Data Structure:**
    - **BlockingQueue<Task>:** This is used in the Server class to hold tasks in a thread-safe way. It ensures that tasks are processed in a FIFO (First-In-First-Out) manner and handles concurrency well, allowing tasks to be safely added and removed by different threads.

- **AtomicInteger**: The waitingPeriod in the Server class is an AtomicInteger, which provides a way to atomically update the counter. It is thread-safe and used here to keep track of the total waiting time for all tasks in the queue.

- **ArrayList<Task>:** In the SimulationManager, an ArrayList is used to store the generated tasks. This data structure allows for dynamic array management, resizing itself as needed when new tasks are added.

- **Comparator<Task>:** The generateNRandomTasks method in SimulationManager uses a Comparator to sort tasks based on their arrival time. This is a functional interface used for comparing objects, allowing the tasks to be sorted in a priority queue fashion if needed.

- **Concurrent Data Structures:** Although not directly shown in the snippets provided, the use of LinkedBlockingDeque and AtomicInteger suggests that concurrent data structures are used to handle multi-threaded access to the data, ensuring thread safety and consistency of the state.

4. Implementation

- **The Queue Management System's** effectiveness hinges on a few pivotal classes, each serving a critical role in the system's operation. Let's delve into these classes:

- ✓ **Task Class:** The foundational element of the system, each Task object represents a discrete unit of work to be processed. With attributes for identification, service time, and arrival time, the Task class encapsulates all the necessary information a server might need to process it. The methods provided allow for easy retrieval and updates to the task's state, which is crucial for dynamic scheduling during runtime.

- ✓ **Server Class**: This class is the workhorse of the system, simulating a real-world server's operations. It maintains a task queue and uses atomic integers to handle concurrent processing securely. Servers are responsible for executing tasks, which involve tracking their waiting period, a critical metric for performance analysis. The Server class's design allows it to work independently or as part of a pool managed by the Scheduler.

```
public void addTask(Task newTask) {
    taskLock.lock();
    try {
        tasks.add(newTask);
        int newWaitingTime = waitingPeriod.addAndGet(newTask.getServiceTime());
        if (newWaitingTime < 0) {
            throw new IllegalStateException("Waiting period should never go below zero.");
        }
    } finally {
        taskLock.unlock();
    }
}

@Override
public void run() {
    while (!Thread.currentThread().isInterrupted()) {
        taskLock.lock();
        try {
            Task currentTask = tasks.peek();
            if (currentTask != null) {
                currentTask.decServiceTime();
                int newWaitingTime = waitingPeriod.decrementAndGet();
                if (newWaitingTime < 0) {
                    throw new IllegalStateException("Waiting period should never go below zero.");
                }

                if (currentTask.getServiceTime() <= 0) {
                    tasks.poll();
                }
            }
        } finally {
            taskLock.unlock();
        }

        try {
            Thread.sleep( millis: 1000);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
```

- ✓ **Scheduler Class**: As the orchestrator, the Scheduler class allocates tasks to servers based on the chosen strategy. It maintains a list of servers, ensuring that the workload is distributed according to predefined rules, whether it be the shortest queue, least processing time, or any other strategy that implements the Strategy interface. The ability to verify queues and change strategies on the fly provides the flexibility needed to adapt to varying operational conditions.

```
8 usages
private List<Server> servers;
no usages
private int maxNrServers;
no usages
private int maxTasksPerServer;
3 usages
private Strategy strategy;

1 usage
public Scheduler(int maxNrServers,int maxTasksPerServer){
    servers = new ArrayList<>();
    for(int i = 0; i < maxNrServers;i++){
        Server ser = new Server();
        servers.add(ser);
        Thread serverThread = new Thread(ser);
        serverThread.start();
    }
}
1 usage
public void changeStrategy(SelectionPolicy policy){

    if(policy == SelectionPolicy.SHORTEST_QUEUE){
        strategy = new ShortestQueueStrategy();
    }
    else if(policy == SelectionPolicy.SHORTEST_TIME){
        strategy = new TimeStrategy();
    }
}

1 usage
public synchronized void dispatchTask(Task task) throws InterruptedException {
    strategy.addTask(servers,task);
}
6 usages
public enum SelectionPolicy{
    2 usages
    SHORTEST_TIME,SHORTEST_QUEUE;
}
```

- ✓ **Strategy Interface and Implementations**: The Strategy interface is central to the system's extensibility. It outlines the contract for adding tasks to servers, allowing for multiple concrete implementations. For example, the ShortestQueueStrategy class prioritizes task assignment to the server with the least number of pending tasks, optimizing for shorter wait times. The design allows new strategies to be introduced with minimal impact on existing code.

```java
public class ShortestQueueStrategy implements Strategy{


    1 usage
    @Override
    public void addTask(List<Server> servers, Task task) throws InterruptedException {
        if (servers.isEmpty()) {
            return;
        }

        Server firstServer = servers.get(0);
        if (firstServer == null) {
            throw new IllegalStateException("No servers available to add the task.");
        }

        Server saveServer = firstServer;
        int sizeMin = firstServer.getTasks().length;

        for (int i = 1; i < servers.size(); i++) {
            Server currentServer = servers.get(i);
            int sizeServer = currentServer.getTasks().length;
            if (sizeServer < sizeMin) {
                sizeMin = sizeServer;
                saveServer = currentServer;
            }
        }
        saveServer.addTask(task);
    }
}
```

```java
public class TimeStrategy implements Strategy {

    1 usage
    public void addTask(List<Server> servers, Task task) throws InterruptedException {
        Server firstServer;
        if (servers.isEmpty()) {
            firstServer = null;
        } else {
            firstServer = servers.get(0);
        }
        if (firstServer == null) {
            throw new IllegalStateException("No servers available to add the task.");
        }

        Server saveServer = firstServer;
        int timeMin = firstServer.getWatingPeriod();

        for (int i = 1; i < servers.size(); i++) {
            Server currentServer = servers.get(i);
            int timeServer = currentServer.getWatingPeriod();
            if (timeServer < timeMin) {
                timeMin = timeServer;
                saveServer = currentServer;
            }
        }
        saveServer.addTask(task);
    }
}
```

- ✓ **SimulationManager Class**: This class serves as the command center, linking the user interface with the backend logic. It manages simulation parameters such as arrival times and the number of clients, and it controls the simulation's execution. The SimulationManager uses the Scheduler to distribute tasks and can change scheduling strategies as needed. It is responsible for the overall coordination of the simulation, ensuring that the system behaves as expected.

```java
public SimulationManager(SimulationFrame frame){
    this.frame = frame;
    scheduler = new Scheduler(numberOfServers,numberofClients);
    scheduler.changeStrategy(selectionPolicy);
    generatedTasks = new ArrayList<>();
    generateNRandomTasks();

}


1 usage
private void generateNRandomTasks(){
    int processingTime;
    int arrivalTime;
    Random random = new Random();
    for(int i = 0; i < numberofClients; i++){
        processingTime = random.nextInt( bound: maxProcessingTime - minProcessingTime) + minProcessingTime;
        arrivalTime = random.nextInt( bound: maxArrivalTime-minArrivalTime) + minArrivalTime;
        generatedTasks.add(new Task( ID: i+1,arrivalTime,processingTime));
    }
    Comparator<Task> timeComparator = Comparator.comparing(Task::getArrivalTime);
    generatedTasks.sort(timeComparator);

}
3 usages
public void writerFile(String output,boolean app){
    String fName = "output.txt";
    try {
        FileWriter fWriter = new FileWriter(fName,app);
        BufferedWriter bWriter = new BufferedWriter(fWriter);
        bWriter.write(output);
        bWriter.close();
    } catch (IOException e) {
        System.err.println("Error writing to the file: " + e.getMessage());
    }
}
```

5. Results:

- ✓ **In our results section**, we compare the outcomes of three distinct test scenarios with varying parameters:

  - Test 1 with N=4 servers and Q=2 queues has a maximum simulation time of 60 seconds, arrival times between 2 and 30 seconds, and service times between 2 and 4 seconds.

```
Finished
Peak Time: 12
Average Service Time: 2.75
Average Waiting Time: 2.75
```

  - Test 2 scales up with N=50 servers and Q=5 queues, maintaining the same simulation cap of 60 seconds but widening the arrival time window to between 2 and 40 seconds and service times between 1 and 7 seconds.

```
Finished
Peak Time: 12
Average Service Time: 4.10
Average Waiting Time: 6.62
```

  - Test 3, our most extensive, utilizes N=1000 servers and Q=20 queues, extending the maximum simulation time to 200 seconds, arrival times span from 10 to 100 seconds, and service times range from 3 to 9 seconds, providing a comprehensive assessment across small to large-scale simulations.

```
Finished
Peak Time: 99
Average Service Time: 5.56
Average Waiting Time: 97.71
```

If you want to see the full results, check the repository where the log files are.

6. Conclusions

What I Have Learned from the Assignment:

1. **Requirements Analysis:** Properly understanding and analyzing the requirements of a software project is crucial for its success. Through this assignment, I learned the importance of capturing both functional and non-functional requirements to ensure that the final product meets the needs of its users.

- **Concurrency Management**: The use of BlockingQueue and AtomicInteger demonstrates how to handle concurrency in Java, ensuring that multiple threads can operate on shared data without causing inconsistencies.

- **Object-Oriented Design**: The program's structure, with distinct classes for servers, tasks, and the simulation manager, showcases the principles of object-oriented design, promoting modularity and reusability.

- **Event-Driven Simulation**: The way tasks are processed according to their arrival and service times illustrates an event-driven approach, which is a cornerstone of discrete-event simulation modeling.

- **Data Structures Utilization**: The choice of data structures like ArrayList and BlockingQueue for task management reveals their importance in developing efficient algorithms.

- **Performance Analysis**: Writing simulation results to a file for post-processing is an essential practice in performance analysis, allowing you to understand and improve system behavior over time.

Future Developments:

- **Graphical Visualization**: Enhancing the GUI to include real-time charts and graphs could provide a more intuitive understanding of the system's performance and behavior.

- **Optimization Algorithms**: Implementing and comparing different scheduling algorithms would make the simulator a more powerful tool for optimizing queuing systems.

- **Distributed Computing**: Modifying the program for a distributed environment could enable the simulation of more complex systems with a higher number of tasks and servers.

- **User Experience**: Improving the GUI for ease of use, allowing users to set parameters dynamically and view results in a more interactive format.

- **Scalability**: Refactoring the code to handle larger scale simulations more efficiently, possibly by improving the underlying data structures or parallelizing certain parts of the code.

- **In conclusion**, the Java program presented is an intricate simulation tool aimed at understanding and optimizing queuing systems. It leverages multithreading and synchronization to manage concurrent tasks across multiple servers. The architecture of the program employs robust data structures for safe and efficient task handling, sorting, and processing, reflecting the complexity of real-world queuing scenarios. Through the use of a graphical user interface, it also allows for real-time interaction and observation of the system's behavior. The ability to log and review simulation data further enhances its utility for analysis and strategic planning. This program is a valuable asset for anyone looking to improve the throughput and efficiency of service-oriented operations.

7. Bibliography

1. Creating Diagrams - https://app.diagrams.net/
2. Information about the project - https://dsrl.eu/courses/pt/
3. BlockingQueue Interface - https://www.geeksforgeeks.org/blockingqueue-interface-in-java/
4. AtomicInteger – https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicInteger.html