DOCUMENTATION

ASSIGNMENT 2

STUDENT NAME: Stoica Sergiu
GROUP: 30226

CONTENTS

1. Assignment Objective

The main objective of this assignment is to develop Design and implement an application aiming to analyze queuing-based systems by (1) simulating a series of N clients arriving for service, entering Q queues, waiting, being served, and finally leaving the queues, and (2) computing the average waiting time, average service time and peak hour.

Sub-objective	**Description**	**Documentation Section**
Analyze the problem and identify requirements	Identify functional and non-functional requirements for the queue management app.	Section 2: Problem Analysis, Scenario Modeling, Use Cases
Design the simulation application	Create an object-oriented design for the application including UML class and package diagrams.	Section 3: Design
Implement the simulation application	Develop the necessary classes and methods to handle more threads to run simultaneously.	Section 4: Implementation
User Interface Development	Design and implement an intuitive user interface that facilitates user interaction with the application.	Section 4: Implementation
Documentation	Prepare comprehensive documentation to detail the development process and support future maintenance and upgrades.	Entire Document

2. Problem Analysis, Modeling, Scenarios, Use Cases

Functional Requirements:

- The simulation application should allow users to set up the simulation.
- The simulation application should allow users to start the simulation.
- The system should calculate and write statistics such as average waiting time, average service time, and peak hours.
- The system should efficiently distribute incoming tasks among available servers based on the selected queue allocation strategy (e.g., shortest queue, shortest time).
- It must continuously monitor and log the status of each server and task, including wait times and service completion.
- The system must display the resulting polynomial after an operation in a standardized algebraic format.

Non - Functional Requirements:

- The simulation must run efficiently, ensuring all operations within the time limits are processed without delay to simulate real-time task handling.
- The user interface for the simulation control should be straightforward, allowing users to easily configure simulation parameters and initiate the process.
- The software architecture should be modular and scalable, supporting the easy addition of new features or changes, such as new queue allocation strategies or performance metrics.

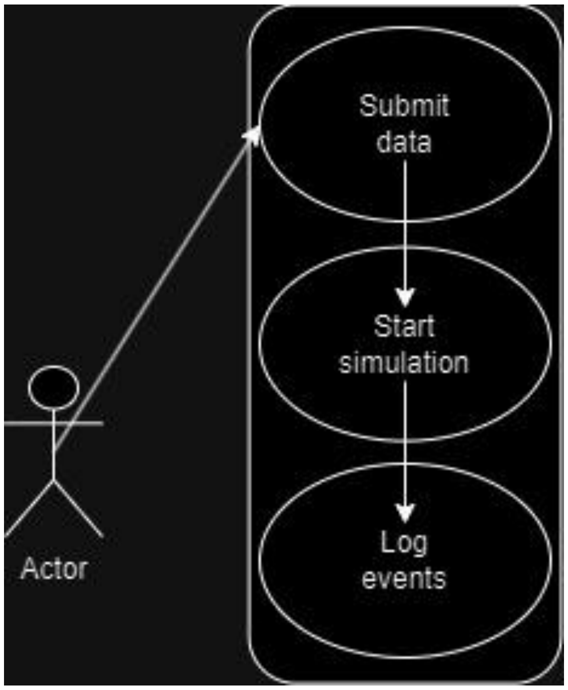
Use Cases:

- **Use Case: setup and run simulation.**
    - Actor: User
    - 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.
2. The user clicks on the submit input data button.
3. The application validates the data and starts the simulation.
4. The events are logged into the txt file.

    - Alternative scenario:

1. The user inserts invalid values for the application's setup parameters.
2. The application displays an error message and requests the user to insert valid values.
3. The scenario returns to step 1.



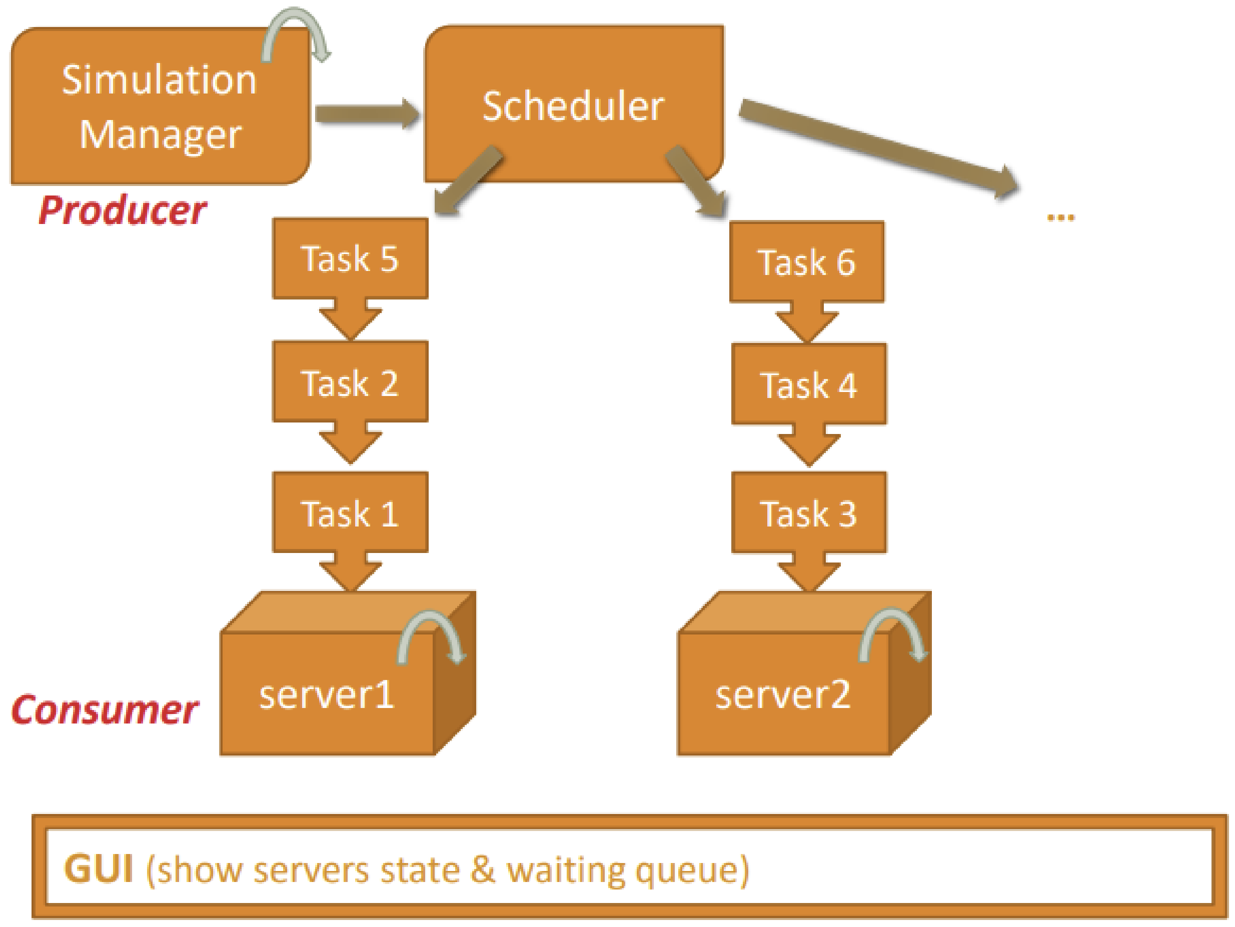
3. Design

Conceptual Architecture

The conceptual architecture of the queue management system is designed around a producer-consumer model, integrating a graphical user interface (GUI) for real-time interaction and monitoring.

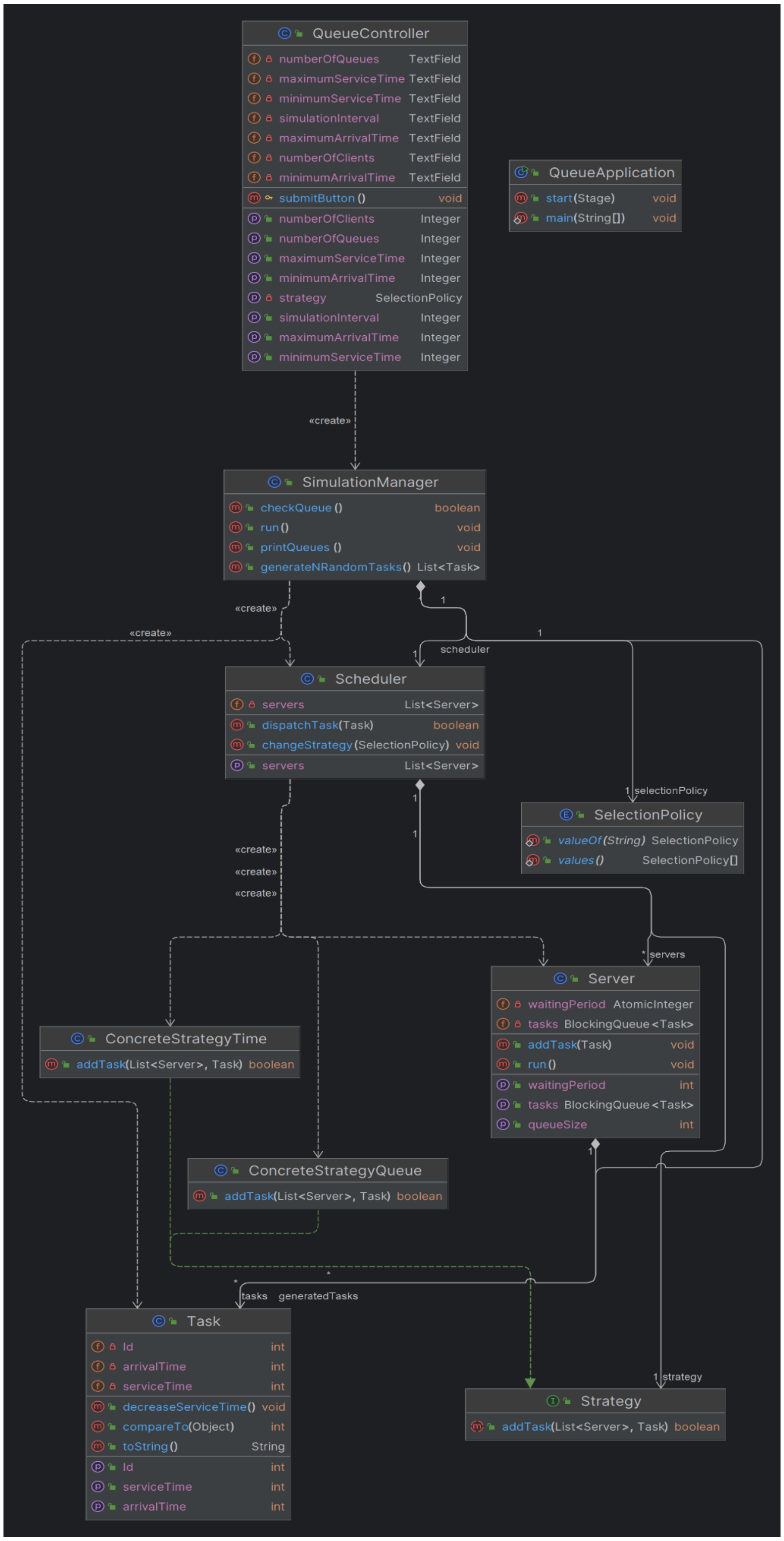
- **Simulation Manager (Producer):** This component is responsible for the generation and timing of tasks. It simulates the arrival of tasks into the system and serves as the producer in the architecture.
- **Scheduler:** Acting as an intermediary, the scheduler receives tasks from the Simulation Manager. It applies the queue allocation strategy to assign tasks to the appropriate server based on the system's current load and predefined criteria (e.g., shortest queue or shortest time).
- **Servers (Consumers):** Each server represents a consumer in the system. They process tasks assigned by the scheduler and simulate task execution. Servers run in parallel, each handling its queue of tasks independently.
- **GUI:** The graphical user interface provides a visual representation of the system's state. It displays the active servers, their current load, and the queue of waiting tasks. The GUI is updated in real-time to reflect the dynamic changes within the system, such as task allocation and completion, enabling users to monitor and interact with the simulation effectively.

This architecture facilitates a clear separation of concerns, with each component focusing on a specific aspect of the queue management system. It allows for efficient task distribution and processing while providing a user-friendly way to visualize and control the simulation.


Simulation Manager
Producer
Scheduler
...
Task 5
Task 2
Task 1
Task 6
Task 4
Task 3
Consumer
server1
server2
GUI (show servers state & waiting queue)

UML Package and Class Diagrams

- **View Package:**
    - **QueueApplication Class:** This class is responsible for rendering the graphical user interface (GUI) and presenting the application to the user. It's where the user interacts with the application.
- **Model Package:**
    - **Task Class**: Represents the individual tasks with properties such as ID, arrival time, and service time. It holds the data and logic for a task's lifecycle within the system.
    - **Server Class:** Symbolizes a server in the simulation. It manages a queue of tasks, processes them, and tracks the server's state (like idle or busy).
- **Controller Package:**
    - **QueueController Class:** Acts as a mediator between the View and the Business Logic. It takes input from the user via the GUI and translates it into actions to be performed by the **SimulationManager**.
- **Business Logic Package:**
    - **ConcreteStrategyTime Class:** Implements the Strategy interface for task allocation based on the shortest estimated time to completion.
    - **ConcreteStrategyQueue Class:** Implements the Strategy interface for task allocation based on the shortest queue.
    - **Scheduler Class:** Coordinates task distribution to servers based on the selected strategy. It maintains a collection of Server instances and routes tasks to them.
    - **SelectionPolicy Enumeration:** Defines the different strategies that can be applied by the Scheduler for task distribution (e.g., shortest queue, shortest time).
    - **SimulationManager Class:** Manages the simulation's execution, including task generation, time progression, and invoking the Scheduler.
    - **Strategy Interface:** Provides a contract for implementing various task allocation strategies by the Scheduler.


QueueController
numberOfQueues TextField
maximumServiceTime TextField
minimumServiceTime TextField
simulationInterval TextField
maximumArrivalTime TextField
numberOfClients TextField
minimumArrivalTime TextField
submitButton () void
numberOfClients Integer
numberOfQueues Integer
maximumServiceTime Integer
minimumArrivalTime Integer
strategy SelectionPolicy
simulationInterval Integer
maximumArrivalTime Integer
minimumServiceTime Integer
QueueApplication
start(Stage) void
main(String[]) void
«create»
SimulationManager
checkQueue () boolean
run () void
printQueues () void
generateNRandomTasks() List<Task>
«create»
«create»
1
1
scheduler
1
Scheduler
servers List<Server>
dispatchTask(Task) boolean
changeStrategy(SelectionPolicy) void
servers List<Server>
1 selectionPolicy
SelectionPolicy
valueOf(String) SelectionPolicy
values() SelectionPolicy[]
1
1
«create»
«create»
«create»
* servers
Server
waitingPeriod AtomicInteger
tasks BlockingQueue<Task>
addTask(Task) void
run() void
waitingPeriod int
tasks BlockingQueue<Task>
queueSize int
ConcreteStrategyTime
addTask(List<Server>, Task) boolean
ConcreteStrategyQueue
addTask(List<Server>, Task) boolean
1
*
*
tasks generatedTasks
Task
Id int
arrivalTime int
serviceTime int
decreaseServiceTime() void
compareTo(Object) int
toString() String
Id int
serviceTime int
arrivalTime int
1 strategy
Strategy
addTask(List<Server>, Task) boolean

Data Structures

- **BlockingQueue<Task>:**
    1. Used within the Server class to hold the tasks that are queued for processing.
    2. A thread-safe queue implementation is essential for concurrent handling of tasks since servers run on separate threads.
    3. Ensures that task insertion and retrieval operations are atomic and prevent race conditions.

- **AtomicInteger:**

4. Utilized in the Server class for managing the waiting period, which is the cumulative time that tasks will take to process.
5. It allows for thread-safe increment and decrement operations, which are necessary as multiple threads may modify this value concurrently.

- **ArrayList<Task>:**
    1. In the SimulationManager class, an ArrayList holds the generated tasks before they are dispatched to the servers.
    2. It provides O(1) time complexity for adding tasks and efficient sorting based on task arrival times.

- **List<Server>:**
    1. In the Scheduler class, a list of Server instances represents the various queues (or servers) available for processing tasks.
    2. It provides easy iteration over servers when distributing tasks according to the chosen strategy.

- **Enum (SelectionPolicy):**
    1. Defines the constants for the different queue allocation strategies.
    2. An enumeration is used for type safety and a fixed set of possible values that represent the different strategies implemented.

4. Implementation

Simulation Manager

The **SimulationManager** is responsible for the generation of tasks and orchestrating the simulation's flow. It acts as the producer of tasks which are then distributed by the **Scheduler**.

```java
public List<Task> generateNRandomTasks() {
    generatedTasks = new ArrayList<>();

    for (int i = 0; i < numberOfClients; i++) {
        int processingTime = (int) (Math.random() * (maxProcessingTime - 
minProcessingTime + 1)) + minProcessingTime;
        int arrivalTime = (int) (Math.random() * (maxArrivalTime - 
minArrivalTime + 1)) + minArrivalTime;;
        generatedTasks.add(new Task(i+1, arrivalTime, processingTime));
        averWaitingTime += processingTime;
    }

    this.averServiceTime = this.averWaitingTime / (float) 
this.numberOfClients;
    this.averWaitingTime /= (float) (this.numberOfClients * 
this.numberOfServers);

    Collections.sort(generatedTasks);
    return generatedTasks;
}
```

This method generates a list of random **Task** objects that represent the tasks to be processed during the simulation. It iterates based on the number of clients you want to simulate, creating tasks with random arrival and processing times within the specified bounds. After creating all tasks, it calculates the average service time and average waiting time. The generated list is then sorted by arrival time and returned.

```
public boolean checkQueue(){
    boolean ok = true;

    if(!generatedTasks.isEmpty())
        return true;
    else {
        for (int i = 0; i < scheduler.getServers().size(); i++) {
            Server server = scheduler.getServers().get(i);
            BlockingQueue<Task> tasks = new ArrayBlockingQueue<>(10000);
            tasks.addAll(server.getTasks());

            if (!tasks.isEmpty()) {
                return true;
            } else {
                ok = false;
            }
        }
    }
    return ok;
}
```

checkQueue()

The **checkQueue()** method determines whether the simulation should continue running. It checks if there are still tasks in the **generatedTasks** list that haven't been processed. If this list is empty, it then checks each server's task queue to see if they are all empty. If at least one queue still has tasks, it returns **true**, indicating there are still tasks to be processed.

```
for(int i = 0; i < scheduler.getServers().size(); i++){
    clientsPerHour +=scheduler.getServers().get(i).getQueueSize();
}

if(clientsPerHour > maxClientsPerHour){
    maxClientsPerHour = clientsPerHour;
    peakHour = currentTime;
}
```

This block of code accumulates the total number of clients across all servers within the current hour (**clientsPerHour**). It then checks if the number of clients served in the current hour surpasses the maximum number recorded (**maxClientsPerHour**). If so, it updates this maximum and records the current time as the new **peakHour**. This peak hour tracking is essential for understanding the busiest period in the simulation, which can help in analyzing the system's performance and for potential optimizations.

```java
public void printQueues()
{
    for(int i=0; i<scheduler.getServers().size(); i++)
    {
        Server server =scheduler.getServers().get(i);
        BlockingQueue<Task> tasks = new ArrayBlockingQueue<>(10000);

        tasks.addAll(server.getTasks());

        if(tasks.isEmpty()){
            try {
                fileWriter.append("\nQueue " + (i + 1) + " : closed");
                System.out.print("\nQueue " + (i + 1) + " : closed");
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
        } else {
            try {
                fileWriter.append("\nQueue " + (i + 1) + " : ");
                System.out.print("\nQueue " + (i + 1) + " : " );
                for(Task t : tasks){
                    fileWriter.append(t.toString() +";");
                    System.out.print(t.toString() +";");
                }
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
        }
    }
    try{
        fileWriter.append("\n");
        System.out.print("\n");
    }catch (IOException e){
        throw new RuntimeException(e);
    }
}
```

The **printQueues()** method iterates over all servers in the **Scheduler**, printing out the current state of each server's task queue. If the server's queue is empty, it prints that the queue is closed. Otherwise, it prints out the tasks currently in the queue. This is used for both debugging and providing real-time updates within the simulation's GUI or output logs.

```
Iterator<Task> ListTasks = generatedTasks.iterator();
while (ListTasks.hasNext()) {
    Task t = ListTasks.next();
    if (t.getArrivalTime() == currentTime) {
        if (scheduler.dispatchTask(t)) {
            ListTasks.remove();
        }
        int waitTimeForThisTask = currentTime - t.getArrivalTime();
        averWaitingTime += waitTimeForThisTask;
        completedTasks++;
    }
}
```

This loop is iterating through the list of generated tasks and handling the ones that are scheduled to start at the current time.

Scheduler

The Scheduler manages a list of Server instances and delegates tasks to them based on the selected strategy.

```
public boolean dispatchTask(Task t) {
    if (strategy.addTask(servers, t)) {
        return true;
    } else {
        return false;
    }
}
```

This method is responsible for dispatching tasks to servers. It uses the current strategy object to determine which server should process the task.

- **If the strategy successfully adds the task to a server**, it returns **true**. This could mean the task has been placed in a server's queue.
- **If the strategy does not add the task**, it returns **false**. This could mean all servers are at capacity, or there is some other reason the task could not be dispatched.

```
public void changeStrategy(SelectionPolicy policy) {
    switch (policy) {
        case SHORTEST_QUEUE:
            strategy = new ConcreteStrategyQueue();
            break;
        case SHORTEST_TIME:
            strategy = new ConcreteStrategyTime();
            break;
        }
}
```

This method ensures that the **Scheduler** can adapt to different strategies at runtime without needing to be recreated.

```
public Scheduler(int maxNoServers, int maxTasksPerServer) {
    servers = new ArrayList<>(maxNoServers);

    for (int i = 0; i < maxNoServers; i++) {
        Server server = new Server();
        servers.add(server);
        new Thread(server).start(); // Starting the server thread
    }
}
```

This constructor initializes the **Scheduler** with a specified number of servers.

- It creates an **ArrayList** to hold the server instances, sized to the maximum number of servers allowed.
- Then, it enters a loop to instantiate the specified number of **Server** objects.
- Each server is added to the list and then started as a separate thread.

Server

Each **Server** represents a consumer in the producer-consumer model. It processes tasks in a queue.

```
public void addTask(Task newTask) {
    tasks.add(newTask);
    waitingPeriod.addAndGet(newTask.getServiceTime());
}
```

This method is called to add a new task to the server's queue:

- **tasks.add(newTask)**: Adds the new task to the **tasks** queue. The **tasks** variable is an instance of a thread-safe queue implementation such as **BlockingQueue**.

  ```
  tasks = new ArrayBlockingQueue<>(1000);
  ```

- **waitingPeriod.addAndGet(newTask.getServiceTime())**: Increments the **waitingPeriod** by the service time of the new task. **waitingPeriod** is an **AtomicInteger**, ensuring thread-safe operations. The method **addAndGet** atomically adds the value and returns the updated value.

```
try {
    if(!tasks.isEmpty()) {
        Task nextTask = tasks.peek();
        if(nextTask.getServiceTime()==0){
            nextTask= tasks.take();
        }
        Thread.sleep(1000);
        nextTask.decreaseServiceTime();
        waitingPeriod.decrementAndGet();
    }
} catch (InterruptedException e) {
    Thread.currentThread().interrupt();
    break;
}
```

This block is part of the **run** method's loop that's continuously processing tasks:

Checks if there are tasks in the queue. Retrieves, without removing, the next task in the queue. Checks if the task at the head of the queue has a service time of zero, implying it's finished and can be removed. If the task is completed (service time is zero), it is removed from the queue. Decreases the service time of the current task, indicating that processing is taking place. Decrements the total waiting period of the server, which keeps track of the time tasks are waiting to be processed or are being processed.

If the thread running this server is interrupted (perhaps due to the simulation ending or other external factors), an **InterruptedException** is thrown. The **catch** block then handles this exception by interrupting the current thread (**Thread.currentThread().interrupt()**).

5. Results

The queue management system was tested across three different scenarios to evaluate its performance under varying workloads. These scenarios were designed to mimic real-world applications ranging from small to large scale systems, reflecting the diversity in client demand and server availability.

Test 1: Small-scale Simulation

- Parameters: **N = 4**, **Q = 2**, **t_simulation = 60 seconds**, **t_ arrival = [2, 30]**,

  t_service = [2, 4].
- The log of events can be seen in Log_1.txt.

Test 2: Medium-scale Simulation

- Parameters: **N = 50**, **Q = 5**, **t_simulation = 60 seconds**, **t _arrival = [2, 40]**,

  t_service = [1, 7].
- The log of events can be seen in Log_2.txt.

Test 3: Large-scale, High-load Simulation

- Parameters: **N = 1000**, **Q = 20**, **t_simulation = 200 seconds**, **t_ arrival = [10, 100]**,

  t_ service = [3, 9].
- The log of events can be seen in Log_3.txt.

Average Waiting Time: Increased with the number of clients and decreased with more servers. Test 3 had the highest average waiting time due to the broad range of arrival times.

Server Utilization: In Test 1 and Test 2, servers were relatively underutilized, while Test 3 showed more balanced usage due to higher demand.

Peak Times: Were clearly observable in Test 3, where the queue lengths were substantially longer, indicating stress periods that could benefit from additional servers.

Comparison Between Tests: Revealed that as **N** increases, without a proportional increase in **Q**, system performance in terms of waiting time and server utilization becomes suboptimal.

6. Conclusions

The queue management system developed in this project is a detailed simulation of a multi-server environment, utilizing principles from object-oriented programming and ways to control tasks that run at the same time. By effectively managing the creation and distribution of tasks through the SimulationManager, and using the Scheduler to ensure tasks are evenly distributed across several servers with different strategies, the system runs smoothly. The Server class acts as an independent unit that safely handles tasks, simulates the time each task takes, and keeps the task queue organized.

What I have learned:

1. **Concurrency Handling:** We gained valuable experience in managing concurrency within a software system. This involved understanding and implementing thread-safe operations to prevent data races and ensure that multiple servers could handle tasks simultaneously without error.
2. **Design Patterns:** The use of design patterns such as the Strategy Pattern for the Scheduler enhanced our ability to design flexible and scalable software. This pattern allowed us to easily swap out task allocation strategies based on different operational requirements or performance considerations.

This project deepens the understanding of queue management systems, showing their crucial role in many real-life situations where managing resources efficiently and saving time are important.

7. Bibliography

w3schools - Java Threads

https://www.w3schools.com/java/java_threads.asp

Geek for geeks - Java Threads

https://www.geeksforgeeks.org/java-threads/

simplilearn - An Introduction to Thread in Java

https://www.simplilearn.com/tutorials/java-tutorial/thread-in-java